This is to certify that the

thesis entitled

An Object-Oriented Framework
for Modeling Dynamic Connections

presented by

Jonathan Robert Engelsma

has been accepted towards fulfillment
of the requirements for

Ph.D. degree in Computer Science

Major professor

Date May 13, 1993

AN OBJECT-ORIENTED FRAMEWORK FOR MODELING DYNAMIC CONNECTIVITY

By

Jonathan Robert Engelsma

A DISSERTATION

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

DOCTOR OF PHILOSOPHY

Computer Science Department

1993

ABSTRACT

AN OBJECT-ORIENTED FRAMEWORK FOR MODELING DYNAMIC CONNECTIVITY

By

Jonathan Robert Engelsma

Science and engineering rely on resolving systems into constituent components whenever possible to simplify the analysis and design process. This "divide and conquer" approach is useful in many of the problems commonly encountered. A system can be divided into components if it can be assumed that: points, edges, surfaces, and regions can be identified where connectivity relations can be reasonably established; and the constitutive equations within each of these interconnected components can be expressed in a localized manner involving only the variables of interconnections and other local variables. The resolution of systems into components is not only a common basis of the design processes that are used, but is at the heart of effective computer-based modeling and simulation.

The advancement to the state of computer-based modeling and simulation of systems presented in this dissertation deals with systems comprised of components whose characteristics, existence, and interconnections vary over time. A simulation framework based on recent results in object-oriented design is proposed. The framework

defines a set of abstractions that serve as building blocks for describing connections, components and aggregations of components. The framework incorporates a set of concurrent objects which cooperatively respond to changes in the model topology and component population during model execution.

The construction of hierarchical models consisting of both continuous and discrete components is supported. The behavior of a component is represented by mathematical equations, and/or by blocks of code that are executed when a component sends or receives messages. Equations representing the connections among components are formulated by the framework when new connections occur and are revoked when the connections are removed. The equations representing connections and the equations representing component behavior are organized into systems of equations and solved on a demand basis.

To my wife Mietje

and my parents James and Marie Engelsma

ACKNOWLEDGMENTS

I wish to thank my thesis advisor, Professor Richard Reid for his guidance throughout my graduate studies at Michigan State University. His encouragement, insight, patience and ability to motivate students, proved to be most valuable.

I am grateful for the helpful suggestions provided by my guidance committee: Professors Phillip Mckinley, Matt Mutka, and Chi Lo. I am also grateful for the suggestions of Professor Thomas Manetsch.

I am indebted to the Science Advisory Board Associates of Motorola Inc. who provided the funding and computer equipment necessary for this research. In particular I thank Dr. Ronald Borgstahl who was our representative at Motorola. Without his efforts this research would not have been possible. I thank Dr. Patrick Reilly who followed my progress closely and provided many useful comments and criticisms along the way.

I thank my friends and fellow graduate students Barb Czerny, Christian Trefftz and Reid Baldwin for the time they spent reading and commenting on this dissertation. I am also grateful for the selfless efforts of J. Daniel Smith, who spent many hours installing and maintaining our document processing software.

Finally, I wish to thank my ever-supportive wife Mietje, and our parents James and Marie Engelsma and Willem and Lydia Henke. Their love and support played an important role in this work.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

Introduction and Motivation

Using the digital computer as a modeling and simulation tool has allowed researchers to investigate ideas and find solutions to problems which before were unapproachable, or at least daunting enough temporarily to discourage more in-depth study. Computer-based modeling enhances problem solving in several ways. First, computer models are generally easier to construct, modify, and execute than traditional pencil and paper analytical models. Second, to obtain information about a system, it is usually less costly (and safer in certain cases) to simulate the system with a computer rather than obtaining the information from the real system. Along a similar vein, Jain [48], Cellier [19], and others have pointed out that often the system being modeled is not available to the model developer. Finally, a simulation model can be used repeatedly to study a particular system or family of systems. However, as noted by Banks and Carson [6], computer-based modeling is not without its disadvantages. First, researchers will sometimes develop a computer simulation model when in fact it is quite possible to derive an analytical model. This is only a disadvantage when the analytical model would give better results, without making assumptions that significantly reduce the viability of the model's representation of the real world. Second, large and involved computer-based models may be very costly to implement, validate and maintain. Third, computer-based models may require extensive amounts of com-

putation. The latter two disadvantages have been the focus of current research in the area of computer simulation. Researchers have begun to investigate visual and interactive simulation, object-oriented simulation, and parallel simulation, in an attempt to reduce the costs associated with the development and execution of computer-based models.

Object-oriented design concepts and visual and interactive simulation techniques have been adapted in an attempt to reduce the costs associated with implementation and validation, and to increase model lucidity. Using object-oriented methodologies allows models to be constructed that more closely represent the real system. For example, in a model of a physical system, each discrete part in the system may be mapped to a corresponding object in the model. Equally important is the fact that the object-oriented approach greatly enhances reusability. Libraries of object classes may be developed and maintained and reused whenever possible. Object-oriented *frameworks*[1] allow entire designs (abstract classes and the relationships among them) to be reused. Research in visual and interactive simulation builds upon the notion that "a picture is worth a thousand words". Sophisticated graphical renditions, both static and animated, can be produced during simulation, giving the user the capability to observe firsthand the behavior of the modeled system. In addition, some modeling tools allow users to modify system parameters during model execution and watch the effects of the parameter adjustments.

Distributed simulation methodologies have been studied in hopes of reducing the long execution times. The basic idea is to distribute the work over a set of processors operating in parallel. As pointed out by Fujimoto [40], it is interesting to note that although simulation is a problem domain which contains substantial amounts of parallelism, it is very difficult to parallelize. If the times required to process the events

[1]Italics are used in this chapter to indicate words which have a precise meaning in the context of this dissertation. These words will be elaborated on eventually, but for now meanings can be assumed from common English usage.

in the simulation model were proportional to the times required for the events in the real world this paradox would not occur since every process in the simulation would be automatically synchronized in time. However, this is not the case. Hence, it is the management of simulation time which is at the heart of research in parallel and distributed simulation. There are two main philosophies for managing time in a parallel event simulation, the conservative approach [23] and the optimistic approach [49]. Almost all the methodologies for parallel event-driven simulation employ one of these two philosophies or a combination of the two.

1.1 Problem Statement

This dissertation is concerned with modeling a certain class of systems - systems comprised of *components* whose characteristics, existence and *connections* vary over time.

A component with varying characteristics, that is, a component which has different regions of operation based on its variable values, is illustrated by an electronic diode. Ideally, a diode has two regions of operation: zero resistance or infinite resistance. The resistance characterizing the diode at any given time is decided by the polarity of the input potential at that time.

In a mobile cellular phone system the power up/down of mobile phone units exemplify components whose existence is dynamic. Similarly, on a multiprogrammed computer system, software processes which can be spawned and terminated are also examples of components with dynamic existence. Dynamic connectivity among components in the mobile cellular phone system is evident when considering the relationships among mobile phone units and cell base stations. Another example of a system with dynamically connected components is a software system which has been implemented as a set of communicating processes. The processes are running on a

set of workstations which are interconnected on a network. In such a system, the components would be the processes and the connections would represent the software connections supported by the network protocol. Interconnections can be more than simply a logical or intangible type. A fault tolerant computer system in which live insertion of circuit boards is permitted, or a mechanical system made up of movable components illustrate cases in which the dynamic connections between components are of a physical nature. In the examples cited so far, dynamic connections are considered to be natural phenomena in the ongoing operation of the system. Dynamic connections of a more artificial nature are introduced by visual and interactive simulations and virtual reality. In these systems, components and connections may be inserted into or deleted from a model at the whim of the user.

This dissertation argues that systems whose components and connections may vary (or are allowed to vary) over time can be successfully modeled, based on the following ideas:

- the development of an abstract, object-oriented framework upon which simulation tools for a wide variety of systems can be constructed. Unlike traditional simulation methodologies, this framework provides for explicitly modeling the dynamic existence of components and the connections among them. The idea of object-oriented frameworks applied to simulation tools is synonymous with the idea of graphical user interface kits applied to general computer applications. The framework captures the essence of components, connections, and hierarchy, and allows the simulation tool designer to focus on the higher level issues associated with the problem at hand.

- the design of a simulation framework which is amenable to parallel implementation. The framework presented in this dissertation is based on a suite of concurrent objects which concurrently manage the time varying connections

and component population.

- the unified treatment of components and connections in the context of both continuous and discrete simulation models.

These ideas are developed in the context of FMDC (Framework for Modeling Dynamic Connections), an object-oriented framework for modeling dynamic connectivity that has been implemented during the course of this research project.

1.2 Organization of this Dissertation

The remainder of this dissertation is organized as follows. Chapter 2 examines the results appearing in the literature that are related to this work. The strengths and weaknesses of several general simulation methodologies are examined. Recent developments in object-oriented frameworks are also discussed.

In Chapter 3 the representations and definitions of the simulation framework are developed. The chapter presents each of the four layers that form the simulation framework. The chapter concludes with an example model that ties the presented concepts together.

Object-oriented concurrency in the context of the simulation framework is discussed in Chapter 4. A second topic developed in Chapter 4 is the framework's underlying support for discrete models.

Chapter 5 develops the simulation and computational layers of the framework in detail. The chapter begins by examining the details of component and connection management. This is followed by the development of the framework's mechanisms for equation formulation, accumulation, and solution.

In Chapter 6 the salient features of the framework are demonstrated with models from several different problem domains. A model of an electrical system which incorporates discrete digital components as well as analog components demonstrates

its ability to represent both continuous and discrete components in one model. Other examples demonstrate its capability to explicitly model systems where components and connections vary over time.

Finally, Chapter 7 summarizes the contributions made and makes recommendations for future research.

An appendix has been included that gives the interfaces of the abstract classes defined by the framework.

CHAPTER 2

Background

A common characteristic of many systems under study today is complexity. This complexity is due in part to the size of the systems, but more significantly it is due to the interactions among their constitutive subsystems. Modern digital computers play an important role in helping researchers model and understand the complexities encountered. One aspect of modeling that has been given very little attention until now is how to represent changing relationships among a model's components explicitly. This lack of attention can be attributed to several reasons. First, it wasn't necessary. In the past, most systems being studied could be represented with static block diagrams. Topology changes were relatively uncommon and most existing modeling tools could be manipulated to handle occasional topological changes. Second, the limitations of the computer technology in terms of memory, processing power and software inhibited the development of simulation tools which explicitly dealt with changing relationships among components. Finally, simulation models were typically executed in a batch mode. Recent developments in visual-interactive simulation and virtual reality increase the utility for a methodology that allows dynamic relationships among components.

This chapter presents the developments from the literature that have influenced the contributions made in this dissertation. An important facet of this dissertation is

its use of recent developments in object-oriented systems research [99] in the area of systems simulation. After the object-oriented material, the developments in systems simulation which had the greatest impact on this dissertation are examined. Some of the results reviewed here approach the problem of modeling dynamic connectivity in a very narrow sense, while the others do not deal with it all and have been included mainly for their influence on related issues.

2.1 Object-Oriented Modeling/Simulation

Although programming languages such as Smalltalk [44] and C++ [86] are typically associated with object-oriented systems, it is interesting to note that object-oriented concepts were first introduced in the simulation language Simula-67 [29]. Traditionally, software developers concentrated on the various operations or procedures which must be carried out. When using an object-oriented approach, developers concentrate on defining the entities (objects) and the interactions among them. In this way the object-oriented approach more closely represents the real world. Since simulation involves real world representations, object-oriented design is especially applicable.

There has been some confusion regarding the terms *object-oriented modeling* and *object-oriented simulation* as pointed out by Cellier *et al.* [22]. The term *modeling* refers to the process of creating a model, while the term *simulation* refers to computing the trajectory of various model variables over time. Thus, object-oriented modeling refers to the process of creating the model using the object-oriented approach, and object-oriented simulation refers to the actual execution of the model employing the object-oriented approach. It has been argued by Cellier *et al.* that the latter is to be avoided, especially for continuous system models. The argument is based on the fact that it is inefficient to have two separate software entities (objects) communicate at least once for every integration step. In the literature and in the remainder of this

dissertation, both terms are used synonymously and imply the definition given above for object-oriented modeling.

2.1.1 Object-Oriented Concepts

The key concepts in object-oriented design are: *objects, classes, inheritance, polymorphism,* and *dynamic binding.* There are many books and articles written on these concepts. Korson and McGregor have provided a fairly complete overview of the object-oriented paradigm [54], as has Booch [12].

An *object* is an actual entity in an object-oriented system. Typically, the object only exists during execution. However, the concept of *persistent objects* has received a great deal of interest lately, especially in the area of object-oriented databases. A persistent object is one whose state can be stored on secondary storage and restored into primary memory when necessary.

A *class* is a template or description of an object. It describes what data and procedures are associated with a particular object. In object-oriented languages the class is thought of as a user-defined type. An object is an *instantiation* of a class. An *abstract class* is a class which cannot be instantiated. Such classes are usually used to abstract the general properties of an entity and force a standard interface for the classes which will inherit its interface. Any class that is not abstract is referred to as either a *concrete class* or as a *class.*

The notion of *inheritance* refers to the "kind of" relationship among classes. Classes can be arranged in hierarchies where the definition of one class is based on the definition of other classes. A relationship in which the definition of a single class is dependent on the definition of two or more other classes is referred to as *multiple inheritance.*

Polymorphism and *dynamic binding* are closely related. Polymorphism refers to the capability of a variable to denote objects of more than one class. Each of these

objects may respond to a common operation in a different manner. Dynamic binding means that the code that will be executed for a particular method is determined by the type of the object at the time of execution.

2.1.2 Recent Research in Object-Orientation - Frameworks

One of the goals of the object-oriented paradigm is to help develop reusable software components. For general purpose programming there are a wide variety of class libraries available which implement standard program components such as stacks, queues, heaps, trees, numerical methods, etc. There are also simulation libraries which provide libraries of simulation components such as resources, delays, priority queues, etc. Typical examples of simulation class libraries include DISC++ [11], SIM++ [59], and PRISM [93].

In addition to reusing software components, researchers have begun to look at ways to reuse designs. One particular approach to reusing designs being promoted by Johnson and colleagues, is the *object-oriented framework* [50, 99]. The term "framework" has been used frequently in many different contexts. In object-oriented design, researchers define a framework to be a set of abstract and concrete classes and the interfaces among them. Instead of reusing single components as is done in the case of class libraries, a framework allows the designer to capture the generalities of a large class of related applications into one abstract framework. To develop a new application, new (more specific) classes are derived from the classes in the framework. A good framework allows new applications to be developed very rapidly.

Frameworks are not easy to develop. Developing a good framework is equivalent to developing a theory - lack of generality will severely limit its applications. Typically, designing a framework is an iterative task that requires several applications to be developed before the significant generalities can be abstracted.

There have been a number of successful frameworks developed. An objected-

oriented operating system called *Choices* has been developed at University of Illinois [18]. *Choices* is actually a conglomerate of several frameworks. *Interviews*, a framework for building graphical user interfaces, has been developed by Linton *et al.* at Stanford University [58]. In an attempt to bring the functionality of the user interface building blocks bundled with the Macintosh computer to UNIX based workstations, Weinand *et al.* of the University of Zurich have developed a framework called *E++* [98]. *VAMP*, developed by the Aldus Corporation, is another example of a framework for developing user interfaces [38]. Researchers at Berkeley have developed an object-oriented framework for mixed-paradigm simulation [16, 17, 72]. These results will be discussed in detail in the next section.

2.2 Results from the Simulation Field

This section examines more directly the results in the simulation field that influenced the work presented in this dissertation.

2.2.1 Object-Oriented Frameworks in Simulation

Notable results in applying object-oriented frameworks to the development of simulation tools have been obtained by Buck *et al.* at Berkeley [16, 17, 72]. Their research has led to the development of a mixed-paradigm simulation/prototyping framework in C++ called Ptolemy. By defining a set of internal object-oriented interfaces, Ptolemy provides a large set of building blocks that greatly reduce the effort of building new simulation and prototyping environments. The key objective of their work is to allow many different computational models to coexist in the same simulation. The basic module in Ptolemy is the *Block*. A *Block* is a module of code which can be invoked at run-time. When invoked a *Block* consumes the data present at its input *PortHoles* and generates data on its output *PortHoles*. The data objects being communicated

are derived from the abstract class *Particle.* A *Galaxy* is a class that contains a collection of interconnected blocks, and a *Universe* is a special *Block* that contains a *Galaxy* and a *Scheduler.* The *Scheduler* is the entity that decides the order of execution among the *Blocks.* An atomic *Block* is called a *Star.* A special class, *Wormhole*, is introduced that allows the mixture of multiple domains (a domain is defined to be the combination of a scheduler and a set of blocks that conform to the behavior expected by this scheduler.) Figure 2.1 illustrates the inheritance diagram among Ptolemy's classes. The XXX in the figure represent the classes derived for a fictitious domain XXX.

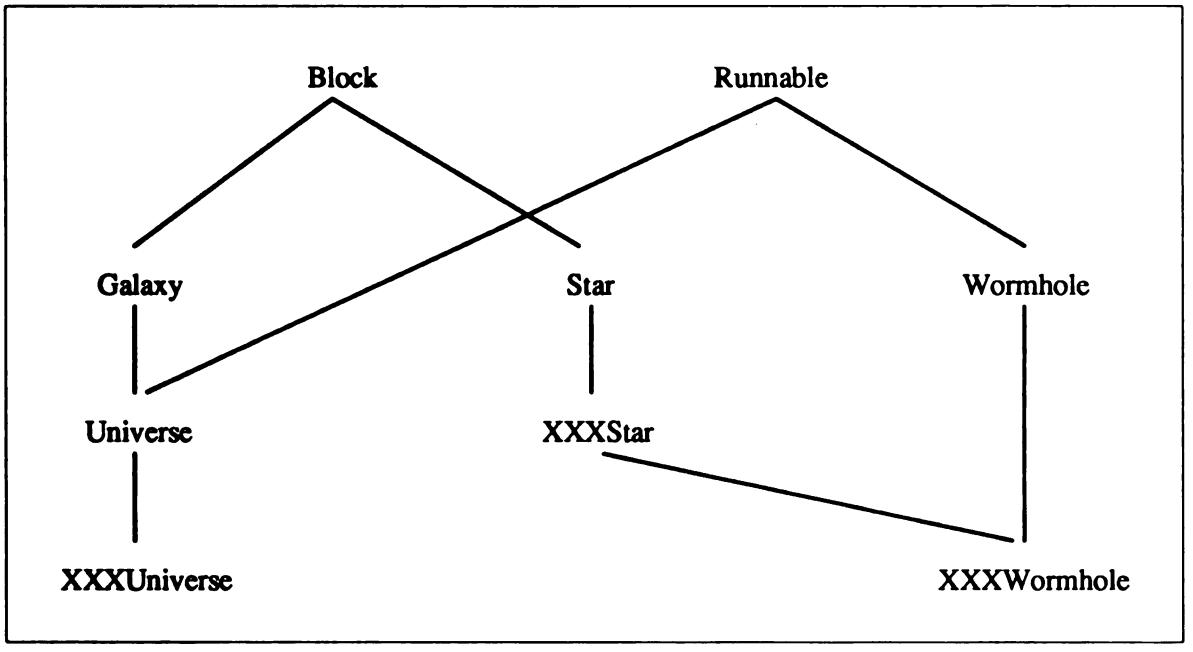

Figure 2.1. The Ptolemy inheritance relationships.

Ptolemy has been used to construct simulation tools in domains such as networking and transport, call-processing and signaling software, embedded microcontrollers, signal processing and others. The Ptolemy framework is quite general. The FMDC

framework differs in that it provides explicit support for generating and maintaining mathematical representations of the relationships among components. Such support is useful in a wide variety of domains and particularly useful when dealing with dynamic connectivity, as will be demonstrated in the chapters that follow.

The flattening of the hierarchies imposed by *Galaxy* classes within certain domains by the associated *Scheduler* may or may not hinder an efficient implementation of the framework on a parallel machine. The approach in the FMDC framework was to preserve the natural hierarchy present in a model during execution.

2.2.2 Results From Interactive/Visual Simulation

PRISM [67, 68, 95, 93, 94], an object-oriented environment for discrete-event driven simulation developed by researchers at Microelectronics and Computer Technology Corporation, integrates several innovative ideas. PRISM consists of a set of standard queuing level primitives such as service nodes, delay nodes, resource pools, etc. Each of these nodes is a class which can be inherited to develop more specialized primitives. There is a container type class that allows hierarchical models to be developed. Since PRISM started out as a research exercise in defining what an interactive extensible modeling environment is, its primary contributions are in the area of visual interactive simulation. A graphical interface was developed which supports graphical construction of models and automatic animation of the simulation during execution. Graphical plotters and summation/integration primitives are used for presentation of variable trajectories during execution. A simulation can be paused, modified and resumed during execution. The following modifications are supported when the simulation is paused:

1. model parameters can be changed.

2. new components can be added to the running model.

3. existing components can be removed from the running model.

4. new behaviors can be loaded for custom primitives.

5. connections among components can be added or removed.

The modification of model parameters is aided by what is referred to as a *symbolic spreadsheet.* In essence this concept is identical to that of the programs run on personal computers to balance checkbooks. The only difference is that the cells, which are actually model parameters or attributes of primitives in the models, are addressed in a hierarchical symbolic format rather than the typical row/column format. Complex expressions and even programs can be associated with a parameter causing it to automatically execute when dependent data is changed interactively during simulation. There has been some reservations in the simulation community regarding parameter modification during simulation [63]. Concerns have been raised regarding the danger of misinterpreting summary statistics when steady state has not been achieved after interactions.

PRISM also incorporates *dynamic linking.* This allows new components to be added to a running model, or existing components to be removed or modified. Dynamic linking refers to the capability of linking new object code into a running program. This concept has been used by debuggers and other object-oriented systems before, but PRISM demonstrates its usefulness in simulation programs. Its usefulness stems from the facts that modeling is typically an iterative activity, and that simulation models can be quite large. Dynamic linking makes it possible to make a modification to a model and only recompile and relink a small part of the model.

Of most interest in the context of this dissertation, is PRISM's support of dynamic connectivity. A connection in PRISM is quite straightforward. When components (which are objects in the object-oriented sense) need to interact in some manner, they simply call each other's member functions. Hence, making a connection in

this context can be accomplished by recording information within the component which tell it the components with which it will communicate. Type checking is done to insure that the ports of the components being connected are compatible. This approach is suitable when a connection indicates that data (called a *Transaction* in PRISM terminology) may be transferred among the components involved. Since PRISM supports a queuing level paradigm and has been used primarily for high-level modeling and prototyping of computer hardware and software this approach has been sufficient. This dissertation however, makes more general assumptions about what a connection is. That is, in addition to representing the potential for data to be passed from one component to another, connections may also establish mathematical relationships among components. Automatic formulation and resolution of these mathematical relationships coupled with the transaction passing approach PRISM takes, results in a more flexible framework that has a greater problem solving scope. PRISM's notion of a connection limits its ability to model the relationships among components considered in this dissertation.

2.2.3 Sim

The work done by Reid [75] in the area of computer systems modeling has served as a springboard for the contributions this dissertation makes. His simulator, conveniently referred to as *Sim* has been used extensively for pedagogical purposes in computer architecture courses. Users assemble a textual description of components and their interconnections. An animated rendition of the model is generated automatically during simulation. Primitive input components such as switches and pulsers are supported, which allow some degree of interaction during simulation execution. The simulator allows users to combine groups of components into submodels which appear in the graphical display as a box. If desired, a user can zoom into a submodel and view its constitutive components. Sim provides a large library of parts commonly used in

the design of electronic systems. Models such as the 8 bit microprocessor depicted in Figure 2.2 can be developed and studied without much difficulty. *Sim* does not support dynamic connections. Its main influence in the context of this dissertation is its notions of components and submodels, and its ability to incorporate hierarchy into designs.

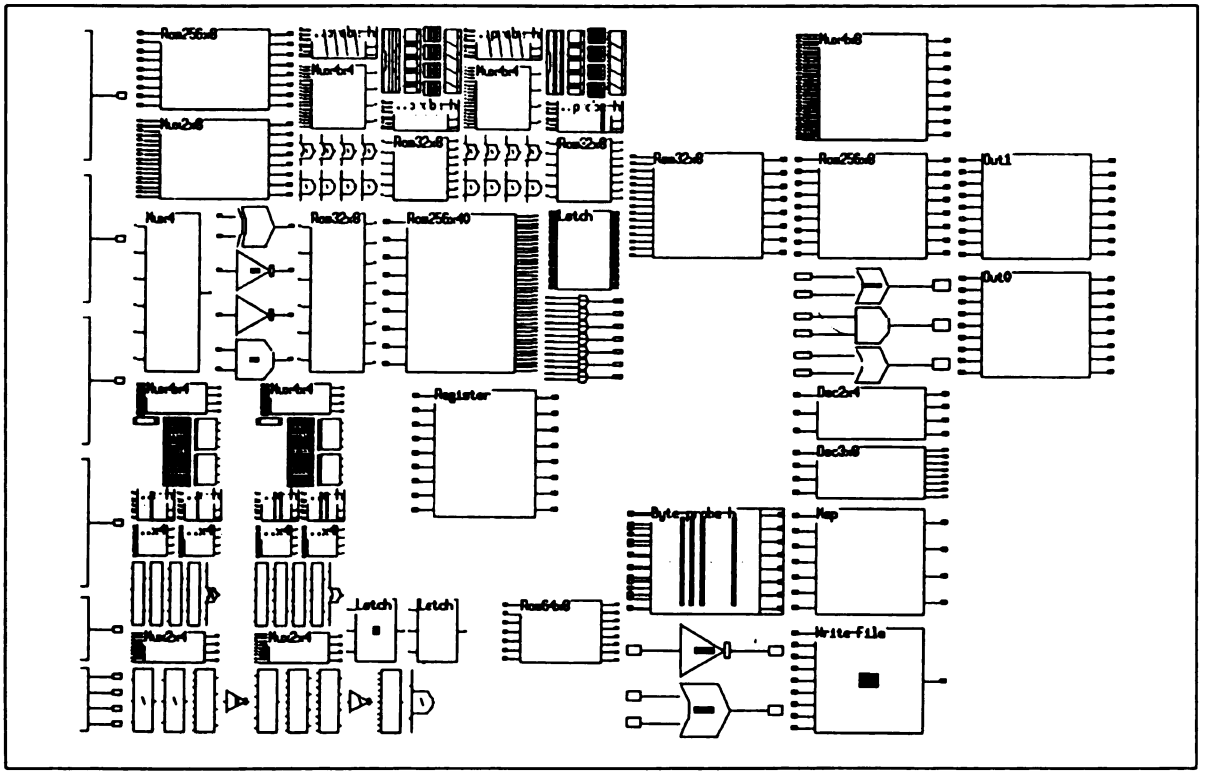

Figure 2.2. A model of an 8-bit microprocessor developed with Sim

2.2.4 Other Results

Others have recognized the importance of automatic equation formulation in continuous simulations. Cellier and Elmqvist [20] have observed that simulation methodologies used today contain artifacts of the technology in place when they were developed. For example, a continuous simulation language may accept equations in a form that

is convenient for the numerical integration algorithms used. They have developed a language called Dymola which allows the user to enter the equations in familiar forms. Dymola is a pre-processor in the sense that it manipulates the symbolic equations into a form acceptable for an existing simulation language. Dymola and other approaches to symbolic manipulation of representative equations could be applied in a more extensive implementation of the underlying mathematical servers utilized by the simulation framework presented here. This dissertation focuses on the management of components and the relationships among them during simulation.

Cremer has noted that in the mechanical engineering field, computer aided design systems have typically focused on the formulation and solution of an unchanging set of equations [28, 52]. He proposes a simulation architecture that provides a natural and uniform treatment of events such as impacts, contact formation or breakage, and control-algorithm state changes. Cremer's simulator, *Newton*, consists of three main components: the definitions and representation module, the analysis module, and the report system. The definition and representation module is responsible for parsing a high-level language object representation of the mechanical system being modeled. The analysis module deals with automatic motion equation formulation, constraints, and an event handling mechanism for handling the impacts, contacts, and other changing relationships among the components being modeled. While Cremer's work focuses specifically on mechanical systems, this dissertation develops a more general object-oriented concurrent framework which has wider application.

Gilmore's results also deal with dynamic connections in the area of mechanical systems [42]. However, the emphasis of Gilmore's dissertation is the automatic detection of topological modifications within a mechanical system.

Other publications in distributed and parallel simulation, object-oriented simulation, and visual-interactive simulation which were useful in preparing this dissertation are included in the bibliography.

2.3 Summary

This chapter has provided the basic background material from the literature that relates to this dissertation. The approach to modeling dynamic connections among constitutive components is based on recent results in object-oriented research, in particular object-oriented frameworks. Results from object-oriented simulation, visual-interactive simulation and physical systems simulation have been examined and contrasted with the methodology proposed in this dissertation.

CHAPTER 3

Representations and Definitions of a Simulation Framework

Object-oriented frameworks can be thought of as skeleton applications that serve as a base from which more specific applications can be built. The general properties (algorithms and data structures) encapsulated by the framework are reused every time a new application is developed. The framework presented here is summarized in Figure 3.1. The framework is divided into four layers: the applications layer, the interface layer, the simulation layer, and the computational layer. This chapter gives an overview of the framework and develops the terminology and concepts pertaining to each of these layers. The applications layer is examined with respect to the facilities the framework offers to applications being derived from it. The interface layer consists of the key abstractions such as component, terminal, submodel, and others, that are used to develop simulation applications. The simulation layer consists of the underlying simulation mechanisms such as event handling, connectivity management, equation formulation and equation accumulation. The discussion on the computation layer will provide insight on the underlying numeric and symbolic computation requirements of the framework. The chapter concludes by assembling an example model that combines the concepts developed in this chapter.

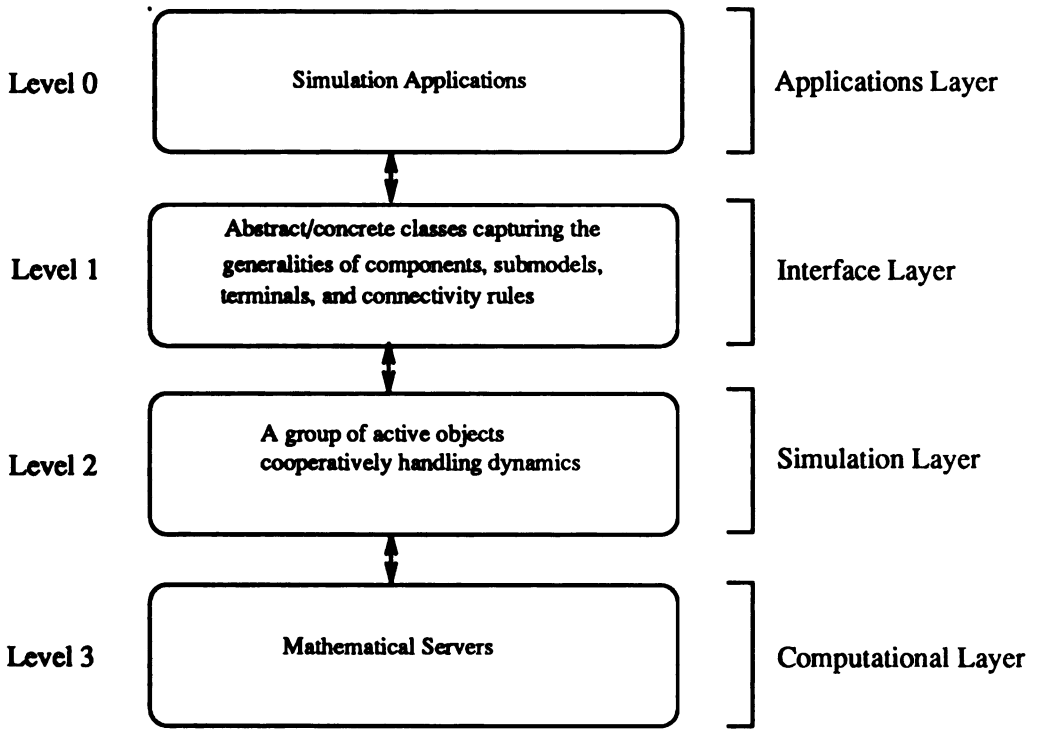

Figure 3.1. The framework with respect to simulation applications and underlying mathematical servers.

3.1 Layer 0 - The Applications Layer

The applications layer is the layer in which the framework is actually instantiated. By inheriting the abstract classes provided by the interface layer (see Section 3.2), a new simulation application can be developed. The application developer need not be concerned with the details of the underlying mechanisms which represent dynamic connections or components, since the underlying framework handles them. The application need only notify the underlying framework when components and connections appear or disappear. To work in this layer, a developer need only be familiar with the abstract classes provided by the interface layer[1] Thus, the framework is a gen-

[1]An obvious exception to this statement is that the developer must be aware of the limitation of the particular implementation of the framework being used. For example, the implementation referred to in this dissertation is limited to systems of linear algebraic and linear differential equations.

eral conceptual model that can be applied to a whole family of related applications. Chapter 6 contains several example applications of the framework.

3.2 Layer 1 - The Interface Layer

The interface layer serves as an interface between the applications built on the framework, and the underlying simulation mechanism. It contains the abstract classes from which the more specialized classes used in applications are derived.

3.2.1 Components

Definition 3.1 *A* **component** *is a primitive constituent part of a system. It is primitive in the sense that it defines only terminals*[2]*, variables, equations and methods to describe its behavior, and contains no other components.*

In the interface layer, the framework defines the abstract class, *Component*[3], from which all primitive components are derived. A detailed discussion of this class definition can be found in Appendix A, Section A.1. The procedures referred to in the definition are blocks of code which are executed when a component receives or sends messages from one of its terminals.

3.2.2 SubModels

Definition 3.2 *A* **submodel** *is a component that contains other components.*

The interface layer represents submodels with the abstract class *SubModel*. The *SubModel* class inherits the abstract class *Component*. Hence, every *SubModel* instance

[2]Terminals will be defined in Section 3.2.3.

[3]For the remainder of the dissertation, the names of classes and their members are italicized.

is also a *Component*. The *SubModel* abstraction is important in that it allows hierarchical models to be constructed. Note that by the definition given, *SubModels* may contain *SubModels*. The hierarchical relationships among the *Components* and *SubModels* in any given model form a tree where the leaf nodes are primitive *Components* and the interior nodes are *SubModels*. The root node of this tree is called the *world SubModel*. Every model must have at least one *SubModel* which serves as the world *SubModel* in the hierarchy.

Instead of defining the *SubModel* abstraction, an alternative was to give the *Component* abstraction a recursive definition; that is, a *Component* instance could be allowed to contain *Component* instances. Introducing the *SubModel* abstraction was stimulated primarily by practical observations. First, there is overhead associated with each *SubModel* instance. As will be seen shortly, each *SubModel* object instantiates three concurrent objects. Collectively these three objects manage the components and connections spawned within the *SubModel* object. In terms of implementation it is useful to be able to specify when these concurrent objects are needed and when they are not. Defining the two abstractions *Component* and *SubModel*, was one way to accomplish this. When it is certain that the behavior of a component can be modeled sufficiently as a primitive component (i.e. it will never contain other components), the *Component* abstraction, which does not instantiate the three concurrent objects, can be used. The decision to define the *SubModel* abstraction was also influenced by our experience with existing simulation tools, all of which explicitly provided an abstraction or mechanism for representing components that are composed of other components.

In addition to its support for hierarchical model construction, the *SubModel* class provides the interface layer with entry points into the simulation layer through its member functions. A more detailed discussion of the *SubModel's* class definition is given in Appendix A, Section A.3.

3.2.3 Connections and Terminals

The notion of a connection may vary depending on what type of system is being modeled. If a circuit is being modeled at the digital logic level, then a connection may be thought of as a printed-circuit board trace that connects the output terminal of one component to the input terminal of another component. If a variable is used to represent the digital signal value for each of the input/output terminals of the digital components, the connection can be represented by an equation which equates the variables associated with the connected terminals. However, if the same circuit is being modeled at a lower level in the design hierarchy where quantities such as voltage and current are to be considered, the notion of being connected can no longer be represented with a simple equality equation. For an electrical component it is sufficient to associate a voltage and current variable with each point of connection. The connection can now be represented with two equations. One equation equates the associated voltage variables, and the other states that the sum of the current variables is zero. A component's potential to connect to other components is established by its *terminals.* In terms of their constituent parts, terminals and connections are defined as follows:

Definition 3.3 *A* **terminal** *is characterized as a set of variables organized into partitions. Each partition is assigned a single connection rule which specifies how each variable in the given partition is to be used in formulating the representative connectivity equation(s).*

Definition 3.4 *A* **connection** *is a set of terminals.*

A component may have zero or more terminals, but a terminal can be in no more than one connection at any instant in time. In addition to the *connectivity equations* which represent the connections, components themselves may define *constitutive equations*

which relate the terminal variables and perhaps other variables internal to the component. For example, a model of a resistor component would have associated with it the equation resulting from Ohm's law: $V = IR$. Thus, there are two general classes of representative equations used in the framework: connectivity equations and constitutive equations.

Definition 3.5 *A* **connectivity equation** *is an algebraic equation which represents in part (or completely) a connection between one or more components.*

Definition 3.6 *A* **constitutive equation** *is an equation which represents a relationship among a component's variables independent of the component's connections to other components.*

In essence, the problem of modeling dynamic connections has been mapped to the problem of formulating, maintaining, and solving a dynamic set of equations.

3.2.4 Connection Rules

It is desirable to anticipate what types of equations can result from connections and to build the respective equation formulation rules into the modeling methodology. Ultimately, one would like to discover some finite set of rules which could be used to formulate equations describing any conceivable connection. Connection rules are defined as follows:

Definition 3.7 *A* **connection rule** *is a procedure that receives a set of component variables as input, and produces a set of connectivity equations as output.*

An exhaustive set of formulation rules has not yet been identified, but there are two general rules which are useful in a variety of domains. As described in the previous section, the set of variables associated with a terminal are organized into partitions, and an equation formulation rule is designated for each partition. Consider a terminal

that consists of two partitions. Denote the set of variables associated with the two partitions by:

$$P_1 = \{X_i \mid 1 \leq i \leq s\}$$

$$P_2 = \{Y_i \mid 1 \leq i \leq t\}$$

where s and t denote the number of variables in the two partitions. Multiple variables are assigned to partitions to allow vector quantities such as position, velocity, acceleration, force, etc., to be associated with a terminal. Assuming that n compatible[4] terminals have been connected, we will use the notation X_i^j to denote the variable X_i which is associated with the jth terminal. The general formulation rule associated with the first partition yields the set of *identity equations*:

$$X_i^1 = X_i^2 = \cdots = X_i^n,$$

where $1 \leq i \leq s$. Using an analogous notation for the second partition, the second general formulation rule yields the set of *conservation equations*:

$$\sum_{j=1}^{n} Y_i^j = 0,$$

where $1 \leq i \leq t$.

3.2.5 Events

The state of a component at any point in time consists of the values and/or configurations of its attributes (variables, terminals, equations, etc.) at that time. The state of a model at any point in time consists of the state of all components populating

[4]A set of terminals is said to be compatible if they define the same number of partitions, the same number and types of variables in each partition, and have the same connection rules associated with each partition.

the model at that time, and the connections among those components. An event is defined as follows:

Definition 3.8 *An* **event** *is a possible change in the state of a model.*

The framework defines an abstract class, *Event.* At a minimum, an event must consist of two members: a time stamp and an evaluation method. The time stamp indicates when the event is scheduled to occur. The evaluation method, which is executed at the scheduled time, initiates the change in model state. The framework defines a set of events for creating components, removing components, making connections, removing connections, querying for variable values, etc. A more detailed discussion of the abstract class *Event* and the classes derived from it is given in Appendix A, Section A.4

3.2.6 Examples

The following two examples, one of a simple electrical system and the other of a simple mechanical system, will clarify the concepts presented so far.

Electrical Example

Three electrical components are illustrated in Figure 3.2. In terms of the framework, each of these components would be derived from the abstract class *Component* and each would have two terminals. In addition, they would be contained by a single world submodel that is derived from the abstract class *SubModel.* The electromotive force component will be referred to as C_1, the resistor as C_2, and the capacitor as C_3. When referencing the variable of a component, it will be prefixed with the component identifier. For example, $C_2.V_a$ would refer to variable V_a of the resistor. Assume that initially the components are unconnected. The constitutive equations

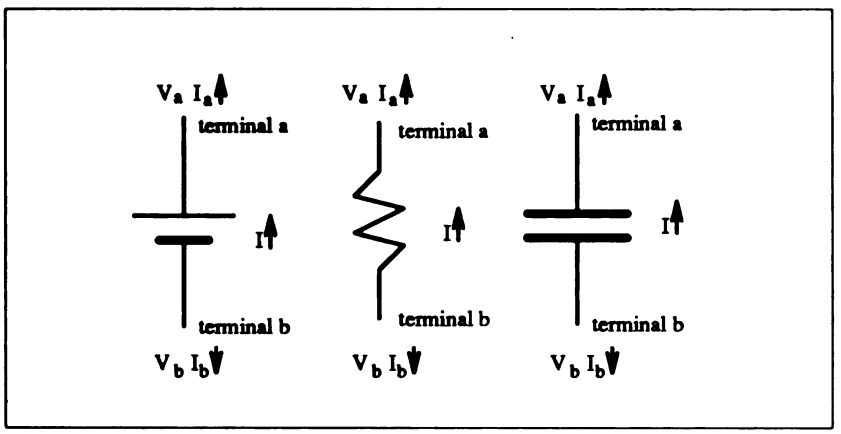

Figure 3.2. Three electrical components and their terminal variables.

for the electromotive force source component:

$$C_1.V_a - C_1.V_b = v$$

$$C_1.I - C_1.I_a = 0.0$$

$$C_1.I_a + C_1.I_b = 0.0$$

indicate that the voltage drop across the component's terminal is v volts and the current flowing through the unconnected device is zero. The constitutive equations for the resistor component would be:

$$C_2.I - C_2.I_a = 0.0$$

$$C_2.I_a + C_2.I_b = 0.0$$

$$C_2.V_b - C_2.V_a - \mu C_2.I = 0.0$$

where μ is a constant which indicates the resistance. These equations represent the fact that the current through the unconnected component is zero and that the voltage and current are related as stated in Ohm's Law. The constitutive equations for the capacitor can be expressed as:

$$\frac{dC_3.V}{dt} - \frac{C_3.I}{c} = 0.0$$

$$C_3.V = C_3.V_a - C_3.V_b$$

$$C_3.I - C_3.I_a = 0.0$$

$$C_3.I_a + C_3.I_b = 0.0$$

where c is a constant denoting capacitance. There are two variable partitions associated with each electrical terminal. The first partition contains the voltage variable, while the second contains the current variable. The two general rules given earlier in Section 3.2.4 are applied to the two partitions whenever a connection occurs. Assume at some later time terminal a of C_1 is connected to terminal a of C_2, terminal b of C_2 is connected to terminal a of C_3, and terminal b of C_3 is connected to terminal b of C_1 in that order (see Figure 3.3). These topological modifications would be scheduled by creating and scheduling the appropriate *Events*. When the first connection occurs, the equations

$$C_1.I_a + C_2.I_a = 0.0$$

$$C_1.V_a - C_2.V_a = 0.0$$

are formulated based on the two rules. In a similar fashion the second and third connections result in the equations:

$$C_2.I_b + C_3.I_a = 0.0$$

$$C_2.V_b - C_3.V_a = 0.0$$

and

$$C_1.I_b + C_3.I_b = 0.0$$

$$C_1.V_b - C_3.V_b = 0.0$$

respectively. At this point in time, there are 16 simultaneous equations, one of which is a first-order differential equation, and 16 unknowns. This system can be solved for all nine of the current variables as well as the voltage drop, $V_a - V_b$, for each of the components.

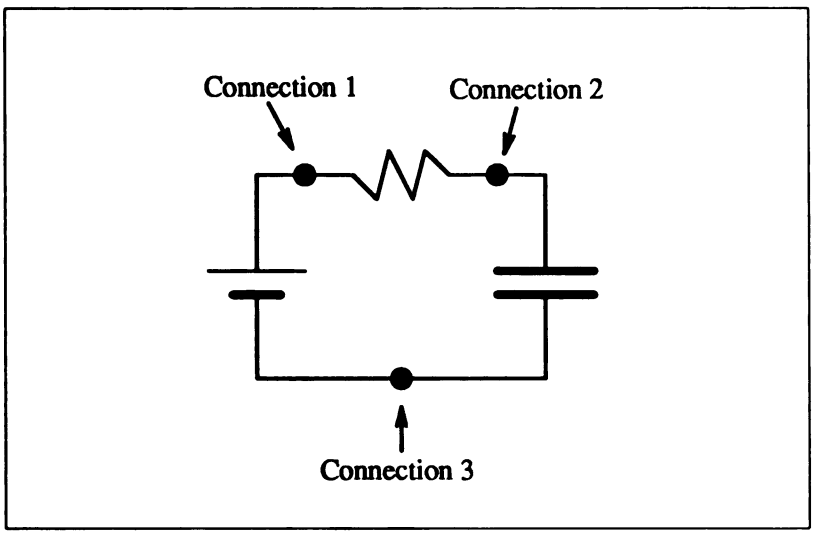

Figure 3.3. Components become connected

Mechanical Example

In the mechanical domain the notion of a connection can be related to the position of the components; that is, if two or more terminals are connected, then the positions

of those terminals are considered to be equal. Assume there are three mechanical components: a spring, a block, and a dashpot, each having one terminal with the associated force and position variables, as shown in Figure 3.4. We will use the same

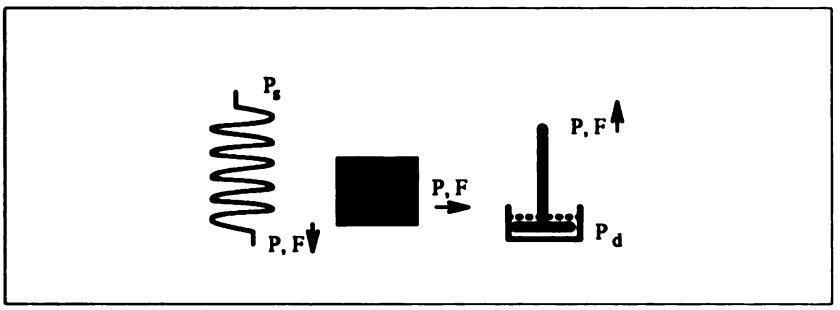

Figure 3.4. A spring, block, and dashpot

variable notation as used in the previous example, with C_1, C_2, and C_3 representing the spring, block, and dashpot, respectively. For the sake of simplicity, assume the position is in one dimension. The spring and the dashpot have associated with them an anchor position designated by P_s and P_d, respectively. Constitutive equations for these three components are given as:

$$C_1.F = -\kappa(C_1.P - C_1.P_s)$$

$$C_2.F = m\frac{d^2C_2.P}{dt^2}$$

$$C_3.F = \gamma\frac{d(C_3.P - C_3.P_d)}{dt}$$

where κ is the spring constant, m is the mass of the block, and γ is the constant associated with the dashpot. There are additional details such as the initial values for the position variables and their derivatives, which are not shown here.

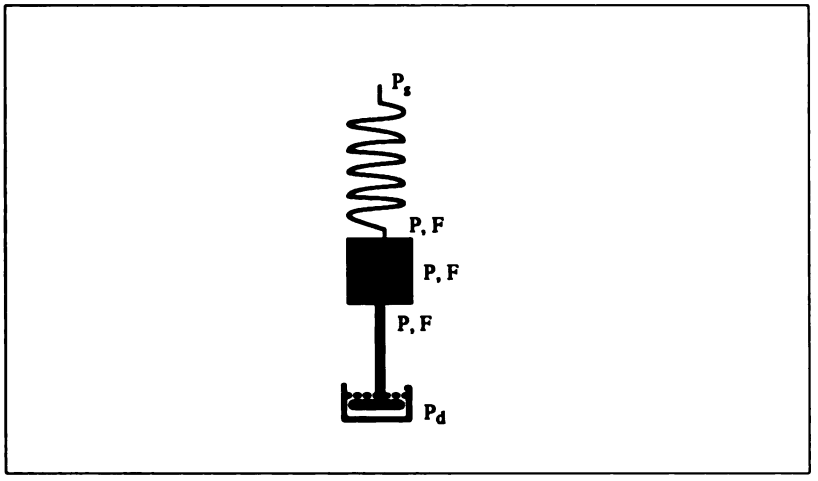

Figure 3.5. The connected components

Assume that at a later time, the components connect as illustrated in Figure 3.5. The equations generated by the two general rules are given as:

$$C_1.P = C_2.P = C_3.P$$

$$C_1.F + C_2.F + C_3.F = 0$$

which indicates that the positions of the three component terminals are equal and that the forces associated with each component sum to zero.

3.3 Layer 3 - The Simulation Layer

Every *SubModel* instance contains three active objects: the *ConnectionManager*, the *EquationFormulator*, and the *EquationAccumulator*; and references to two other active objects: the *EventHandler* and *MathSolver*. These objects are active in the sense that each one of them is actively executing their own thread of control. It is important to understand that every *SubModel* instantiates these three objects. Hence, a

model consisting of n *SubModels* consists of $3n + 2$ active objects (the two extra are for the *EventHandler* and the *MathSolver*). The algorithm executed by each of these objects is a simple consumer algorithm which is illustrated in Figure 3.6. Step 2 in the

```
1) repeat the following two steps indefinitely
2)     Get the next message from the work queue
3)     Process the message
```

Figure 3.6. Algorithm executed by active objects.

algorithm is assumed to be a blocking operation. That is, if no messages are in the queue, the object will wait until there are. Each of these objects, with the exception of the EventHandler, maintains a first-in-first-out queue into which messages can be deposited. The EventHandler maintains a priority queue instead of a first-in-first-out queue. The messages which are enqueued cause the active object to invoke its own local member functions (step 3 in Figure 3.6) which may result in sending messages to the other active objects. The relationships among the *EventHandler, ConnectionManager, EquationFormulator* and *EquationAccumulator* objects are shown in Figure 3.7. The blocks represent the active objects and an arrow from one block to another indicates that the block may send messages to the other block. If a particular implementation of the framework maps the active objects onto multiple physical processors, the objects can be synchronized using either an optimistic or conservative approach [49, 23]. The following sections describe each of these objects briefly.

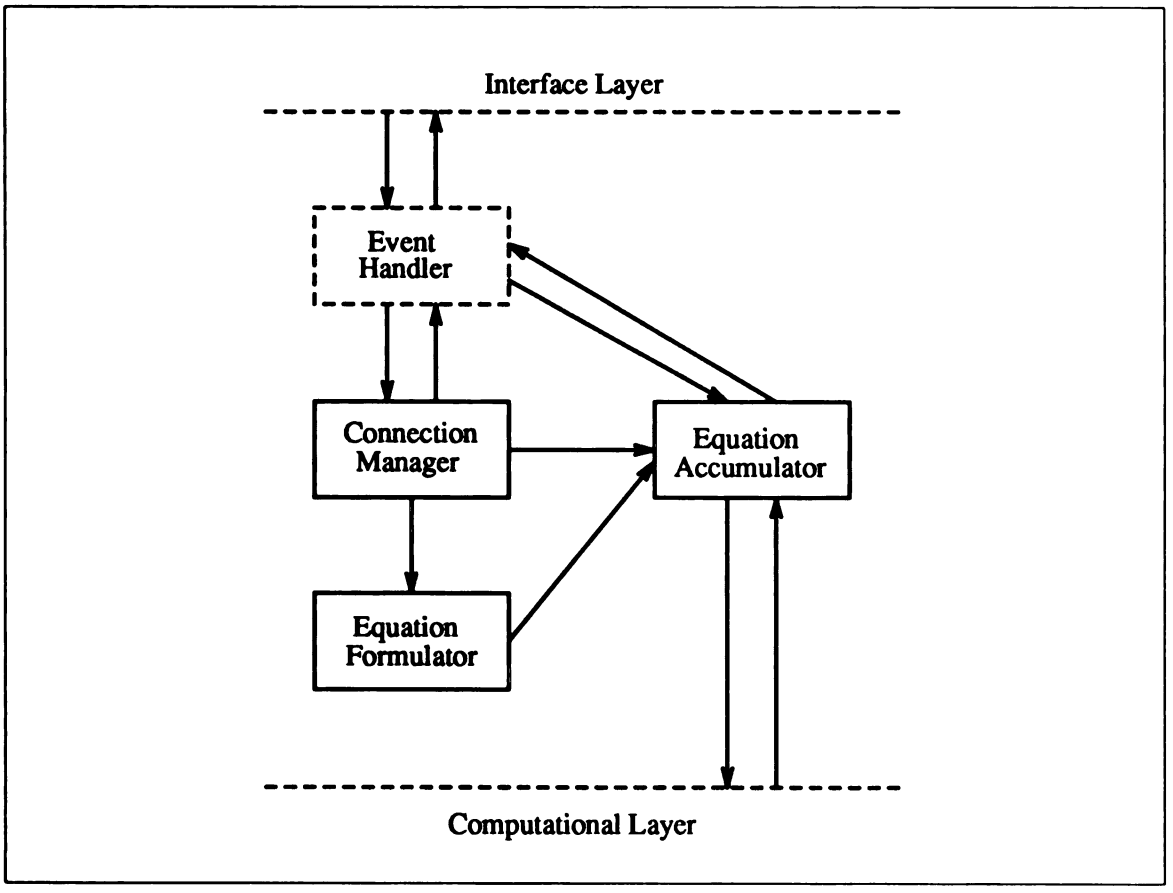

Figure 3.7. The relationships among the active objects in the simulation layer.

3.3.1 The EventHandler

The *EventHandler* executes an algorithm very much like that illustrated in Figure 3.6. However, instead of a first-in-first-out queue, the *EventHandler*'s incoming messages (events) are stored in a priority queue. During each iteration the *EventHandler* selects the event with the oldest time stamp and evaluates it. If the priority queue is empty, the *EventHandler* blocks until a new event is scheduled. Events themselves are objects. Every event defines a member function called *eval.* Using polymorphism, multiple types of events can be derived from the abstract class *Event* and executed with a single generic algorithm.

The granularity of a single event in this framework may be very coarse. For example, a single variable query event may result in the organization and solution of a large system of simultaneous algebraic and differential equations. Hence, parallelism can be exploited at the event level.

3.3.2 The ConnectionManager

The *ConnectionManager* object maintains lists of the registered components and connections. The *ConnectionManager* is not responsible for deciding when two or more components are considered connected, but rather, it is informed when components become connected or disconnected, so that it can update its lists and initiate connectivity equation formulation. The *ConnectionManager* is responsible for providing error detection. Error situations may arise in a number of ways. For example, an application may attempt to connect a set of non-compatible terminals or attempt to remove a connection that does not exist.

3.3.3 The EquationFormulator

The *EquationFormulator* takes a set of terminals (a connection), and applies the connection rules which are associated with those terminals. The set of connectivity equations which are generated are registered with the *EquationAccumulator*.

3.3.4 The EquationAccumulator

The *EquationAccumulator* maintains the set of constitutive and connectivity equations. It is responsible for organizing systems of equations whenever the values of variables are demanded by the application. An *EquationAccumulator* instance has access to only the connectivity and constitutive equations defined in the context of the *SubModel* that has instantiated it. Since an *EquationAccumulator* contains only the equations representing the local components and connections, all the *EquationAccumulators* must cooperatively organize complete systems of equations.

3.4 Layer 4 - The Computational Layer

The computational layer is where the systems of equations are solved. Since it is impossible to anticipate which types of equations may be encountered, special attention was given to developing a meaningful protocol between this layer and the simulation layer. The advantage of this is that new computational servers can be developed and plugged into the simulation layer without difficulty. The computational servers will communicate asynchronously with the simulation layers. In addition to solving equations, the protocol also allows for equation parsing to be carried out by the computation layer. This is important since it allows new capabilities, in terms of the types of equations that can be solved, to be added to the framework, without modification of the simulation layer. The simulation layer accesses the computational layer via the *MathSolver* object.

3.5 Putting It All Together

The simple demonstrative model shown in Figure 3.8 is used to clarify the concepts developed in this chapter. The outermost *SubModel*, labeled C_1, contains a primitive component labeled C_2 and a nested *SubModel* labeled C_3. The *SubModel* C_1 contains *ConnectionManager*, *EquationFormulator* and *EquationAccumulator* instances, as well as references to the *EventHandler* and *MathSolver* objects. The primitive component C_2 defines n variables, labeled $V_1, V_2, ...V_n$, and m constitutive equations which are denoted by $E_1, E_2, ...E_m$. The only internal details shown for the *SubModel* C_3 are its three active objects, and its references to the *EventHandler* and *MathSolver* objects. Details regarding the components and connections contained within C_3 are not shown.

Within the context of *SubModel* C_1, the components C_2 and C_3 each define a single terminal. These two terminals form a connection which is labeled C_a in the figure. The *ConnectionManager* instantiated by *SubModel* C_1 maintains two lists: a list of the components registered within its context (C_2 and C_3) and a list of the connections registered within its context (C_a). The *EquationFormulator* applies the appropriate connection rules when connections are registered. The *EquationAccumulator* is shown with its accumulation of registered variables, constitutive equations, and connectivity equations (denoted by Equations(C_a) in the figure).

Both of the *SubModels* in the figure are shown with a dashed line separating the interface layer from the simulation layer. The items found above the dashed lines are defined by the application in the interface layer. Those items below the dashed lines are defined by the framework in the simulation layer. Therefore, when a *SubModel* object is instantiated, the framework automatically instantiates *ConnectionManager*, *EquationFormulator*, and *EquationAccumulator* objects within the *SubModel*'s context, but it is the application's responsibility to populate the *SubModel* with compo-

nents and connections.

The *EventHandler* and *MathSolver* objects are automatically instantiated without intervention from the application layer. Each *SubModel* instance maintains references to these objects.

3.6 Summary

This chapter has developed the definitions and terminology of an object-oriented framework for modeling dynamic connections. The framework has been divided into four layers: the applications layer, the interface layer, the simulation layer, and the computational layer. The terms: component, submodel, terminal, connection, connection rule, constitutive equation, connectivity equation and event were defined and discussed. Examples were provided to help clarify the definitions.

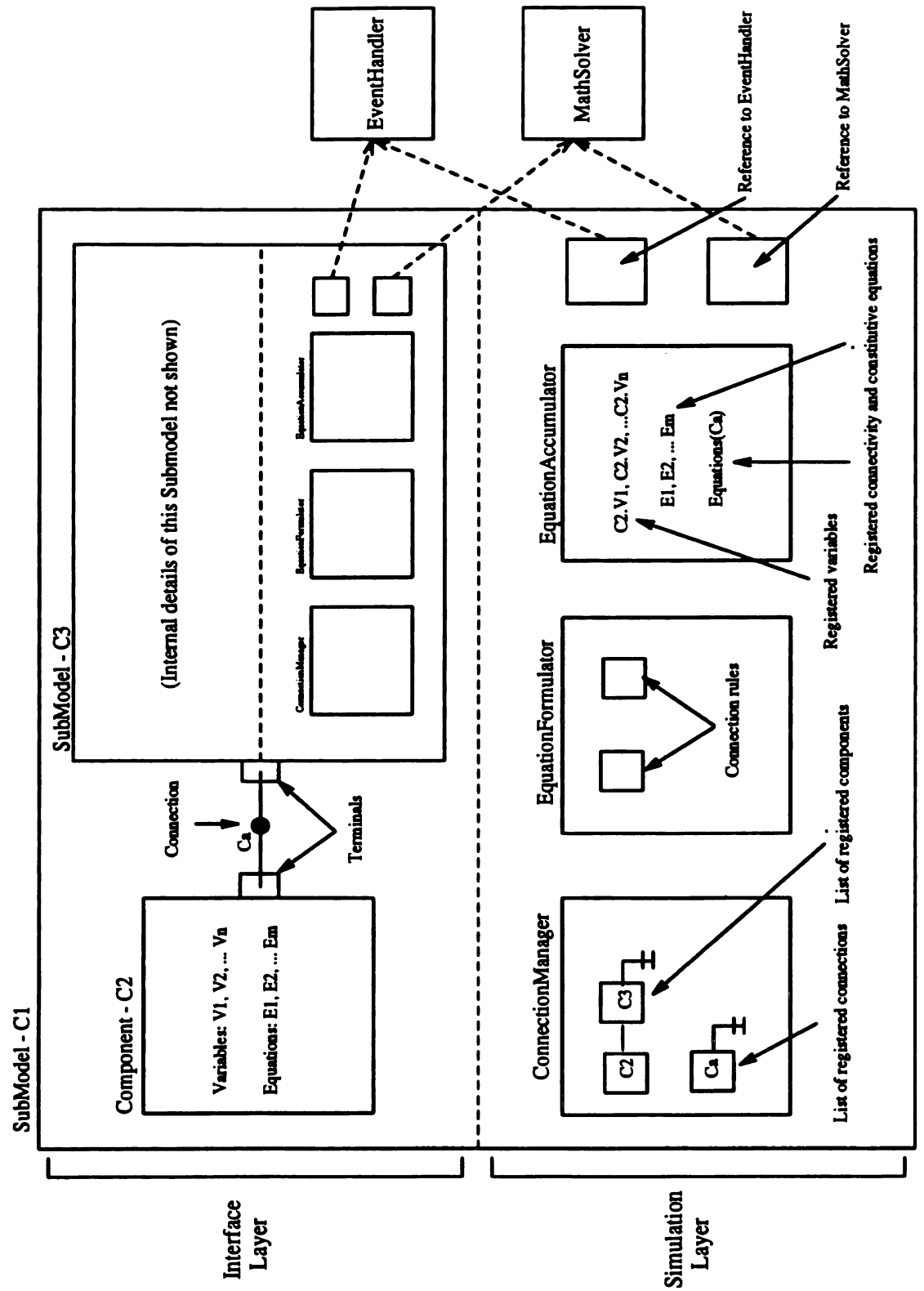

Figure 3.8. Putting it all together.

CHAPTER 4

Concurrency, SubModels and Discrete Models

Concurrent object-oriented programming is an emerging software technology that has much to offer in the way of providing a solid software foundation for multiprocessor computers. The first section of this chapter develops the rationale for basing the simulation framework on a set of concurrent objects, the implications of this approach and the *SubModel*'s encapsulation of the concurrent objects.

The second section of this chapter describes how support is provided for developing discrete components which define methods instead of constitutive equations to describe their behavior. All the examples presented so far were continuous models in which components were described using algebraic and differential constitutive equations. In some situations it is more intuitive to allow the internal behavior of components to be described with methods rather than constitutive equations. This mechanism used in concert with constitutive equations results in a powerful, yet flexible, simulation framework.

4.1 Object-Oriented Concurrency

There has been a great deal of interest in concurrent object-oriented programming [4, 27, 30, 55, 57, 91, 103]. A concurrent object-oriented program is a program based on the object-oriented paradigm that contains more than one thread of execution. Although conceptually the program expresses concurrency, in actuality the program can be executed sequentially or in parallel. In addition to the merits of the object-oriented paradigm, concurrent object-oriented programming methodologies are desirable in that they allow the programmer to focus on the objects and their interactions without having to be concerned with the anomalies of the underlying computing platform. In other words, the programmer is given the ability to express potential parallel execution in the program without having to get involved in the details on how this is actually implemented on the computing platform being used. The concurrent object-oriented approach was adopted here for this reason.

There are three common patterns for parallel computation [4]. Pipeline concurrency can be achieved when a problem is divided into a sequence of stages. After a computation finishes the first stage it moves sequentially to the second stage. At the same time, a new computation can enter the first stage. When both of these computations are completed the first computation moves on to the third stage, the second computation moves on to the second stage, and a new computation can enter the first stage. This process is repeated until eventually every stage of the pipeline will be processing. Thus, in an n stage pipeline, n different computations are carried out in parallel. An occasional lack of input may will cause a "bubble" of idleness to pass through the pipeline. A second pattern for parallelism is the divide and conquer pattern. Here the problem is divided into sub-problems, each of which is processed in parallel. The results are joined to form the complete solution. A third pattern involves cooperative problem-solving. In this pattern a complex network of processes

work together in parallel on a given computation. The processes share intermediate results with each other through message passing.

Both the pipeline and the cooperative problem-solving patterns can be identified in the simulation framework. The pipeline pattern arises when a connection is registered within a given *SubModel*. The *ConnectionManager* receives the connection registration message first. After error checking, it updates the appropriate symbol tables and sends a message to the *EquationFormulator* for the formulation of connectivity equations. Once the new equations are formulated, the *EquationFormulator* sends them in a message to the *EquationAccumulator*. The *EquationAccumulator* adds the new equations to its list of active equations. This sequence of computations forms a three stage pipeline. The cooperative problem-solving pattern of parallelism arises when a set of *EquationAccumulator*s cooperatively organize a system of equations for solution.

From the framework application's perspective, the concurrent object-oriented approach may complicate matters. That is, if the underlying framework consists of a large number of concurrent objects, how does the application know to which object to send its requests? This problem has been dealt with by having the *SubModel* abstraction provide an interface to the *ConnectionManager*, *EquationFormulator* and *EquationAccumulator* objects that it encapsulates.

4.1.1 The SubModel Interface

The *SubModel* abstraction plays two important roles in the framework:

- it supports hierarchical modeling.
- it hides the details of the simulation layer from the applications layer.

Each *SubModel* instance contains several active objects which cooperatively formulate and maintain the mathematical representation of components local to the *SubModel*

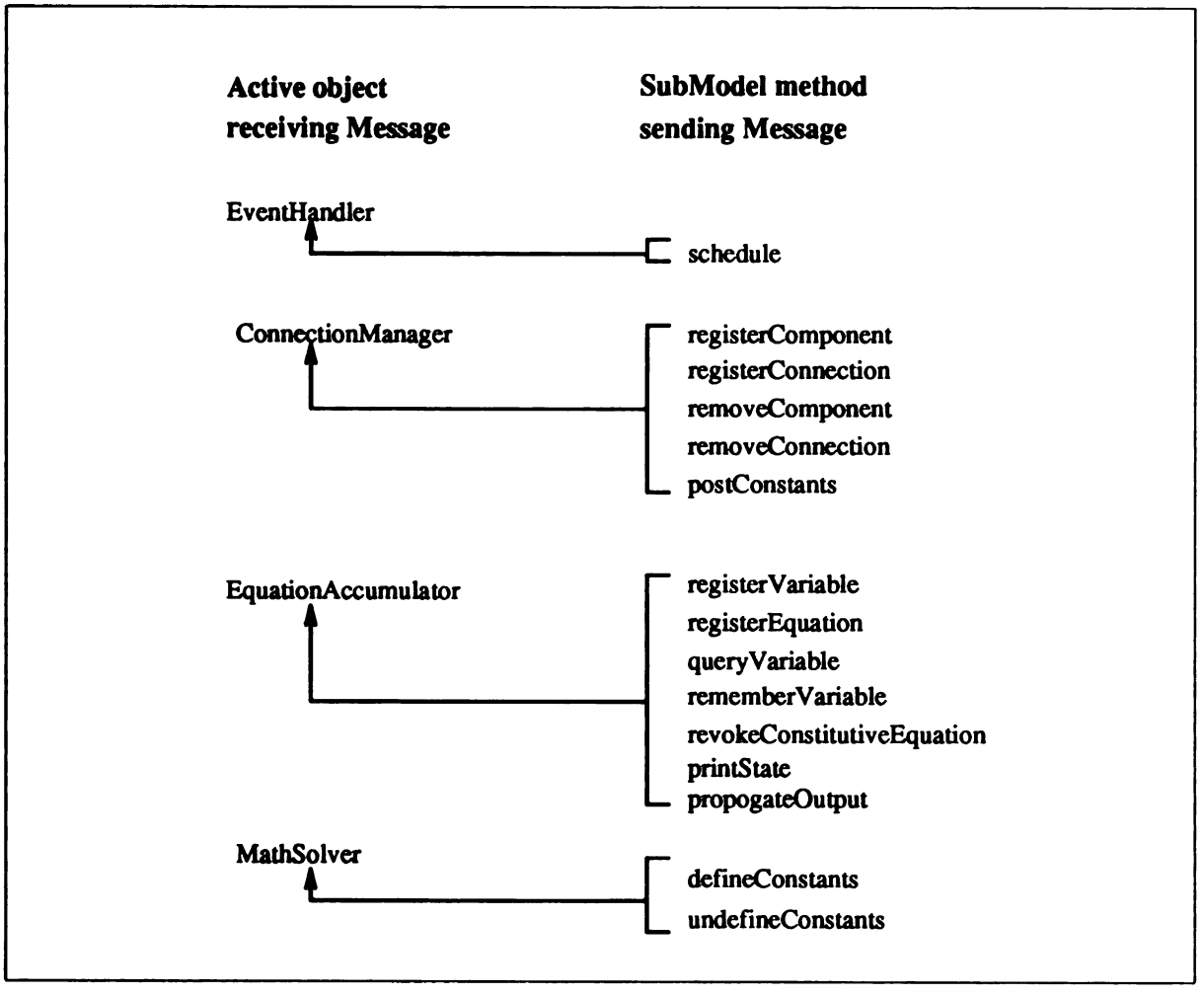

Figure 4.1. SubModel methods to local objects mapping.

and the connections among them. The visibility of the concurrent objects in the applications layer would complicate matters. If an application needs to register a new connection a message must somehow be sent to the appropriate *ConnectionManager* object. If an application wishes to query for the value of a variable at a certain instant in time, a message must be sent to the appropriate *EquationAccumulator* object. Sending the message concerns not only the type of the receiving object (*ConnectionManager* vs. *EquationAccumulator*), but also which instance of that type. There are two alternative approaches to the problem of sending messages from the application to the concurrent objects. One approach is to allow the application to pass messages directly to the active objects. In this approach the application is responsible for deciding which object to send a particular request to. A second approach is to have the *SubModel* class provide a set of methods which receive messages and forward them to the appropriate active object. Instead of sending messages directly to the active objects, the application invokes the appropriate *SubModel* method. The first approach is not desirable, since requiring the applications layer to determine which active object is responsible for a particular operation is cumbersome and will limit the framework's usability. The second approach has been adopted here. This approach is more desirable because it completely hides the active objects from the application layer. Thus, the application only has to deal with the interface of a single class instead of a group of concurrent objects. The *SubModel* method will automatically forward messages to the appropriate active object when invoked by an application. If two components are to be connected the application will call the *registerConnection* method of the *SubModel* instance containing the two components. This method places a message in the input queue of the *SubModel*'s local *ConnectionManager* object. If a variable is to be queried, the application will call the *queryVariable* method of the *SubModel* containing the variable. This method places a message in the input queue of the *SubModel*'s local *EquationAccumulator* object. The complete mapping of *SubModel*

methods to the internal objects is illustrated in Figure 4.1. The set of cooperating objects and their interactions which form the simulation layer is invisible to applications using the framework. Simulation tools derived from a parallel implementation of the framework (those which assign the active objects to multiple processors, and/or those that perform the symbolic or numeric computations in parallel) are distributed applications without the applications' developer providing for it explicitly.

4.2 Modeling Discrete Phenomena

The electrical and mechanical examples given in Section 3.2.6 are both continuous models; the same concepts apply to discrete models. The constitutive equations need not be limited to differential and algebraic equations. Difference equations can be used as long as the underlying servers in the computational layer are capable of parsing and solving them along with the equations generated by the associated connectivity rules.

In some cases it is more intuitive to replace the constitutive equations with a method that describes the behavior of the component. For example, consider multi-level digital logic simulation. A signal change on a gate's input terminal may result in zero or more signal changes on an output terminal. This can be represented by associating a behavior describing method with each gate. Whenever the inputs change, the method is called. The method will schedule events to represent the output signal changes.

The framework provides support in the simulation layer that allows connected *Component* instances to receive messages on their input terminals and send messages from their output terminals. Whenever an input event occurs, the *EventHandler* executes the *input* method of the *Component* named by the event. Whenever an output event occurs, the *EventHandler* executes the *output* method of the *Component*

named by the event. The connectivity rule described in Section 3.2.4 which generates the identity equations must be associated with each terminal in order for messages to propagate correctly. Message propagation can be initiated by invoking the *propagateOutput* method of the *SubModel* class. This method is normally called by a *Component*'s *output* method . All message propagation is based on the relationships established by the underlying connectivity equations. Since these equations appear when new connections are made, and disappear when connections are removed, correct message propagation is guaranteed. Since the definitions of connection and terminal given earlier also apply to discrete models, it is possible for one *Component* to model its behavior with both constitutive equations, and behavioral functions. Such a component can be used as a coupling component between continuous and discrete models. The model of the digital-to-analog converter in Chapter 6 is an example of this type of component.

4.2.1 An Example Discrete Model

Consider the simple manufacturing model in Figure 4.2. In this system a certain product is arriving at a testing station with 20 units per pallet. The testing station tests each product individually, and sends them to one of two warehouses, depending on whether or not they passed the test. In addition, if more than one unit per pallet is faulty, a report is sent to purchasing to order replacements. A copy of the report is also sent to quality control. Each of the entities in the model can be represented as a *Component*. The test station has four terminals, while the other components have one terminal each. Assuming that the letters $a - i$ represent the variables associated with the terminals, and connections are represented by the dots, the following connectivity equations are generated when the connections are registered:

$$a = b$$

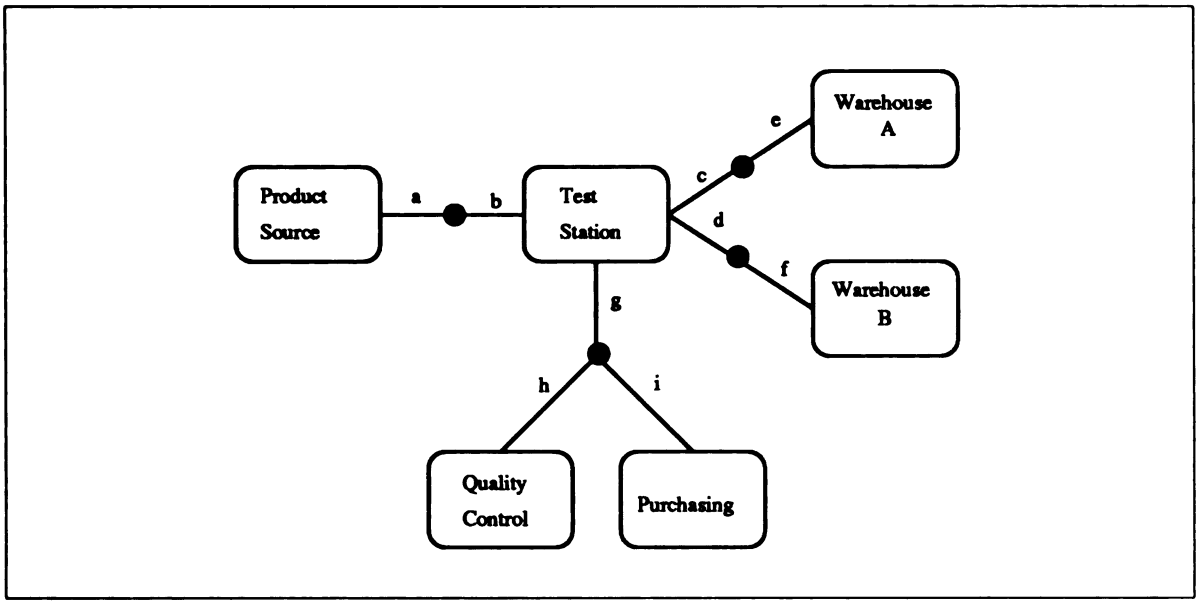

Figure 4.2. An example discrete model.

$$c = e$$

$$d = f$$

$$g = h$$

$$h = i.$$

There are no constitutive equations defined by these components. Instead, each of the components will provide code in their *input* and *output* methods that describe their behavior. Note that in discrete models the connectivity equations are utilized differently than the connectivity equations of the continuous models. In a continuous model the connectivity equation establishes a mathematical relationship among the internal variables of the connected components. In a discrete model the connectivity equations establish the propagation of messages sent from a particular terminal.

Assume the product source block outputs a pallet containing 20 units, two of

which are faulty. The *output* method of the product source block will invoke the *propagateOutput* method of the world *SubModel* containing the source block. From the connectivity equation $a = b$ the method determines that an input event must be scheduled for the test station. When the test station's *input* method is executed it begins testing each unit individually, scheduling output events to send good units to warehouse A and faulty units to warehouse B, from the equations $c = e$ and $d = f$, respectively. When the second faulty unit is discovered, output events are scheduled for warehouse B, quality control, and purchasing. The connectivity equations $g = h$ and $h = i$ cause the *propagateOutput* method to schedule input events for both quality control and purchasing.

The model could be extended by dynamically instantiating and connecting new warehouses, product sources, etc. In addition, components with constitutive equations could be added to the model. In this example, it may be desirable to model the transport from the testing station to the warehouses. The transport mechanism could be a conveyor belt whose dynamics could be described with a differential equation. Models mixing continuous and discrete phenomena will be developed in Chapter 6.

4.3 Summary

This chapter has provided the rationale for using the concurrent object-oriented approach. The implication of using this approach in terms of the framework's usability at the applications layer was pointed out. This problem has been dealt with by having the *SubModel* abstraction provide a simple concise interface to the active objects, completely hiding them from the applications layer. In addition, the framework's mechanism for modeling components that cannot be described with constitutive equations was presented. This mechanism allows behavioral methods to be associated with component descriptions.

CHAPTER 5

The Simulation and Computation Layers

This chapter focuses on the four main tasks carried out by the simulation and computation layers.

- component and connection management.
- connectivity equation formulation.
- equation organization and accumulation.
- equation solution.

Within each *SubModel* instance, there is a group of cooperating active objects carrying out these tasks.

5.1 Component and Connection Management

The simulation framework must provide support for maintaining which components are active in the simulation and the changing relationships among these components. This support is needed when the application is adding or removing components and

connections. If an application indicates that component X should be removed from the model, then a mechanism is needed to decide whether X exists, and how its removal affects the set of equations stored by the *EquationAccumulator*. If an application indicates that a new connection is to be established, a mechanism is needed to determine if the components and terminals involved actually exist, if the connection already exists and if the types of terminals being connected are compatible. This support is provided in each *SubModel* by an instance of a *ConnectionManager*.

Information regarding the existence of a particular component cannot always be determined from the equation set. The existence of a component can be determined from the equation set in situations similar to the following. Assume that an application informs the framework that it would like to remove component C_2, a resistor, which does exist. The simulation layer would search through the set of registered equations and locate the following equations:

$$C_2.I - C_2.I_a = 0$$

$$C_2.I_a + C_2.I_b = 0$$

$$C_2.V_b - C_2.V_a - \mu C_2.I = 0.$$

Since a set of constitutive equations tagged with the C_2 identifier were located, the simulation layer can safely assume that C_2 exists and can proceed with its removal. The framework could correctly determine that the component existed based on the existence of its constitutive equations in the equation set. To see a situation where the existence of a component cannot be determined from the equation set, assume the application program informs the framework that it would like to remove component C_3, a logic gate. The designer of the logic gate associated behavior describing methods (as described in Chapter 4) with the logic gate rather than constitutive equations.

When the framework searches through its equation set, it will find no constitutive equations for C_3 and will incorrectly inform the application that C_3 does not exist.

Making assertions about the existence of connections based on only the equation set, is also a problem. Consider the following equation:

$$C_5.X_a - C_5.X_b = 0$$

Without further information, this could be interpreted in two ways as illustrated in Figure 5.1. It could mean that a two-terminal component C_5 has its two terminals

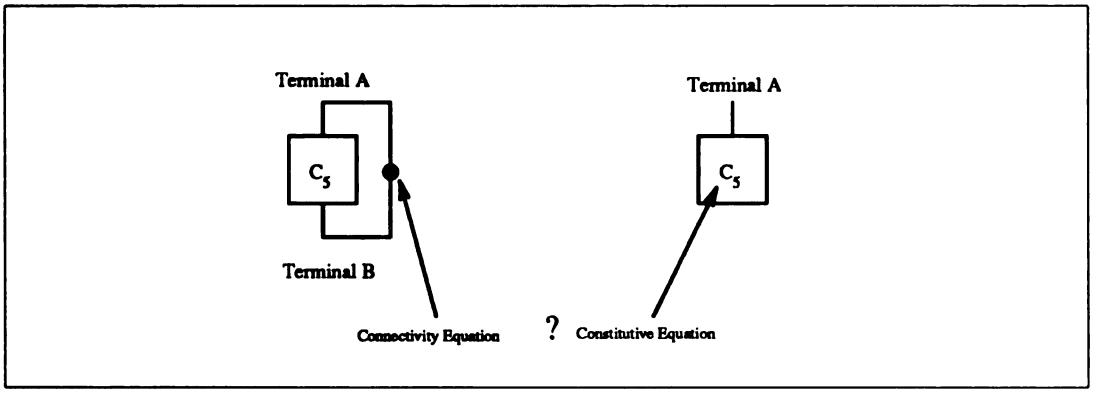

Figure 5.1. Equation ambiguity: connectivity or constitutive?

connected, and the equation is a connectivity equation where X_a and X_b are the variables assigned to terminals A and B (shown on the left in Figure 5.1.) It could also be the case that C_5 is a single component with a single unconnected terminal, and the equation is simply one of its constitutive equations (shown on the right in Figure 5.1.) In addition to this ambiguity problem, different connections may apply different connection rules, making it even more difficult to deduce connectivity

information from the equations alone. The *ConnectionManager* maintains its own information regarding the registered components and connections, making it unnecessary for it to consult the equation set for information regarding the existence of a particular component or connection.

The *ConnectionManager* executes the algorithm outlined in Figure 3.6 of Chapter 3. It accepts messages requesting one of four tasks: component registration, component removal, connection registration or connection removal. Internally, it maintains two symbol tables: one for components and one for connections. Entries are made in these tables when components and connections are registered, and are modified when components and connections are removed. The *ConnectionManager* associates an identifier with each equation. For a constitutive equation, the identifier identifies the component that defined it. For a connectivity equation, the identifier identifies which connection the equation was formulated from.

5.1.1 Component Registration

Components may be spawned within a model at any time during the model's execution. In order for this instantiation to take effect, the application must call the *registerComponent* method of the *SubModel* which contains the component. This method sends a message to the *ConnectionManager* that is encapsulated by the *SubModel*. When the *ConnectionManager* receives the message, it will first consult its component symbol table, to see if another component with the same identifier already exists. If the component's identifier is already present in the symbol table, an error condition has been encountered.[1] If the component's identification number is not present in the component symbol table, an entry is added. The *ConnectionManager*

[1]The manner in which errors are handled or reported to the application layer is an implementation detail and will not be dealt with in this dissertation. The recommended approach is to use an exception handling mechanism like the one proposed for the C++ language [86].

then invokes the *registration* method of the *Component* being registered. This method is defined by the applications layer, and is responsible for the following tasks:

- the definition of the component's variables.
- the definition of the component's constitutive equations.
- the definition of any initial conditions, and/or constants.
- the instantiation of the component's terminals.

The *ConnectionManager*'s component registration method is summarized in Figure 5.2.

```
input:  component
1) if component is not in the component symbol table
2)     insert component into the component symbol table
3)     invoke component's registration method
4) else if component is already in the component symbol table
5)     error condition has occurred
```

Figure 5.2. The *ConnectionManager*'s method for component registration.

5.1.2 Component Removal

When a component is to be removed from the simulation, the application must call the *SubModel*'s *removeComponent* method. This method will send a message to the *ConnectionManager*. When the *ConnectionManager* receives the message it first checks the component symbol table to see if the component to be removed exists. If the

component exists and it is not connected to any other components, a message is sent to the *EquationAccumulator*. The *EquationAccumulator* will then remove any constitutive equations belonging to the component. Finally, the component's entry in the component symbol table is removed. The *ConnectionManager*'s method for removing components is summarized in Figure 5.3.

```
input: component
1) if component is not in the component symbol table
2)     error condition has occurred
3) else if component is in the component symbol table
4)     if component is connected to other components
5)         error condition has occurred
6)     else if component is not connected to other components
7)         send message to EquationAccumulator to revoke all
               component's constitutive equations.
8)         remove component from the component symbol table
```

Figure 5.3. The *ConnectionManager*'s method for component removal.

5.1.3 Connection Registration

An application can form a new connection by calling the *SubModel*'s *registerConnection* method. When invoking this method the application must specify the component and terminal identifiers of the two components to be connected. The method forms a message containing these four identifiers and sends it to the *ConnectionManager*. Upon receiving the message, the *ConnectionManager* first determines if the component and terminal identifiers are valid and if the two terminals are compatible. Next, it checks to see if either terminal is already connected. If both terminals are involved

in different connections, those connections are merged to form one connection. If one of the terminals is connected and the other is not, then the unconnected terminal is merged into the connected terminal's connection. If neither terminal is connected, then the two terminals together form a new connection and a new entry is placed in the connection symbol table. Before formulating new connectivity equations, the *ConnectionManager* invokes the *beforeConnecting* method of every component connected, directly or transitively, to either of the two components being connected. The application uses this method for computing changes in initial conditions during topology changes. When the derivative(s) of a component's variables appear in its constitutive equations, the application must define the initial condition(s) for that variable. These conditions are valid the first time they are used in solving a system of equations. If a topological change occurs at a later point in time, a new system of equations is formulated and solved. The initial conditions given by the application when the component was created are no longer valid at the time of the topological change. These conditions must be computed by the application at the time of the topological change. The values of variables are computed on a demand basis (see Section 5.4). This requires the application to demand and store the values of the appropriate variables whenever a topology change occurs. The *beforeConnecting* method is used for this purpose. After the pre-connection processing is completed, the *ConnectionManager* encapsulates the set of terminals forming the connection, into a message. This message is sent to the *EquationFormulator* which will apply the appropriate connection rules to generate the connectivity equations. The *ConnectionManager's* method for registering connections is given in Figure 5.4.

5.1.4 Connection Removal

An application can remove a component from a connection by calling the *SubModel*'s *removeConnection* method. This method will send a message to the *ConnectionMan-*

```
input: Component identifiers c1 and c2
       Terminal identifiers t1 and t2
 1) if c1 and c2 are in the component symbol table and the terminals
    t1 and t2 are valid:
 2)     if both terminals t1 and t2 are already connected
 3)         merge both into one connection
 4)         update connection symbol table appropriately
 5)     else if only 1 terminal is connected
 6)         merge unconnected terminal into connection
            of connected terminal
 7)         update connection symbol table appropriately
 8)     else if neither terminal is connected
 9)         create a new connection containing t1 and t2
10)         add new connection to connection symbol table
11)     invoke pre-connection processing methods of all directly
        or transitively  connected components
12)     send connection message to the EquationFormulator
13) else if one of c1, c2, t1, or t2, is invalid
14)     error condition has occurred.
```

Figure 5.4. The *ConnectionManager*'s method for connection registration.

ager containing a component and terminal identifier. When the message is received the validity of the component and terminal is verified. If the component exists and the terminal is connected, the terminal is not removed from the connection until after the *ConnectionManager* invokes the *beforeUnConnecting* method for every component connected directly or transitively to the component being removed from the connection. Next, a message is sent to the *EquationAccumulator* which instructs it to remove all connectivity equations for the given connection. If there are only two terminals involved in the connection, the connection is removed from the connection symbol table. If there are more than two terminals involved in the connection, the entry in the connection symbol table is updated and the modified connection is sent in a message to the *EquationFormulator* for the formulation of a new set of connectivity equations. The *ConnectionManager*'s method for removing connections is summarized in Figure 5.5.

5.2 Connectivity Equation Formulation

Connectivity equations are formulated by the *EquationFormulator* object. The *EquationFormulator* executes the algorithm outlined in Figure 3.6 of Chapter 3. It receives and processes only one type of message. This message consists of a set of connected terminals. For each variable partition defined by the terminals, the *EquationFormulator* applies the associated connection rule. The generated connectivity equations are sent in a message to the *EquationAccumulator* for registration. Line 3 in the algorithm presented in Figure 3.6 is expanded in Figure 5.6.

The framework can be augmented with new connection rules whenever necessary. The abstract class *ConnectRule* specifies the formal interface which all new connection rules must adhere to. The details regarding the implementation of connection rules will vary depending on how a particular framework implementation represents

```
input: component
       terminal to be removed
 1) if component is in the component symbol table and terminal
    is connected
 2)     invoke post-connection processing methods for all
        directly or transitively connected components
 3)     send message to EquationAccumulator revoking this
        connection's connectivity equations
 4)     if terminal is connected to more than 1 other terminal
 5)         remove terminal from the connection
 6)         update connection symbol table appropriately
 7)         send message to EquationFormulator for reformulation
            of connectivity rules
 8)     else if terminal is connected to only 1 other terminal
 9)         remove connection from connection symbol table
10) else if component does not exist or terminal is not connected
11)     error condition has occurred.
```

Figure 5.5. The *ConnectionManager*'s method for connection removal.

```
Input: set of homogeneous terminals
1) For each partition defined by the terminals
2)     generate new connectivity equations by applying the
       associated connection rule
3) send a message containing all generated connectivity equations
   to the EquationAccumulator
```

Figure 5.6. The *EquationFormulator*'s algorithm for processing messages.

equations and variables internally, and the format of the equations to be generated. The methods provided by the framework will only reference connection rules via the interface specified by the *ConnectRule* class. New connectivity rules can be added without modifying the framework.

5.3 Equation Accumulation

The goal in developing the mechanisms for equation accumulation and organization is to isolate these mechanisms as much as possible from the details of the simulation application. Two of the reasons for doing this are: first, to minimize the interactions among the objects that have some understanding of the simulation application and the objects that are responsible for solving the constitutive and connectivity equations, and second, to increase the modularity of the framework. In a parallel implementation interactions among active objects require synchronization which reduces the degree of parallelism. Modularity is important because it allows the internal details of equation accumulation and organization to be modified independent of the internal details of connection management as long as the proper interface is maintained.

New constitutive and connectivity equations are accumulated as new components and connections are added to the model. Representative equations are removed as components and connections are removed from the model. A topological change in the model of a single connected set of components represented by a single system of equations, may result in the system being split into two connected sets of components represented by independent systems of equations. Since explicit support for dynamic connectivity is considered a salient feature of the framework, it must not be assumed that topological changes are sporadic events. Topological changes may occur frequently and the mechanisms used to model them must be efficient. That is, when a new connection is added or an existing connection is removed, the framework

must efficiently reorganize the mathematical representation into systems that can be solved by the underlying computational layer. In the remainder of this section, the equation organization problem is addressed by distributing the computations among a hierarchy of cooperating active objects.

5.3.1 The Notion of Hierarchy

In object-oriented design the word *hierarchy* is used in two different ways. The inheritance relationships among classes form one type of hierarchy. This type of hierarchy is illustrated in Figure 5.7. The *Telephone* class is the most general abstraction and is inherited by the *RegularPhone* and *MobilePhone* classes. The *MobilePhone* is used as a base class for the more specialized classes: *PortableUnit* and *HandsFreeUnit*. This type of hierarchy is referred to as a "kind of" hierarchy.

A second type of hierarchy in object-oriented design arises when objects are aggregated by other objects. This is illustrated in Figure 5.8. A cellular phone system is represented by the object *CellularSystem*. The *CellularSystem* contains an arbitrary number of objects of type *Cell*. Each *Cell* object contains an arbitrary number of *MobilePhone* objects, and a single object of type *BaseStation*. The ellipsis indicate an arbitrary number of *Cells* and *MobilePhones*. This type of hierarchy is referred to as a "part of" hierarchy. An example of this type of hierarchy in the simulation framework is when a *SubModel* contains primitive *Components* and other *SubModels*.

The "part of" and "kind of" terminology follows from the way in which the classes are referred to in the two hierarchies. The *MobilePhone* class is referred to in both hierarchies. The "kind of" hierarchy shows that the class *MobilePhone* is a kind of *Telephone*. The "part of" hierarchy shows that objects of type *MobilePhone* are a part of an object of type *Cell*. It is useful to consider both types of hierarchies with respect to the simulation framework. The notion of a "kind of" hierarchy is useful in capturing generalities into reusable classes at the applications layer. In Chapter 6 a

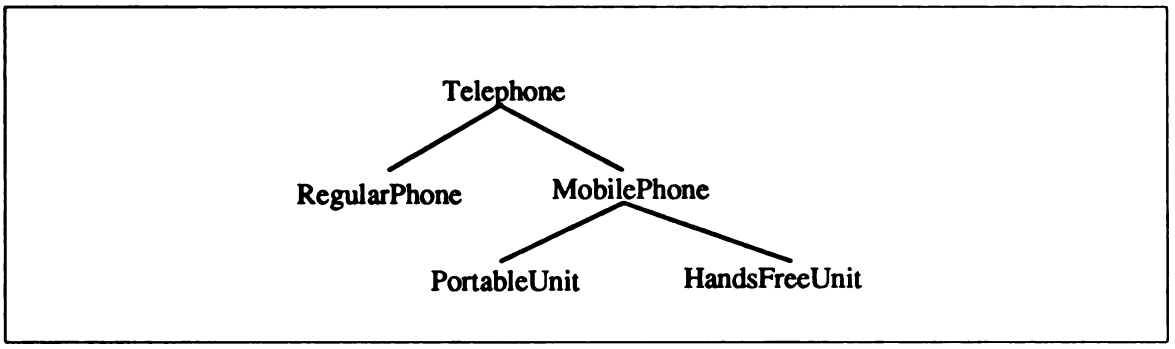

Figure 5.7. Example of a "kind of" hierarchy.

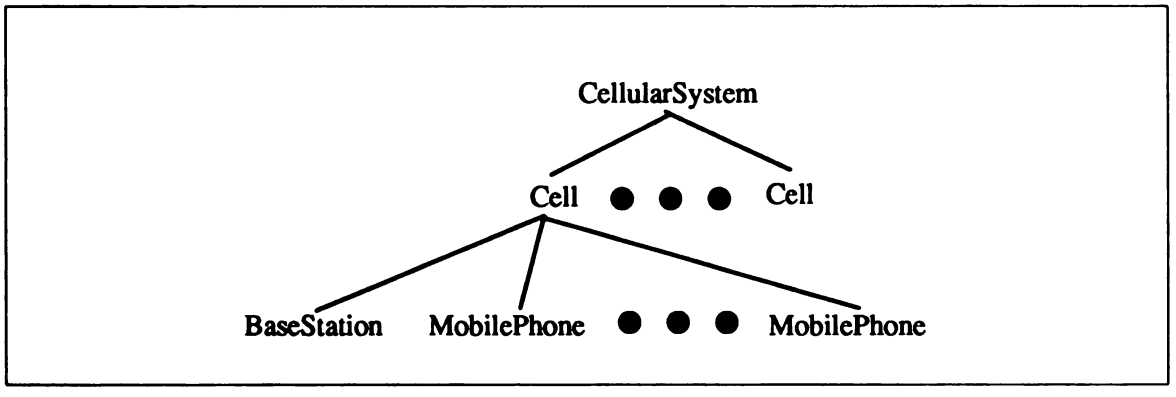

Figure 5.8. Example of a "part of" hierarchy.

group of logic gates are developed all of which inherit the base class *Gate*. The class *Gate* captures the generalities of logic gates once and for all, eliminating the need to reimplement those details every time a new logic gate is to be modeled. The "part of" hierarchy formed by the simulation model is used by the equation accumulation and organization mechanisms in order to efficiently respond to topological changes. The "part of" hierarchy can be used by the simulation layer to partition the representative equations into subsets of equations. The subset of equations defined within a particular *SubModel* is not changed by topological modifications outside the context of that *SubModel*.

5.3.2 The *EquationAccumulator* Object

The *EquationAccumulator* is the active object responsible for accumulating and organizing equations. Like the *ConnectionManager* and *EquationFormulator*, the *EquationAccumulator* executes the algorithm summarized in Figure 3.6 of Chapter 3. Every *SubModel* instance creates its own *EquationAccumulator* upon instantiation. The *EquationAccumulator* maintains an internal table of the variables registered by components that are contained in the *SubModel*, as well as variables the *SubModel* has registered. A second table contains all the equations, both constitutive and connective, which characterize the components and connections contained within the *SubModel*. Each variable is tagged with its component's identifier. Each equation is tagged with either a connection identifier or a component identifier, depending on whether it is a connectivity equation or a constitutive equation. These tags are the only details the *EquationAccumulator* has regarding specific connections and components, and are used only for reference purposes when components and connections are removed from the simulation. The *EquationAccumulator* does not need to record which components are connected. It only needs to have recorded the constitutive and connectivity equations which arise from components and their connections.

For every variable table entry the *EquationAccumulator* stores the following fields:

1. variable name - a string of characters which identifies the variable.

2. component identifier - the identifier of the component that defined the variable.

3. equation list - a list of the identifiers of the equations that the variable appears in.

4. solution flag - a flag which is set if a computation has been scheduled that will solve for this variable.

5. stored value - a value of the variable that was stored immediately before a topological change. These values are used as initial conditions in newly formulated systems of differential equations.

6. time stamp of stored value - the time the value was stored.

7. default value - used by applications to indicate initial conditions.

8. *SubModel* context - indicates which *SubModel* instance the variable is contained in.

9. parent *SubModel* context - indicates the parent *SubModel* instance of the *SubModel* containing the variable. This field is used by variables which are associated with terminals of nested *SubModels.*

10. internal formulation cache identifier - a reference to the internal formulation cache entry which contains this variable. Formulation caches are defined in Section 5.4

11. external formulation cache identifier - a reference to the external formulation cache entry which contains this variable.

For each equation table entry the *EquationAccumulator* stores the following fields:

1. equation identifier - an integer generated by the framework which uniquely identifies the equation.

2. component identifier - if the equation is a constitutive equation this field holds the identifier of the component that defined it.

3. connection identifier - if the equation is a connectivity equation this field holds the identifier of the connection it represents.

4. list of variables - this field contains a list of all the variables appearing in the equation. The list also associates an integer with each variable appearing in the equation, that is set to the order of the highest derivative of that variable. The integer is set to zero if no derivative of the variable appears in the equation. These order indicators are used to decide which type of equation solution mechanism is needed.

5. equation string - the string representation of the equation.

To facilitate the equation organization process, each variable entry in the variable table contains a list of equation identifiers that identify the equations in which the variable appears. Similarly, each equation entry in the equation table contains a list of the variables appearing in that equation. Figures 5.9 and 5.10 illustrate the two tables maintained by the *SubModel* instance which encapsulates the electrical example given in Section 3.2.6 of Chapter 3. A '-' in a table entry indicates the field is not applicable to that entry. Not all fields are shown.

Messages received by the *EquationAccumulator*

Within a given *SubModel* instance, the *EquationAccumulator* accepts five types of messages from the *ConnectionManager* and the *EquationFormulator*. Each of these

Variable Name	Component Identifier	Default Value	Equation Identifiers
$C_1.V_a$	1	-	1, 12
$C_1.V_b$	1	-	2, 16
$C_1.I_a$	1	-	2, 3, 11
$C_1.I_b$	1	-	3, 15
$C_1.I$	1	-	2
$C_2.V_a$	2	-	6, 12
$C_2.V_b$	2	-	6, 14
$C_2.I_a$	2	-	4, 5, 11
$C_2.I_b$	2	-	5, 13
$C_2.I$	2	-	6
$C_3.V_a$	3	-	8, 14
$C_3.V_b$	3	-	8, 16
$C_3.V$	3	0	7, 8
$C_3.I_a$	3	-	9, 10, 13
$C_3.I_b$	3	-	10, 15
$C_3.I$	3	-	7

Figure 5.9. Example variable table entries.

Equation Identifier	Component Identifier	Connection Identifier	Equation	List of Variables
1	1	-	$C_1.V_a - C_1.V_b = v$	$C_1.V_a, C_1.V_b$
2	1	-	$C_1.I - C_1.I_a = 0$	$C_1.I, C_1.I_a$
3	1	-	$C_1.I_a + C_1.I_b = 0$	$C_1.I_a, C_1.I_b$
4	2	-	$C_2.I - C_2.I_a = 0$	$C_2.I, C_2.I_a$
5	2	-	$C_2.I_a + C_2.I_b = 0$	$C_2.I_a + C_2.I_b$
6	2	-	$C_2.V_b - C_2.V_a - \mu C_2.I = 0$	$C_2.V_b, C_2.V_a, C_2.I$
7	3	-	$\frac{dC_3.V}{dt} - \frac{C_3.I}{c} = 0$	$C_3.V, C_3.I$
8	3	-	$C_3.V = C_3.V_a - C_3.V_b$	$C_3.V, C_3.V_a$
9	3	-	$C_3.I - C_3.I_a = 0$	$C_3.I, C_3.I_a$
10	3	-	$C_3.I_a + C_3.I_b = 0$	$C_3.I_a, C_3.I_b$
11	-	1	$C_1.I_a + C_2.I_a = 0$	$C_1.I_a, C_2.I_a$
12	-	1	$C_1.V_a - C_2.V_a = 0$	$C_1.V_a, C_2.V_a$
13	-	2	$C_2.I_b + C_3.I_a = 0$	$C_2.I_b, C_3.I_a$
14	-	2	$C_2.V_b - C_3.V_a = 0.0$	$C_2.V_b, C_3.V_a$
15	-	3	$C_1.I_b + C_3.I_b = 0$	$C_1.I_b, C_3.I_b$
16	-	3	$C_1.V_b - C_3.V_b = 0.0$	$C_1.V_b, C_3.V_b$

Figure 5.10. Example equation table entries.

messages will cause the *EquationAccumulator* to update its variable and equation tables.

1. variable registration message - This message is sent by the *ConnectionManager* when it encounters a variable definition in the *registration* method of a *Component.*

2. constitutive equation registration message - This message is sent by the *ConnectionManager* when it encounters the definition of an equation in the *registration* method of a *Component.*

3. connectivity equation registration message - This message contains a set of connectivity equations which represent a particular connection. It is sent to the *EquationAccumulator* by the *EquationFormulator* after it has formulated them from a set of terminals.

4. revoke constitutive equation message - This message is sent by the *ConnectionManager* after it has received a message to delete a particular *Component.*

5. revoke connectivity equation message - This message is sent by the *ConnectionManager* after it has received a message to add or remove a connection. When a terminal is added to an existing connection, the set of connectivity equations representing the old connection must be revoked and replaced with the equations representing the new connection.

In addition to these five messages, the *EquationAccumulator* also accepts two types of messages from the computational layer:

1. parse results message - contains the results of a request to parse a constitutive equation.

2. query result message - contains the results of a request to query a particular variable at a particular time.

Figures 5.11 - 5.17 outline the methods executed by the *EquationAccumulator* upon receiving these seven messages.

```
Input: variable table entry
1) Concatenate uniqueness information onto variable string
2) If variable is already in variable table
3)     error condition has occurred
4) else if variable is not in variable table
5)     insert variable into variable table
```

Figure 5.11. The *EquationAccumulator*'s method for variable registration.

```
Input: equation table entry
1) Insert equation into equation table
2) Send equation to MathSolver for parsing
```

Figure 5.12. The method for constitutive equation registration.

```
Input: a set of equation table entries
1) for each equation entry in the set
2)     insert equation entry in the equation table
3)     for each variable in the equation
4)         add equation to variable's list of equations
5) invalidate the solutions of all related variables
```

Figure 5.13. The method for connectivity equation registration.

```
Input: component identifier
1) remove all equations from the equation table which have a
   component identifier equal to the input component identifier
2) remove all variables from the variable table which have
   component identifier equal to the input component identifier
```

Figure 5.14. The method for revoking constitutive equations.

```
Input: connection identifier
1) invalidate the solutions of all related variables
2) For each entry in the equation table which has connection
   identifier equal to the input connection identifier
3)     for each variable referenced by the equation
4)         remove equation's identifier from the variable's
           entry in the variable table
5)     remove entry from equation table
```

Figure 5.15. The method for revoking connectivity equations.

```
Input: list of variables and order indicators
       equation identifier
1) copy list of variables and order indicators to the
   equation's table entry
2) for each variable V in the input list of variables
3)     retrieve V's entry from the variable table
3)     add the equation identifier to table entry's list
       of equation identifiers.
```

Figure 5.16. The method for handling results from an equation parse.

```
Input: the result of the query
1) forward result to the application layer.
```

Figure 5.17. The method for handling results form a variable query.

5.4 Equation Organization

Two alternative approaches regarding the question of when equations should be organized into systems and solved were considered. The first approach is to organize equations into systems and attempt to solve them whenever new equations are accumulated or existing equations are removed. The second approach is to organize and solve equations only when the application's demands for variable values necessitate it. The advantage of the first approach is that at any given time the equations are organized into systems and solutions for variables can be computed immediately. The disadvantage is that everytime the model's topology or component population changes, the equation set must be reorganized. The overhead associated with reorganizing and solving can be great, especially when symbolic solutions are being computed in the computation layer. The demand-driven approach was adopted. In this approach, equation organization is delayed until a variable query demands it. Changes in the equation set since the last equation organization that affect the variable being queried are taken into account before the variable value is computed.

5.4.1 Demand-Driven Equation Organization

The framework organizes equations into systems and solves them independent of the concepts of component and connection. Equation organization and solution is done only on demand [85]. When a topology change occurs, the *EquationAccumulator*'s equation table is updated with the appropriate connectivity equations, but the equations are not organized into systems and solved until the value of one of the variables is demanded. Demands for variable values can originate from two sources. First, an application can explicitly schedule query events such as the *SQueryEvent* and *SRptQueryEvent* events described in Figure A.9 of Appendix A. Second, certain topological changes may automatically trigger query events. If differential equations

are involved, initial conditions are established via queries immediately before new connectivity equations are registered.

5.4.2 Queries for Variable Values

A query event results in a message being sent to the *EquationAccumulator* with which the variable being queried is registered. The message contains a reference to the variable to be queried and the time of the query. Upon receiving the message, the *EquationAccumulator* locates the variable's entry in the variable table. If the solution flag is set, the query is forwarded on to the computational layer. If the solution flag is not set, the system of equations containing this variable has not yet been organized and solved. Before sending the query on to the computation layer, the system of equations containing the variable must first be organized and passed on to the computation layer for solving. The *EquationAccumulator*'s method for processing query messages is summarized in Figure 5.18.

```
Input: variable reference, and time of query
1) retrieve variable's entry from the variable table
2) if variable's solution flag is not set
3)     initiate equation organization and solution mechanisms
4) send query message to MathSolver
```

Figure 5.18. The *EquationAccumulator*'s query method.

5.4.3 Cooperative Organization Via Active Objects

To organize the equations and variables related to a given variable within its *SubModel* context, the *EquationAccumulator* applies the closure algorithm which is given in Figure 5.19. The algorithm begins with a single variable and iteratively collects all the equations and variables from the tables that are either directly or transitively related to the variable. In some cases an *EquationAccumulator*'s local variable and equation tables will yield a system of equations which can be solved independent of external components and connections. This is the case for the example tables given in Figure 5.9 and Figure 5.10. However, in models where *SubModels* are nested within *SubModels*, some variables may appear in equations which are external to the local *EquationAccumulator*. In this case, the local *EquationAccumulator* sends messages to the external *EquationAccumulators* instructing them to apply the closure algorithm to their equation sets. The external *EquationAccumulators* apply the closure algorithm to their local equation and variable tables concurrently, propagating messages to other *EquationAccumulators* as necessary. When an *EquationAccumulator* finishes computing closure for the requested variables, it will wait for messages from the *EquationAccumulators* it has sent messages to. If there is an *EquationAccumulator* involved that is higher in the "part of" hierarchy formed by the nested *SubModels*, the *EquationAccumulator* will send a message containing the equations it has gathered to that *EquationAccumulator*. Eventually, the *EquationAccumulator* highest in the hierarchy of participating *EquationAccumulators* will send the complete system of equations on to the computational layer.

Detecting External Equations

Variables that are associated with the terminals of nested *SubModels* must be registered within the *SubModel* that defines them as well as with the *SubModel* containing that *SubModel*. Consider the simple model illustrated in Figure 5.20. The compo-

```
Input:
    Variable vin
Output:
    a set of equations and variables related to vin
Local Variables:
    unXvars - set of unexpanded variables
    Xvars - set of expanded variables
    equs - set of equations

 1) insert vin into unXvars
 2) while unXvars is not empty
 3)     select a variable V from unXvars
 4)     for each equation E whose identifier appears in V's list
        of equation identifiers
 5)         if E is not already in equs
 6)             for each variable V' in E's list of variables
 7)                 if V' is not in unXvars or Xvars then
 8)                     insert V' in unXvars
 9)             insert E into equs
10)     remove V from unXvars
11)     insert V into Xvars
```

Figure 5.19. The closure algorithm.

nent identifiers appear in the lower right corner of each block. The black dots indicate connections. The world submodel C_1 contains components C_2 and C_3. C_3 is a primitive component, and C_2 is a submodel containing a single primitive component C_4. Components C_2, C_3, and C_4 each have a single terminal. Each terminal has a single variable v associated with it. Assuming that the connection rule will generate only identity equations, the following equations will be formulated to represent the two connections:

$$C_3.v = C_2.v \tag{5.1}$$

$$C_2.v = C_4.v. \tag{5.2}$$

Since C_1 and C_2 are instances of the *SubModel* class, both of them will contain their own *ConnectionManager, EquationFormulator* and *EquationAccumulator* instances. The connection between C_3 and C_2 is in the context of submodel C_1, and is registered with C_1's *EquationAccumulator.* Thus, C_1's *EquationFormulator* and *EquationAccumulator* formulate and accumulate equation 5.1, and C_2's *EquationFormulator* and *EquationAccumulator* formulate and accumulate equation 5.2. Since the variable $C_2.v$ is referenced by equations in both C_1 and C_2, it must be registered within both contexts. When the application assigns the variable $C_2.v$ to C_2's terminal, the framework registers it in the context of submodel C_1, as well as in the context of submodel C_2. In addition, the parent *SubModel* context field in the variable table entry of both *EquationAccumulator*'s is set to C_1. When executing the closure algorithm, *EquationAccumulators* will anticipate external equations whenever a variable is encountered that is using both of its *SubModel* context fields.

Equation Formulation Caches

In hierarchical models a single topological change may make it necessary to reorganize the systems of equations. Only the equations within the *SubModel* context containing

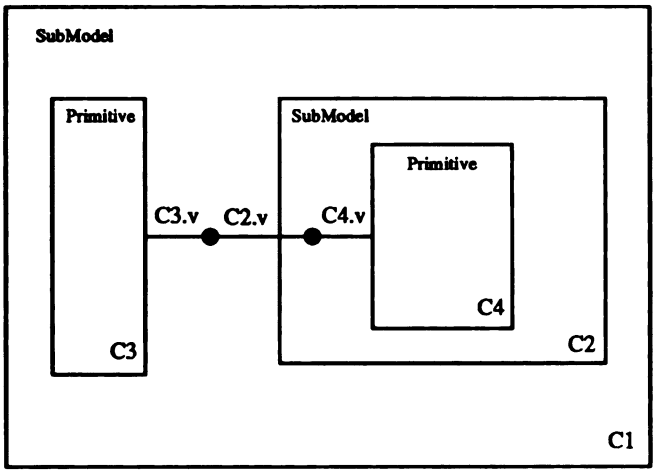

Figure 5.20. Nested *SubModels* and their variables.

the topological change are affected. If *SubModel* contexts external to or encapsulated within the *SubModel* context containing the topological change have previously applied the closure algorithm to their variable and equation tables, it is not necessary for them to apply the algorithm again. Every time the closure algorithm is executed, the equations and variables collected are placed in a formulation cache. A cache entry is valid until a topological change modifies the set of equations it contains. When an *EquationAccumulator* receives a message requesting it to compute closure for a certain set of variables it will first check for entries in the formulation cache that contain these variables. If entries exist, their contents can be sent back to the requesting *EquationAccumulator* without executing the closure algorithm. Each variable that is listed in a formulation cache entry has its internal formulation cache field updated to refer to that cache entry. Variables that are registered in an external context – those associated with the terminals of a nested *SubModel* – may also have their external formulation cache field refer to an external formulation cache entry. Each formulation cache entry contains the following fields:

1. identifier - an unique identifier which is generated by the framework.

2. set of variables - the set of variables which will appear in the same system of equations.

3. set of equations - the set of equations which reference only those variables found in the entry's set of variables.

4. set of remote formulation entries - indicate the possibility of remote equations and variables.

A remote formulation entry consists of the following fields:

1. remote *SubModel* context - the remote *SubModel* context which may contain related variables and equations.

2. set of variables - the set of variables which are also registered in the remote *SubModel* context.

3. parent flag - set if the remote *SubModel* contains the current *SubModel*, and reset otherwise.

While applying the closure algorithm, an *EquationAccumulator* groups variables with external references according to their *SubModel* context. Each of these groups form a remote formulation entry which is stored in the formulation cache entry created from the output of the closure algorithm.

The algorithm executed by the *EquationAccumulator*s that are cooperatively organizing the system of equations is outlined in Figure 5.21. This algorithm is executed whenever an *EquationAccumulator* receives an organization message. It is initiated in step 3 of the query method listed in Figure 5.18, with the input remote formulation entry containing the single variable to be queried.

The algorithm is best understood with an example. Three primitive component definitions, A, B, and C, are shown in Figure 5.22. The figure shows the terminals,

```
Input:
    remote formulation entry -  RFE
    invoking context - Icontext
    list of formulation cache entries - Flist
Local Variables:
    flag - parent (reset by default)
    remote formulation entry - ParentEntry
    list of formulation cache entries - Flistnew
    local SubModel context - this

 1) for each variable V in RFE
 2)     if V has a formulation cache entry associated with it,
        and this entry is not yet in Flist
 3)         record the formulation cache entry in Flist
 4)     else if there is no formulation cache entry
        associated with V
 5)         invoke the closure algorithm for V
 6)         form a new formulation cache entry with output from
            the previous step and insert it in Flistnew.
 7) for each remote formulation entry R contained within
    the entries of Flistnew
 8)     if R leads to the parent SubModel then
 9)         set flag parent
10)         save R in ParentEntry
11)     else if R leads to an unexplored child SubModel then
12)         send organize message to the EquationAccumulator
           in  R's context with  parameters (R, this, NULL)
13) wait for all EquationAccumulators initiated in children
    SubModels to return their results.  Accumulate all results
    in Flistnew.
14) append Flistnew to Flist
15) if parent flag is set
16)     if Icontext is not equal to the parent context
17)         send organize message to the EquationAccumulator
           in the parent SubModel context with parameters
           (ParentEntry, this,Flist)
18)     else if Icontext is equal to the parent context
19)         send message containing all the results in Flist
           to the parent SubModel.
20) else if parent flag is reset
21)     format solve command and submit to computational layer.
```

Figure 5.21. The equation organization algorithm.

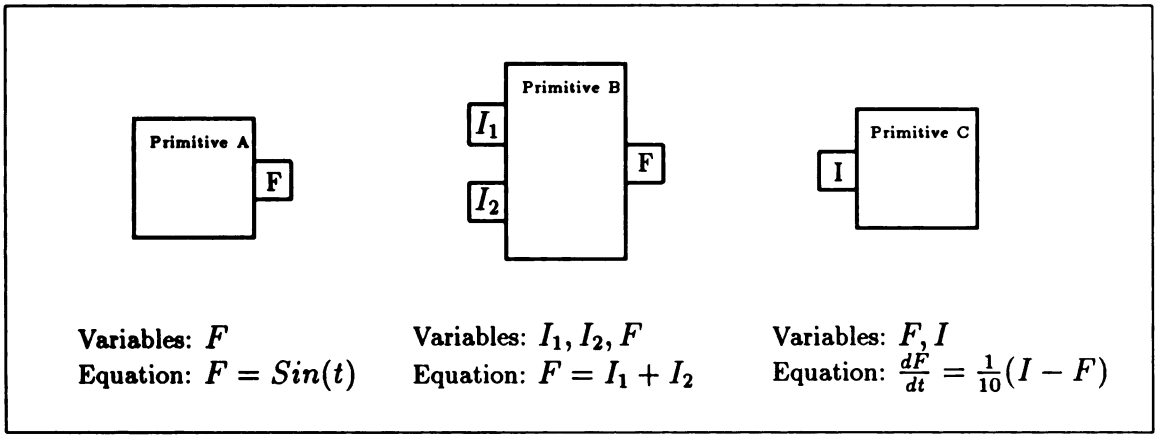

Figure 5.22. Primitive component definitions.

variables and constitutive equations defined by each of the components. The terminals are labeled with the single variable that is assigned to them. The component of type A has one terminal defined to which the variable F is assigned. The component of type B has three terminals defined to which the variables I_1, I_2, and F are assigned. The component of type C has one terminal defined to which the variable I is assigned.

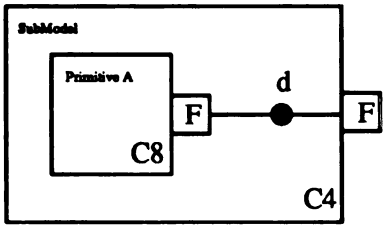

Figure 5.23. An instance of component type A nested within a *SubModel*.

A simple hierarchical model is assembled from these primitive components as follows. An instance of component type A, C_8, is nested within *SubModel* C_4, as shown in Figure 5.23. *SubModel* C_4 contains a single connection (labeled d in the

figure) that connects component C_8's terminal with component C_4's terminal. The single constitutive equation:

$$C_8.F = sin(t) \tag{5.3}$$

arises from C_8's definition and is registered within the context of *SubModel* C_4. Assuming the equality equation connection rule, the connectivity equation:

$$C_8.F = C_4.F \tag{5.4}$$

is generated from connection *d*. This equation is also registered within the context of *SubModel* C_4. An instance of component type C, C_7, is nested within *SubModel* C_3, as shown in Figure 5.24. The constitutive equation arising from component C_7 is:

$$\frac{dC_7.F}{dt} = \frac{1}{10}(C_7.I - C_7.F) \tag{5.5}$$

and the connectivity equation formulated from connection *g* is:

$$C_3.I = C_7.I. \tag{5.6}$$

Both Equation 5.5 and Equation 5.6 are registered within the context of *SubModel* C_3. *SubModel* C_2 shown in Figure 5.25 contains C_5, an instance of component type

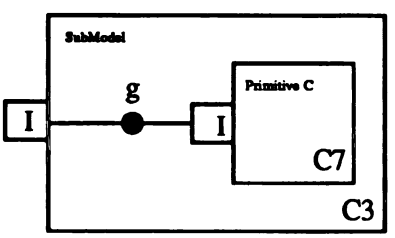

Figure 5.24. An instance of component type C nested within a *SubModel*.

A, C_6, an instance of component type B and the *SubModel* C_4 which is shown in Figure 5.23. The constitutive equations registered in the context of *SubModel* C_2:

$$C_5.F = sin(t) \tag{5.7}$$

$$C_6.F = C_6.I_1 + C_6.I_2 \tag{5.8}$$

arise from components C_5 and C_6, respectively. *SubModel* C_2 contains three connections which are labeled *a*, *b*, and *c* in the figure. The connection equations formulated from these connections and registered in the context of C_2 are:

$$C_4.F = C_6.I_1 \tag{5.9}$$

$$C_5.F = C_6.I_2 \tag{5.10}$$

$$C_6.F = C_2.F. \tag{5.11}$$

Note that *SubModel* C_2 has no information regarding the components and connections within *SubModel* C_4. The highest *SubModel* in this model's hierarchy, C_1, is shown in Figure 5.26. This *SubModel* contains *SubModel* C_2 and *SubModel* C_3, and a connection (labeled *f* in the figure) that connects their terminals. The connectivity equation formulated from this connection and registered in the context of *SubModel* C_1 is:

$$C_2.F = C_3.I. \tag{5.12}$$

There are no constitutive equations registered in the context of *SubModel* C_1. Figure 5.27 shows the complete model hierarchy (minus connections) of the example model.

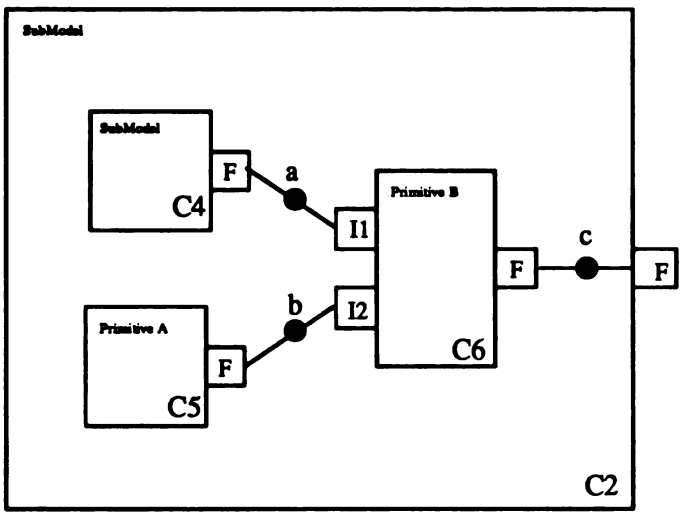

Figure 5.25. A *SubModel* containing two primitive components and a nested *SubModel*.

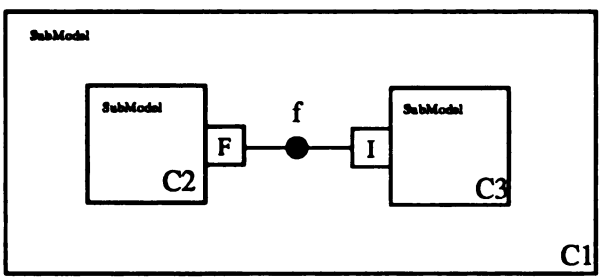

Figure 5.26. The highest *SubModel* in the hierarchy.

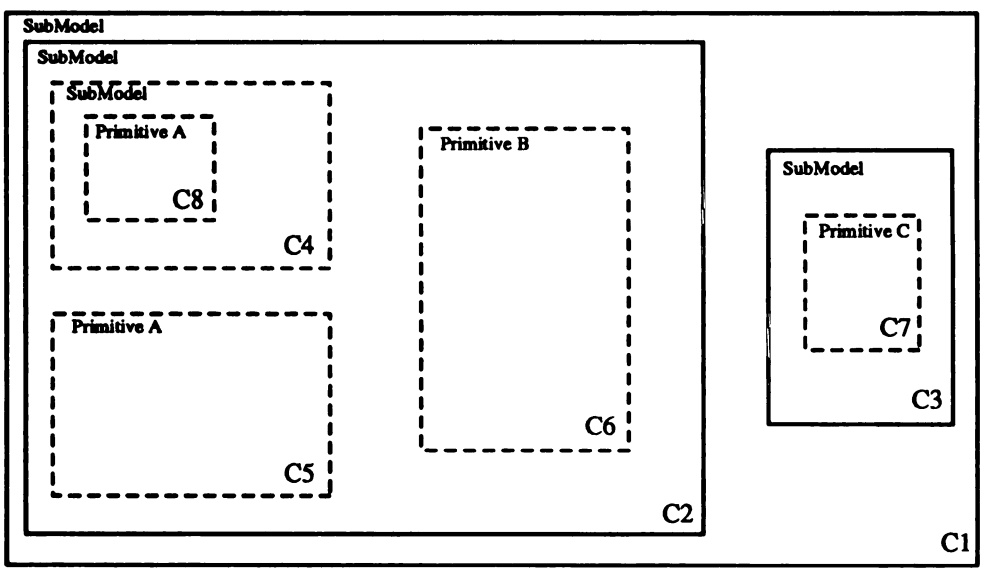

Figure 5.27. Example model hierarchy.

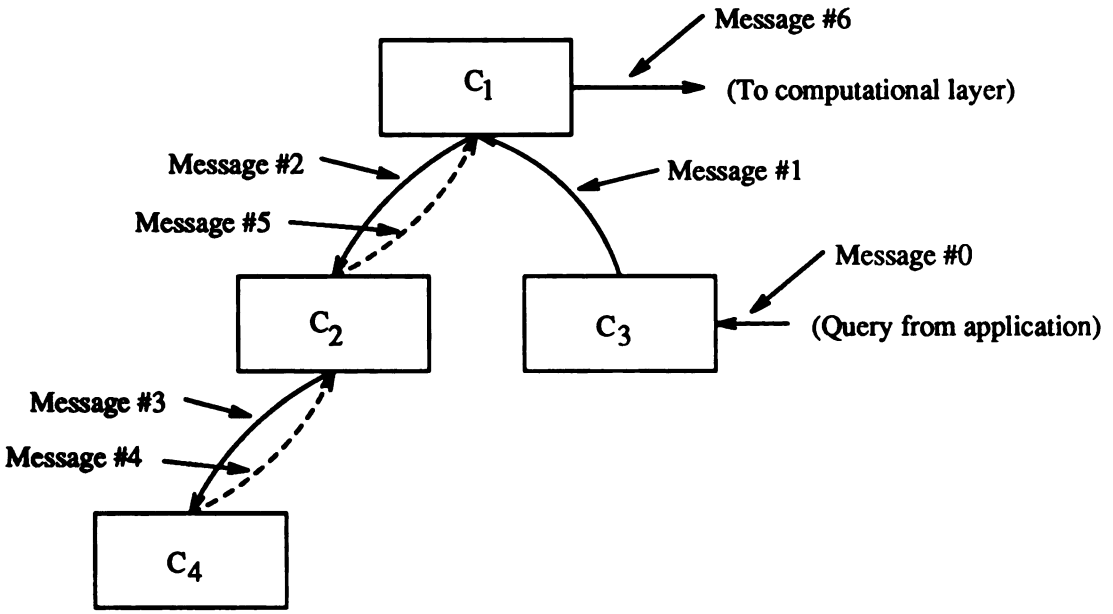

Figure 5.28. Messages passed during equation organization.

Figure 5.28 illustrates how messages are sent among the *EquationAccumulators* when organizing a system of equations to be solved in response to a query. Initially, it is assumed that no queries have been made. No formulation cache entries exist and the equations have not yet been organized into systems or solved. Some time later it is assumed that a query occurs for variable $C_7.F$. The query is made by calling the *queryVar* method of *SubModel* C_3 which sends a message to C_3's *EquationAccumulator*. This message is labeled #0 in Figure 5.28. Upon receiving the message the *EquationAccumulator* will execute the method outlined in Figure 5.18. Since no solution exists yet for $C_7.F$, step 3 in the algorithm will cause the organization algorithm in Figure 5.21 to be executed. Steps 1-6 will result in the creation of a formulation cache entry containing the variables: $C_7.I$, $C_7.F$, $C_3.I$, and the equations: 5.5, 5.6. The formulation cache entry will contain one remote formulation entry with remote *SubModel* context set to C_1, the set of variables set to $C_3.I$, and its parent flag set. Since there are no child *SubModels* in this context, no organize messages are sent out

or waited for in steps 7-14. In step 14, the single formulation cache entry generated in step 6 is added to Flist. In step 19 an organize message, message #1 in Figure 5.28, is sent to C_1's *EquationAccumulator*. The message will contain the equations and variables from the formulation cache entry , the remote formulation entry, and C_3 as the invoking context.

When C_1's *EquationAccumulator* receives the message, it will apply the organization algorithm to its local equation and variable tables. In steps 1-6, C_1's *EquationAccumulator* will generate a single formulation cache entry that consists of Equation 5.12 and two remote formulation entries leading to *SubModels* C_2 and C_3. In steps 7-12, the only remote formulation entry leading to an unexplored nested *SubModel* is the one leading to C_2. C_1's *EquationAccumulator* will send an organize message, message #2 in Figure 5.28, to C_2's *EquationAccumulator*. In step 13, C_1's *EquationAccumulator* will wait for a message from C_2's *EquationAccumulator* in response to the message it sent in step 12.

Upon receiving its organize message, C_2's *EquationAccumulator* applies the organization algorithm. In steps 1-6 a new formulation cache entry is created from the output of the closure algorithm. The cache entry consists of Equations 5.7, 5.8, 5.9, 5.10, 5.11 and two remote formulation entries which lead to *SubModels* C_1, and C_4. In steps 7-12, the only remote formulation entry leading to unexplored nested *SubModels* is the one leading to C_4. An organize message is sent to the *EquationAccumulator* of C_4. This message is labeled #3 in Figure 5.28. In step 13, C_2's *EquationAccumulator* will wait for a message from the *EquationAccumulator* of C_4.

After receiving the organize message from C_2's *EquationAccumulator*, the *EquationAccumulator* of C_4 invokes the organization algorithm. In steps 1-6 C_4's *EquationAccumulator* creates a formulation cache entry which contains Equations 5.3 and 5.4, and a remote formulation entry that leads to *SubModel* C_2. Since C_4 does not

contain unexplored nested *SubModels*, no messages are sent or waited for in steps 7-13. In step 19 C_4's *EquationAccumulator* sends a message containing the equations it gathered to its parent *SubModel* C_2's *EquationAccumulator*. This message is labeled #4 in Figure 5.28. C_2's *EquationAccumulator* unblocks in step 13 and sends a message to the *EquationAccumulator* of C_1 containing the equations it has gathered, plus those it received from C_4 and C_5. This message is labeled #5 in Figure 5.28. Upon receiving the message from C_2's *EquationAccumulator* C_1's *EquationAccumulator* unblocks in step 13. Since C_1 has no parent *SubModel*, the condition in step 15 fails, and step 21 is executed. In step 21, all 10 equations gathered are sent to the computational layer for solution. This message is labeled #6 in Figure 5.28.

Invalidating Solutions

Whenever a topological change occurs, all variables related either directly or indirectly to the variables referenced in the connectivity equations must have their solution flags reset. When there are related variables external to the *SubModel* containing the topological change, messages must be sent to the external *EquationAccumulator* instructing it to reset those variables' solution flags. The external *EquationAccumulator* may in turn send messages to other *EquationAccumulators*. This process continues until all pertinent variable solution flags are reset. Note that all formulation caches external to the *SubModel* containing the topological change remain intact. Resetting the solution flags will force a new system of equations reflecting the topological change to be submitted to the computational layer the next time a variable affected by the change is queried.

5.5 Equation Solution

The requirements placed on the computation layer are dictated by the application. The constitutive and connectivity equations defined and generated by some applications may consist of only algebraic equations, while other applications may define differential equations as well. It is unreasonable to expect the framework to anticipate every type of equation that applications could possibly define. The framework defines an abstraction that serves as an interface between the simulation and computational layers. Communication between the simulation and computational layers must conform to this interface. If an application defines equations that a particular framework implementation cannot solve, the computational layer implementation can be replaced by a new implementation without affecting the simulation, interface, or application layers.

5.5.1 The *MathSolver* Object

From the perspective of the simulation layer, the computational layer is represented by a single object of class *MathSolver*. The *MathSolver* object is global in the sense that all *EquationAccumulators* may send it messages. Since they will vary by application, the internal details of the *MathSolver* are left unspecified. The framework only defines the messages it receives and their semantics. Thus, the *MathSolver* could process the messages it receives by distributing the computations across a network of mathematical servers completely transparent to the simulation layer. The *MathSolver* object accepts six types of messages:

1. *EquationAccumulator* registration,

2. parse equation,

3. solve system,

4. query for value,

5. define constant, and

6. remove constant.

When a *SubModel* is instantiated, it must register its *EquationAccumulator* with the *MathSolver* by sending the *MathSolver* an *EquationAccumulator* registration message. The registration message establishes the identity of the sending *EquationAccumulator* with the *MathSolver*. The *MathSolver* needs to know the identity of every *EquationAccumulator* in order to send messages to them when computations they request are completed.

The decision regarding which framework layer would be responsible for parsing constitutive equations was based on modularity. The responsibility of parsing constitutive equations was delegated to the computation layer. Coupling the parsing mechanism with the solution mechanism allows a *MathSolver* to be replaced with another *MathSolver* that can recognize and solve equations which the former could not. *MathSolvers* can be replaced without modifying the interface and simulation layers of the framework. The *parse equation* message sent by an *EquationAccumulator* (step 2 in Figure 5.12) contains a sequence of characters which represents the equation, and an identifier indicating which *EquationAccumulator* is requesting the equation to be parsed. The *MathSolver* will parse the equation and send a message back to the *EquationAccumulator* containing the results. This message contains a list of all variables that appear in the equation. This information is used by the *EquationAccumulator* to organize systems of equations from its internal variable and equation tables. Each variable returned in the message from the *MathSolver* also has an integer associated with it. This integer indicates whether or not derivatives of the variable appeared in the equation. A message containing the variable v_1 and a corresponding integer value set to two indicates that the second derivative of v_1 appeared in the equation.

Implementation dependent results may also be returned from the parse.

The *solve system* message is sent to the *MathSolver* by an *EquationAccumulator* whenever the solution flag of a queried variable is not set (steps 2 and 3 in Figure 5.18). The message contains a command string that indicates which type of computation is to be performed, and which variables are to be solved for. The framework generates two different command strings. The first command is the *Solve* command, and is used for solving systems of algebraic equations. The syntax for this command is:

```
identifier=Solve[
                {equations},
                {variables}
        ]
```

The second command is used for solving systems of equations that contain both algebraic and differential equations. The syntax for this command is:

```
identifier=DSolveMix[
                    {equations},
                    {variables},
                    {initial conditions},
                    time
        ]
```

The format for the **{equations}** parameter is implementation dependent. The **{initial conditions}** parameter gives the initial conditions for the differential equations at the time indicated by the parameter **time**. The **identifier** is used to identify the solution of the equations. The *MathSolver* maintains a table that relates variables to the solution identifiers. Whether the solutions are symbolic or numeric is also implementation dependent. In the former case, a solution would be a set of rules that maps variables to symbolic expressions based on simulation time

and system constants. In the latter case, the solution may consist of a set of state equations which are iteratively evaluated whenever variables are queried.

The *query for value* message is sent to the *MathSolver* by an *EquationAccumulator* whenever a variable registered with that *EquationAccumulator* is queried by an application (step 4 of Figure 5.18). The message contains the name of the variable and the simulation time at which it is to be evaluated. If the *MathSolver* has solved a system of equations containing the variable, it will use that solution to compute the numerical value of the variable. This value is encapsulated into a message and sent back to the requesting *EquationAccumulator*. If the *MathSolver* cannot solve for the variable, it simply sends a message back to the *EquationAccumulator* indicating solution was available. This situation may arise when the set of constitutive and connectivity equations does not provide enough information to solve for every variable.

In many simulation applications, it is common for certain numerical quantities to remain fixed during simulation. If an electrical system is being modeled, the resistance across a particular resistor component is usually considered to be a constant value. When defining constitutive equations it may be desirable to refer to these constants with a symbol. This can be accomplished by registering an additional variable and constitutive equation for each of these constant values. For the resistor component, a variable R could be registered along with the constitutive equation:

$$R = 100. \tag{5.13}$$

The constitutive equation relating current and voltage could then be registered as:

$$v = i * R \tag{5.14}$$

instead of,

$$v = i * 100. \tag{5.15}$$

The model of the resistor could represent variable resistance by replacing Equation 5.13 with the desired initialization of the variable R. This modification of the equation set however, will cause the current solution (if any) to be invalidated, and requires a new system of equations to be organized and solved. In systems where symbolic solutions are possible, the need for reorganizing and re-solving can be eliminated if the constant value can be changed without changing the current set of equations. The framework supports such a mechanism by allowing the application to explicitly define constants. An application can send a *define constant* message to the *MathSolver* which establishes the value for a symbolic constant. For the resistor component, the application sends a define constant message to the *MathSolver* instead of registering Equation 5.13 as a constitutive equation. The *MathSolver* then establishes R as a symbolic constant with the value 100. The constant can be redefined by sending the *MathSolver* a *remove constant* message followed by a define constant message containing the new value. In models where the computational layer computes symbolic solutions for variables, the systems of equations remain the same and do not need to be reorganized and re-solved when a symbolic value is redefined. The support for symbolic constants by the computational layer is especially useful for interactive simulation applications, in which users can make arbitrary changes to constant values during simulation execution.

5.6 Summary

This chapter has defined the objects which are responsible for managing connections and components, and formulating connectivity equations. These objects, the *ConnectionManager* and the *EquationFormulator* have been defined in terms of the types

of messages they send and receive, and the methods they execute when messages are received. This chapter has also presented the mechanisms by which the framework accumulates, organizes, and solves the connectivity and constitutive equations. Terminology was developed regarding object-oriented hierarchies and how they relate to the framework. The *EquationAccumulator* object, which is responsible for the accumulation and organization of equations was defined in terms of the messages it receives and sends. From the perspective of the simulation layer, the interface to the computational layer is via a single *MathSolver* object. The *MathSolver* encapsulates methods for parsing constitutive equations and solving the systems of equations organized by the *EquationAccumulator*s.

CHAPTER 6

Applications

This chapter presents an implementation of the framework and models that demonstrate the applicability of the framework in several different areas. A model of the control channels in a cellular phone network demonstrates the framework's capability for modeling discrete systems where components and connections appear and disappear over time. A model of an electrical circuit containing both continuous and discrete components demonstrates the framework's capability for continuous/discrete simulation and reusability via object-oriented inheritance. A third model of the population dynamics of a beehive, demonstrates the framework's utility in modeling biological and ecological systems.

6.1 The FMDC Framework

FMDC (Framework for Modeling Dynamic Connections) is an implementation of the simulation framework presented in this dissertation. It was implemented in the C++ programming language [86]. The concurrent objects were implemented using the AT&T C++ task library [87]. The AT&T task library is an efficient tasking system with non-preemptive scheduling and real-time control facilities for responding to external events [84]. The computational layer was implemented using Mathematica, a

software system for numeric and symbolic mathematical computation [100]. A system based on Mathematica consists of two parts: the *kernel* and the *frontend.* The kernel is the software that carries out the mathematical computations. The frontend is the software that handles interaction with the user. Mathematica's *MathLink* standard is the communication standard between the frontend and the kernel [101]. The FMDC implementation conforms to the MathLink standard, allowing it to interface directly to the Mathematica kernel. A substantial amount of programming was necessary in the Mathematica programming language in order to manipulate the equations into a form suitable to Mathematica's built-in equation solving commands. Communication between the simulator and the computational layer is implemented using the UNIX *socket* mechanism [5]. This mechanism hides the details of the lower level network protocols, and allows a process to communicate with another process running on a remote machine. An additional layer of software was developed on top of the socket mechanism. This layer hides the socket data structures and system calls with a set of C++ classes.

The design of FMDC is given in Figure 6.1. The figure shows an example model and its relationship to the *MathSolver*, *EventHandler* and Mathematica kernels. The circle includes the components and submodels derived by the user from the base classes provided by the framework. Each shaded rectangle is a primitive component. Each unshaded rectangle is a submodel containing its own *ConnectionManager*, *EquationAccumulator*, and *EquationFormulator* objects, as shown for the "world submodel". The portion of the figure enclosed in the dotted-line rectangle is implemented by a single Unix process. In this particular model, the process consists of 20 concurrent tasks - three for each of the six submodels, one for the *MathSolver*, and one for the *EventHandler.* The *MathSolver* has a connection to another process which is labeled "Solver Interface". The solver interface process is responsible for initiating and communicating with the Mathematica kernels on remote hosts. If multiple Math-

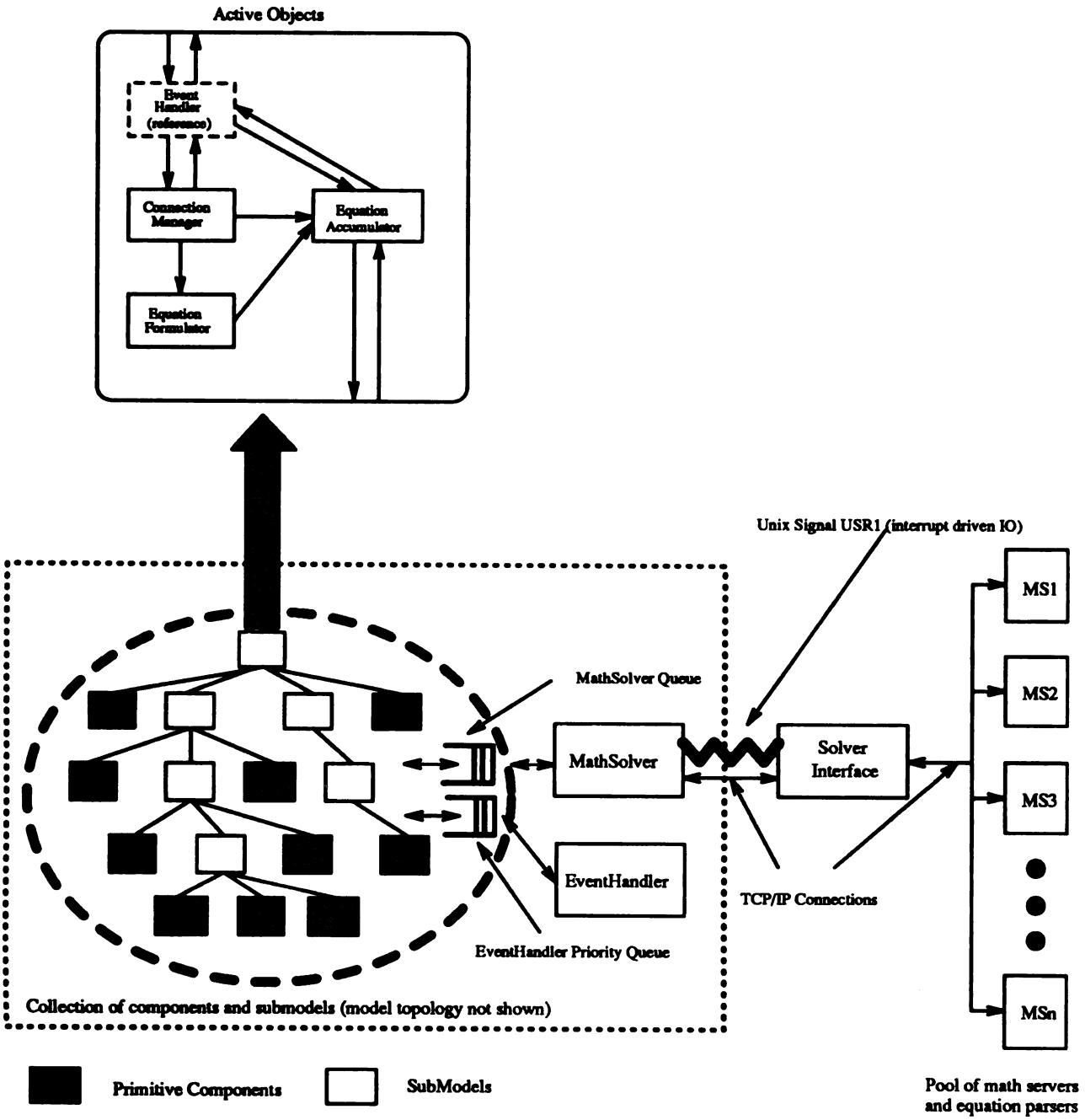

Figure 6.1. The FMDC implementation.

ematica kernels are running, the solver interface must also maintain information on which server a computation has been assigned to. Communication between the solver interface process and the *MathSolver* is asynchronous. The *MathSolver* is interrupted by the solver interface whenever the result of a computation is ready. The task library provides mechanisms for handling interrupts. Synchronization among the concurrent tasks enclosed in the dotted-line rectangle is achieved by a global time clock provided by the task library. Synchronization with the solver interface is achieved by forcing all query events to be blocked until all pending equation parsing and solution requests are completed.

Simulation applications are developed by deriving new classes from the abstract classes, *Component*, *SubModel* and others that are defined by the framework. New terminal types can be developed by deriving new classes from the base class *Terminal*. The framework user need only be familiar with the the applications layer. All other layers are hidden from the user. It is possible to build additional layers on top of the applications layer. A graphical user interface could be built on top of the applications layer for example. Users could then build models using a graphical editor without being required to know or understand the abstract classes defined by the framework or the syntax of the language the framework is implemented in. The examples included in this chapter are written in the applications layer.

6.2 A Cellular Phone Model

A cellular phone system is a two-way radio system which allows mobile users to connect to the public phone system. These systems are called "cellular" because of the way in which the service areas are divided into smaller areas called cells. Each cell contains a base station that the mobiles in the cell communicate with to access the public phone network. In the remainder of this section, we will examine how

the FMDC framework can be used to study a particular aspect of a cellular system, the common control channel. The model which will be described here is a simplified version of a model developed by Engelsma and Reilly [37]. The model was originally designed to develop a better understanding of the common control channels in the new Pan-European digital cellular network.

6.2.1 Background Information

The common control channel is a radio channel shared by the mobiles and base station within a particular cell. Among other things, the common control channel is used by the base station and mobiles as follows:

- paging - The base station uses the common control channel in the downlink direction (base station to mobile) whenever it wishes to inform a mobile that somebody wants to connect to it.

- channel allocation - The base station uses the common control channel in the downlink direction whenever it wishes to inform a mobile it has been allocated a dedicated channel.

- channel request - the mobile uses the common control channel in the uplink direction (mobile to base station) whenever it wishes to request a dedicated channel to make a connection on.

The common control channel is split up into three logical channels as illustrated in Figure 6.2. The *paging subchannel* (PCH) is used by the base station in the downlink direction to notify a mobile that a connection is desired. The *random access subchannel* (RACH) is used by mobiles in the uplink direction whenever a mobile requests a channel. A mobile will make a channel request whenever it needs to perform a location update (telling the network where it is), when it is paged, or

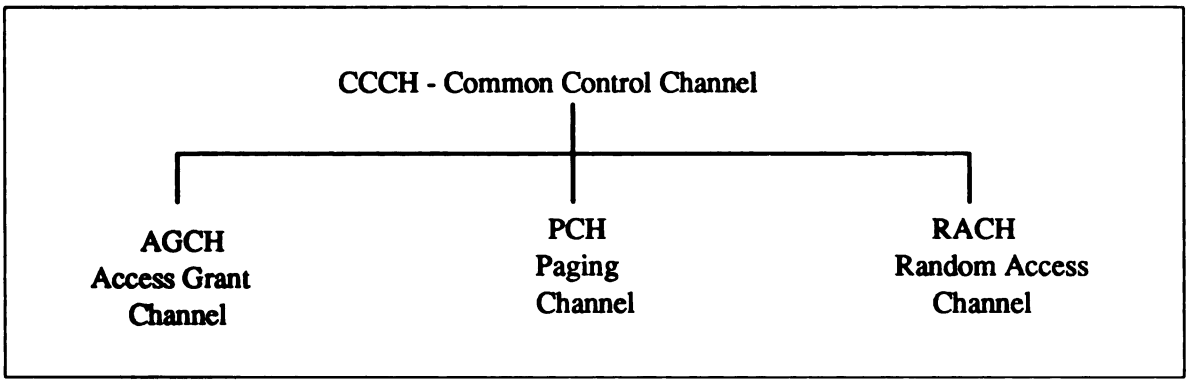

Figure 6.2. The common control channels.

when it is going to originate a call. The *access grant subchannel* (AGCH) is used by the base station in the downlink direction for notifying a mobile whether or not its channel request (issued via the RACH) can be satisfied. A common control channel may be assigned to a dedicated physical radio channel, or it can share a physical channel with other types of control channels. There are also system parameters for specifying how much of the channel should be divided between the AGCH and PCH subchannels, configuration of the PCH, number of retries on the RACH, etc., all of which effect the overall system performance. The common control channels are used only for control purposes. Mobiles use dedicated data channels when they are engaged in a call.

6.2.2 The Cellular Model

Dynamic connections are commonplace in cellular systems. Upon entering a cell, a mobile uses the common control channel to communicate with the base station in that particular cell. When the mobile exits the cell, it no longer communicates with that base station. In a busy metropolitan area, the cellular network may consist of hundreds of cells and thousands of mobiles. In addition, new mobiles will power-on and

become part of the system, while existing mobiles may power-off and disappear from the system. The abstractions for components, terminals, and connections, provided by the FMDC framework are useful in modeling this type of system.

Cells, BaseStations, Mobiles, and Events

A single cell of the system is modeled here. The system is decomposed into three different object types: *Cell*, *BaseStation*, and *Mobile*. Since cells contain base stations and mobiles, the *Cell* class is derived from the *SubModel* class, while the *BaseStation* and *Mobile* classes inherit the *Component* class directly. The "kind of" and "part of" hierarchies for the model are shown in Figures 6.3 and 6.4.

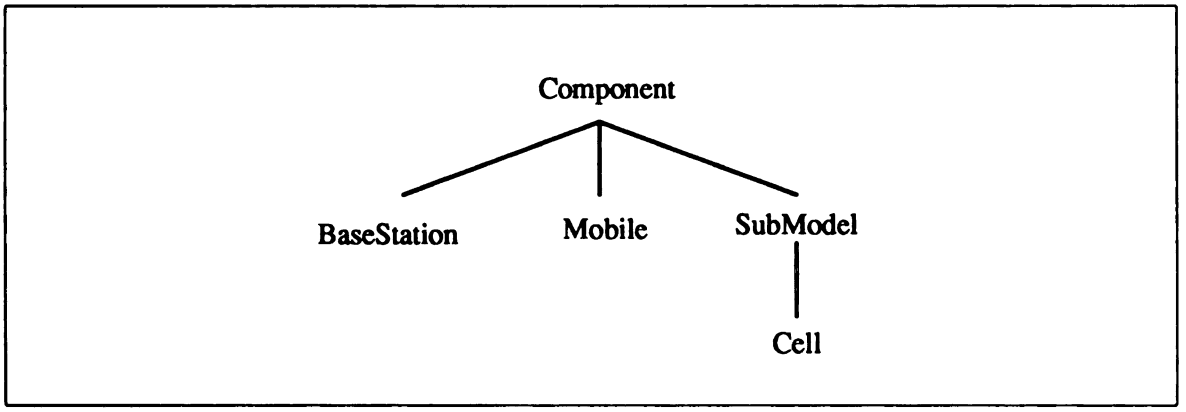

Figure 6.3. The "kind of" hierarchy for the cellular model.

The *Mobile* class defines two terminals. The terminals consist of a single variable partition and use the identity equation connectivity rule. The variable *downlinkChannel* is associated with the first terminal, and the variable *uplinkChannel* is associated with the second terminal. Similarly, the *BaseStation* also has two terminals of this same type and associates *downlinkChannel* and *uplinkChannel* variables

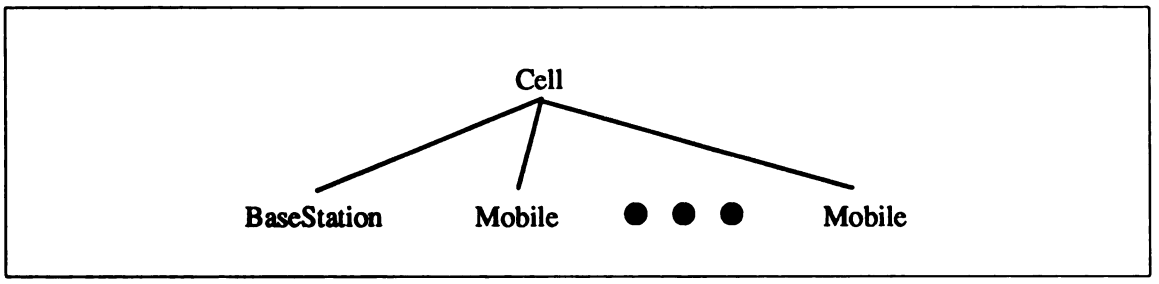

Figure 6.4. The "part of" hierarchy for the cellular model.

with them. The terminals and variables of the mobiles and base station represent the bi-directional common control channel described in the previous section.

A mobile may be in one of three states: *busy*, *idle*, or *requestPending*. The mobile is *idle* if it is not engaged in a call. A mobile is *busy* if it is engaged in a call. A mobile is in the *requestPending* state if it has requested a dedicated channel from the base station but has not yet been granted one.

The *input* and *output* methods (see Section 4.2 of Chapter 4) of the *Mobile* class specify that a *Mobile* will only receive messages on the terminal it has associated the variable *downlinkChannel* with, and will only send messages from the terminal with which it has associated the variable *uplinkChannel*. The *input* and *output* methods of the *BaseStation* class will only receive messages on the terminal it has associated the variable *uplinkChannel* with, and will only send messages from the terminal with which it has associated the variable *downlinkChannel*. In addition to the *input* and *output* methods, the *Mobile* class defines methods for originating calls, terminating calls, requesting a channel and receiving a channel grant. The *BaseStation* class defines methods for processing requests to page mobiles and processing requests for dedicated channels. The *Cell* class defines two methods: one which is called when a mobile enters the cell and one which is called when the mobile exits the cell.

The model defines the following six events:

1. *EnterCellEvent* - causes a new mobile object to be instantiated. The mobile is placed in the cell by executing the *Cell*'s *enterCell* method. This event schedules the next *EnterCellEvent* to occur at some random time interval.

2. *ExitCellEvent* - randomly selects a mobile in the cell and removes it by calling the *Cell*'s *exitCell* method. This event schedules the next *ExitCellEvent* to occur at some random time interval.

3. *CallOriginate* - randomly selects an idle mobile and causes it to send a channel request to the *BaseStation*. This event schedules the next *CallOriginateEvent* to occur at some random time interval.

4. *MobilePageEvent* - randomly selects a mobile to be paged, and causes the *BaseStation* to send a page message to it. This event schedules the next *MobilePageEvent* to occur at some random time interval.

5. *CallTerminateEvent* - causes a mobile to release its dedicated channel, and return to an idle state.

6. *MobileTimeOutEvent* - causes a mobile to retry a channel request if the previous request was not granted.

All random time intervals between events are exponentially distributed. The distribution means are defined as model parameters. The model also allows the user to configure the downlink channel in terms of how much bandwidth is reserved for the access grant subchannel, and how much is reserved for paging subchannel. The model outputs the average page delay (the elapsed time between the time the *BaseStation* received the page, and the time the mobile was granted a dedicated channel to respond to the page), and the average channel request delay (the elapsed time between the time a mobile requested a dedicated channel and the time it was granted a channel by the *BaseStation*).

Model Execution

Execution of the model proceeds as follows. Instantiation of a *Cell* object causes its constructor to schedule the following events: *EnterCellEvent, ExitCellEvent, CallOriginateEvent* and a *MobilePageEvent.* A new mobile object is instantiated when an *EnterCellEvent* occurs. The mobile is placed in the cell by calling the cell's *enterCell* method with the new *Mobile* object as a parameter. This method connects the mobile's downlink terminal to the base station's downlink terminal and the mobile's uplink terminal to the base station's uplink terminal. These connections are made explicitly by calling the *Cell* object's *registerConnection* method. (Recall that the *Cell* class inherits the *SubModel* class which provides the *registerConnection* method.) When an *ExitCellEvent* occurs a mobile is selected at random to be removed from the cell. The mobile is removed from the cell by calling the cell's *exitCell* method, with the selected mobile as a parameter. This method explicitly disconnects the mobile's terminals from the base station's terminals by calling the *Cell* object's *removeConnection* method.

When a *CallOriginateEvent* occurs, an idle mobile is selected at random. This mobile requests a dedicated channel by sending a channel request message to the base station from its uplink terminal. The framework uses the connectivity equations generated from the connections to route the message from the mobile's uplink terminal to the base station's uplink terminal. The base station schedules a channel grant message to be sent to the mobile during the downlink channel's next available access grant frame. The time of this frame is determined by the configuration of the downlink channel and the number of access grant messages waiting to be sent. The framework uses the connectivity equations generated from the connections, in sending the channel grant message to the mobile. Whenever a channel request is granted, the mobile changes its state to busy and schedules a *CallTerminateEvent* at some random future

time. When the *CallTerminateEvent* occurs, it sets the mobile state to idle.

When a *MobilePageEvent* occurs, a mobile is selected at random. The base station schedules a page message to be sent to the mobile during the next available page frame. The time of this frame is determined by the configuration of the downlink channel and the number of page messages waiting to be sent. If the mobile is not busy upon receipt of this message, it will send a channel request message to the base station on its uplink terminal.

6.2.3 Discussion

Even though this model is discrete and does not utilize the framework's facilities for continuous modeling, the framework's support for explicitly modeling dynamic connections and components resulted in a model implementation which was easier to develop and understand than a similar model using a more traditional modeling tool. Originally, the model was implemented using a commercial simulation tool which employs the process-interaction[1] approach to discrete event simulation. Our intention was to develop a model which could easily be extended to include new control channel algorithms as they became available. Although this objective was achieved to some degree from our point of view, to the casual user the model is a confusing clutter of nodes and edges. Figure 6.5 shows a single module of the model. The reason for the difficulty was that the modeling tool gave no support for explicitly modeling connections and components which appear and disappear over time. Since mobiles are transient entities, they were modeled as transactions. The base station was modeled as a graph where the nodes in the graph were primitive building blocks provided by the tool. The notion of connection is implicit. A mobile was considered

[1]The process-interaction approach represents a system with a static graph, upon which transactions can move from node to node. Each node has a block of executable code associated with it, that is executed each time a transaction enters the node.

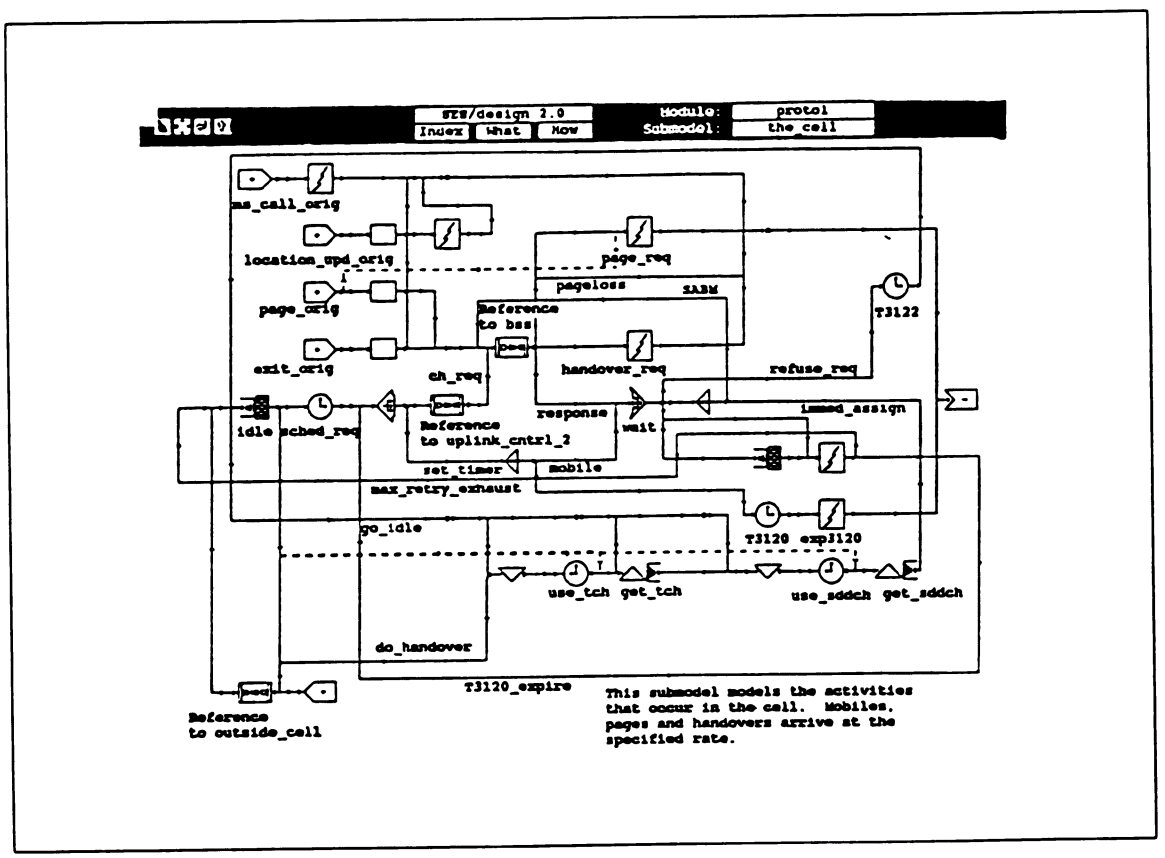

Figure 6.5. Part of the control channel model using a commercial tool.

"connected" to the base station whenever it passed over a certain edge and entered into the portion of the graph that represented the base station.

In contrast, the FMDC implementation of the model represents all the physical entities of the system with three simple class definitions: *Cell, BaseStation*, and *Mobile.* All three of these classes inherit the *Component* base class, which is provided by the framework. Basing these classes on the same abstraction is more reflective of the real world, where cells, base stations, and mobiles, are all physical entities. A modeling approach which forces the user to represent one physical entity with one mechanism (such as a graph), and another physical entity with another mechanism (such as a transaction) results in a model which is difficult to understand. In this example the appearance of new components and connections, and the disappearance of existing components and connections, are a common occurrence. By dealing with them directly, the FMDC framework represented *Cells*, *BaseStations*, and *Mobiles* in a uniform manner, resulting in a model which is more reflective of the actual system. Such a model is easier to understand, develop and extend.

6.3 An Electrical Circuit

The framework's support for explicitly modeling connections among components makes it possible to model both discrete and continuous components within a single model. A component may have terminals by which it will receive messages from other components connected to that terminal. A component may also have terminals that are used to establish mathematical relationships among the component's internal variables and the internal variables of other connected components. A component which has both types of terminals may serve as a coupling between the discrete and continuous portions of the model. A model of an electrical circuit which contains both discrete parts modeled with three level digital logic, and continuous parts modeled

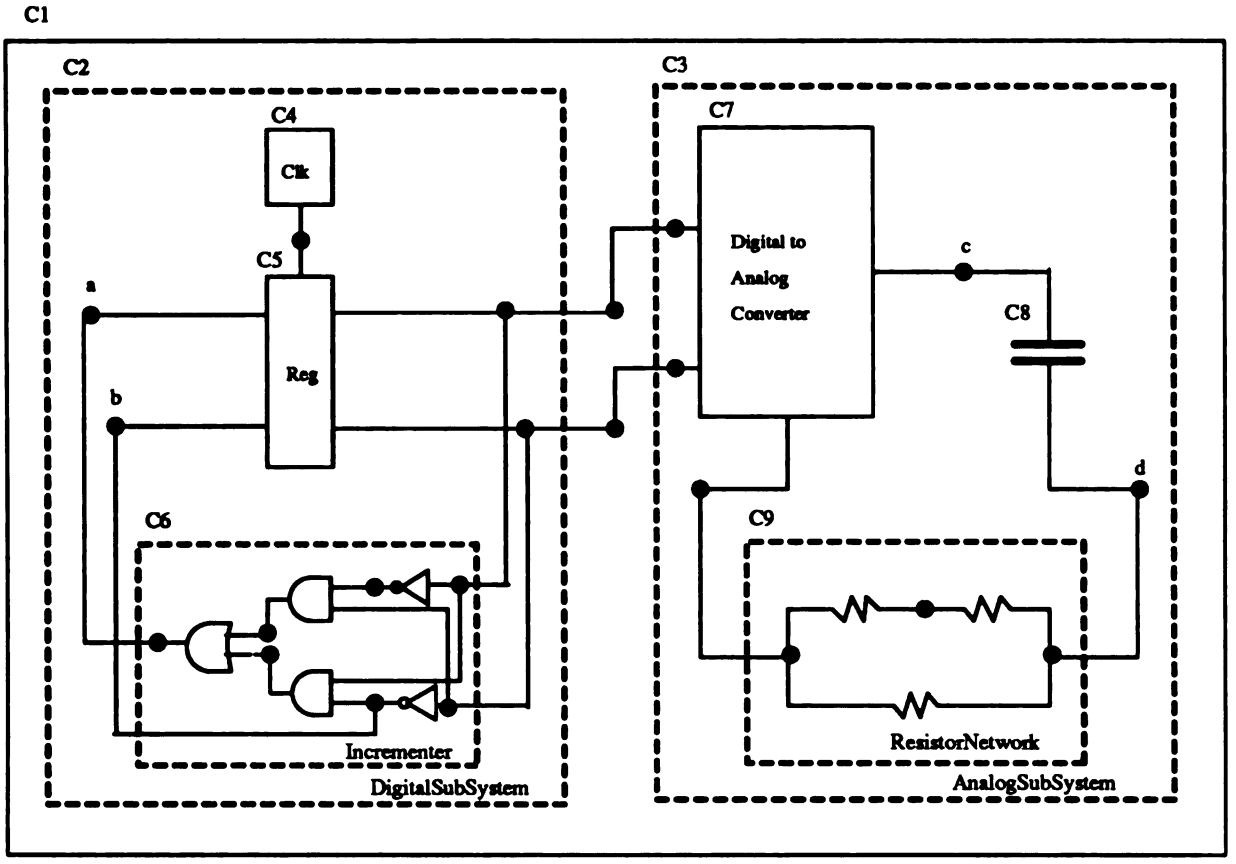

Figure 6.6. Model of an electrical circuit with discrete and continuous parts.

with algebraic and differential equations is presented in the remainder of this section.

6.3.1 A Model of an Electrical Circuit

An hierarchical model of an electrical circuit is shown in Figure 6.6. The model consists of a digital subsystem labeled C_2, and an analog subsystem labeled C_3. The digital subsystem is a state machine built with logic gates, a register, and a clock. There are a total of four states: 00, 01, 10, and 11. Component C_5 is a register that stores the machine's state on the positive edge of the clock. The combinational logic encapsulated in component C_6, computes the next machine state. The next state is calculated by incrementing the current state modulo four.

The electrical subsystem component, C_3, contains components C_7, C_8, and C_9. Component C_8 is a capacitor which is modeled by the constitutive equations:

$$\frac{dC_8.V}{dt} - \frac{C_8.I}{c} = 0.0 \tag{6.1}$$

$$C_8.V = C_8.V_a - C_8.V_b \tag{6.2}$$

$$C_8.I - C_8.I_a = 0.0 \tag{6.3}$$

$$C_8.I_a + C_8.I_b = 0.0. \tag{6.4}$$

Component C_9 consists of three resistors, each of which defines the constitutive equations (component identifiers are not shown for the variables):

$$I - I_a = 0.0 \tag{6.5}$$

$$I_a + I_b = 0.0 \tag{6.6}$$

$$V_b - V_a - \mu I = 0.0. \tag{6.7}$$

The digital-to-analog converter, component C_7, serves as a coupling component between the discrete components, and the continuous components. In addition to the constitutive equations:

$$C_7.V_a - C_7.V_b = C_7.V_{out} \tag{6.8}$$

$$C_7.I - C_7.I_a = 0 \tag{6.9}$$

$$C_7.I_a + C_7.I_b = 0, \tag{6.10}$$

C_7 also contains two terminals from which it receives digital logic signal values. Thus, the digital-to-analog component has four terminals: two digital terminals, and two electrical terminals. The component has an *input* method defined, which is used

to process incoming digital signal values. This method sets the value of $C_7.V_{out}$, the voltage drop across the two electrical terminals, according to the table given in Figure 6.7. I_1 and I_0 are the most recent signal values received on the two digital terminals.

I_1	I_0	V_{out}
0	0	0
0	1	4
1	0	8
1	1	12

Figure 6.7. Computing V_{out} for the digital-to-analog component.

The "kind of" hierarchy for the circuit model is given in Figure 6.8. The *Gate* class is an example of how reuse can be facilitated by object-oriented inheritance when developing primitive components. The *input* and *output* methods are the same for all logic gates. The *input* method evaluates the gate's response to the input signal and schedules any changes of the output signal that may arise. The *output* method propagates the signal change to any gates connected to the output terminal. Logic gates derived from the *Gate* class inherit the *input* and *output* methods, and a general *registration* method which defines its terminals and variables. A class derived from *Gate* need only define a method which implements the relevant truth table.

There are two types of terminals used in this model: the *ElectricalTerminal*, and the *DigitalTerminal*. The two partitions of the *ElectricalTerminal* use the identity and conservation equation connection rules. Each electrical component assigns a voltage variable to the first partition and a current variable to the second partition. For example, the connections within the electrical subsystem connecting the capacitor to

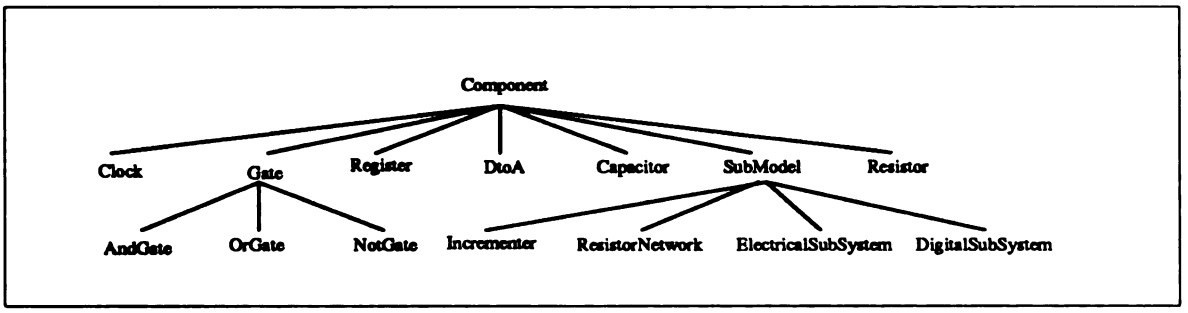

Figure 6.8. The "kind of" hierarchy for the circuit model.

the digital-to-analog converter and the resistor network (c and d in Figure 6.6), consist of *ElectricalTerminals* and result in the formulation of the following connectivity equations:

$$c : C_7.I_a + C_8.I_a = 0 \tag{6.11}$$

$$c : C_7.V_a = C_8.V_a \tag{6.12}$$

$$d : C_9.I_b + C_8.I_b = 0 \tag{6.13}$$

$$d : C_9.V_b = C_8.V_b \tag{6.14}$$

The *DigitalTerminal* consists of a single partition to which the identity equation connection rule is assigned. The connections within the digital subsystem connecting the register's inputs to the outputs of the incrementer (a and b in Figure 6.6), result in the formulation of the following connectivity equations:

$$a : C_6.out_1 = C_5.in_1 \tag{6.15}$$

$$b : C_6.out_0 = C_5.in_0 \tag{6.16}$$

The output resulting from queries for the voltage drop across the capacitor's ter-

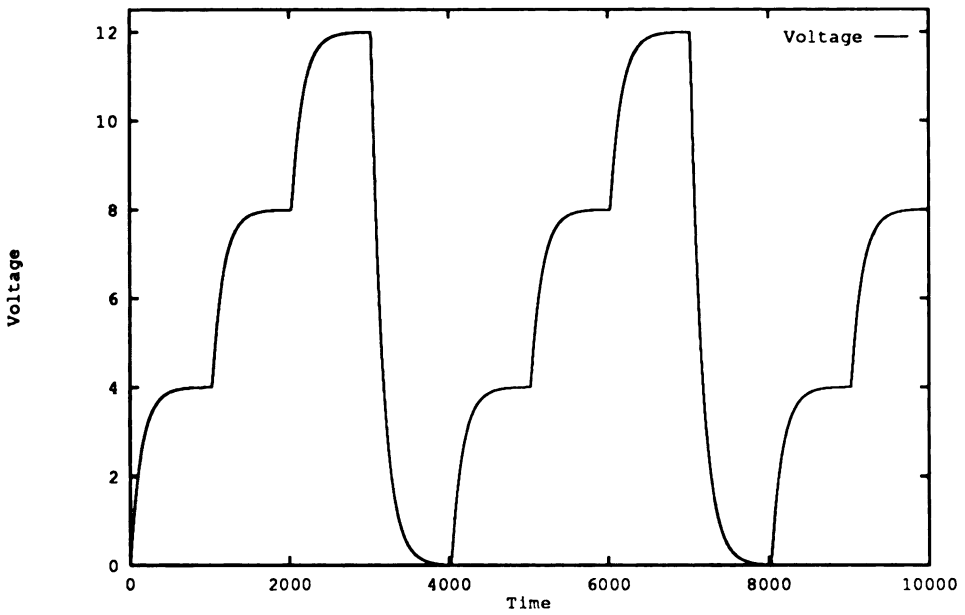

Figure 6.9. Circuit model output: the voltage across the capacitor.

minals is shown in Figure 6.9. The variations in the voltage over time correspond to the states the state machine in the digital subsystem cycles through on the positive going edge of the signal originating from the clock component.

6.3.2 Discussion

With respect to the FMDC implementation, this model consists of 17 active objects (three for each of the five *SubModels*, one for the *EventHandler* and one for the *MathSolver*). Although dynamic connections have not been discussed in this example, they can be modeled by simply calling the appropriate *SubModels*' *registerConnection* or *removeConnection* methods. The framework will automatically formulate the new connectivity equations and remove outdated equations. The new equations are organized into systems whenever a solution is demanded by a variable query. In addition to changes to the equation set, topological changes will also cause the initial

condition for the differential equation defined by the capacitor to be recomputed. The closure algorithm presented in Chapter 5 is only applied within the scope of the *SubModel* containing the topological change. For example, if another component was spawned and added within component C_9 (the resistor network), only the *EquationAccumulator* encapsulated within that *SubModel* will apply the closure algorithm. The equations cached by *EquationAccumulators* external to that *SubModel* are still valid.

6.4 Population Dynamics of a Beehive

Systems where components and connections dynamically appear and disappear over time can also be found in disciplines such as biology, ecology, economics and others. In this section, a model of the population dynamics of a domestic honey bee colony is developed using the FMDC framework.

6.4.1 Background Material

Apiculture[2] was revolutionized by the invention of the removable frame beehive by Langstroth in the middle of the 19th century [76]. Langstroth discovered that if wooden frames were hung in a beehive with a 5/16 inch space (called "bee space") between the frames, and between its borders and the walls, ceiling, and floor of the hive, the bees would build combs in the frames that could be removed and replaced leaving the hive intact. If the space is less than 5/16 of an inch, the bees will fill the space with wax and propolis[3] If the space exceeds 5/16 of an inch, the bees will fill the space with combs. The removable frame was revolutionary for two reasons. Not only did they allow surplus honey to be removed from the hive without destroying

[2]The domestic keeping of bees on a large scale.
[3]A resin like substance bees gather from a variety of plants.

the colony, but they also allowed the apiarist to closely monitor the hive for diseases, starvation, and other harmful problems.

The combs constructed within the frames are used by the bees for the rearing of brood[4] and the storage of honey and pollen. Three types of bees are in the honeybee colony: the queen, the worker, and the drone. The queen is a female bee with reproductive capacities. Each colony under normal circumstances has only one queen. The worker is a female with no reproductive capacities. The worker bee is responsible for tending the brood, building combs, and gathering nectar and pollen from plants. The gestation period for a worker bee is approximately 21 days. During nectar flow, the worker bee's life span is around six weeks. The worker spends the first three weeks of its life working in the hive, and the last three weeks gathering nectar and pollen outside the hive. The drone is the male bee and does little more than consume honey.

The worker population of a colony may range from several thousand bees, to 60,000-70,000 bees, depending on the time of the year, the prolificacy of the queen, and a number of other factors. It is important that the colony have the right population at the right time of the year. For example, during the height of the bloom, when the nectar is most plentiful, the colony needs a maximum population. At other times of the year, such as early spring, and fall, a large population of idle worker bees can be detrimental. A large worker population in the early spring will inevitably lead to swarming, a natural process in which the queen will leave the hive with a sizable proportion of the worker population, in an attempt to start another colony elsewhere. The workers and the new queen left behind rarely recover in time to harvest surplus honey from the main nectar flow.

Since the invention of the removable frame beehive, apiarists have been able to manipulate colonies in an attempt to control the population of the colony. A typical

[4]Bees still in gestation.

procedure in the spring time is to equalize the strength of the colonies in a particular apiary. The apiarist removes frames of brood from strong colonies and places them in the hives of weak colonies. This procedure strengthens the weaker colony, and at the same time may help alleviate the stronger colony's swarming tendency.

Since the wild bee population is no longer sufficient for crop pollination the agricultural industry relies heavily on domestic beekeeping for pollination. Entomologists are constantly researching new techniques for managing honeybee colonies, preventing diseases etc. Computer-based models of honeybee colonies are desirable because they allow the researcher to perform an experiment in minutes on a computer, that would take months or even years to carry out in the apiary. The computer model is also a more cost-effective way to carry out research, since some experiments may result in the destruction of the colony and other expensive equipment. Rutges and Vens have recently published results describing their use a computer-based model to study diseases in honeybee colonies [79]. The following section demonstrates how FMDC can be used to model the worker population of a honeybee colony. The apiarist's manipulations of the hive, such as the colony equalization procedure described previously, can be handled in a straightforward manner using the framework's mechanisms for modeling dynamic connections.

6.4.2 Modeling the Population Dynamics

The model will simulate the dynamics of the worker population for 130 days, starting in early spring (mid April) and ending in early fall (late September). Because there are many bees within one colony, individual bees are not modeled. Instead, an aggregate model which approximates the behavior of the colony will be developed. On the 30th day (mid May), we will assume the apiarist places four frames of brood in the colony. It is assumed that the brood contained in these frames are in the latter stage of gestation and will emerge as adult worker bees with a negligible delay. The

components and connections of the model in its initial configuration are shown in Figure 6.10.

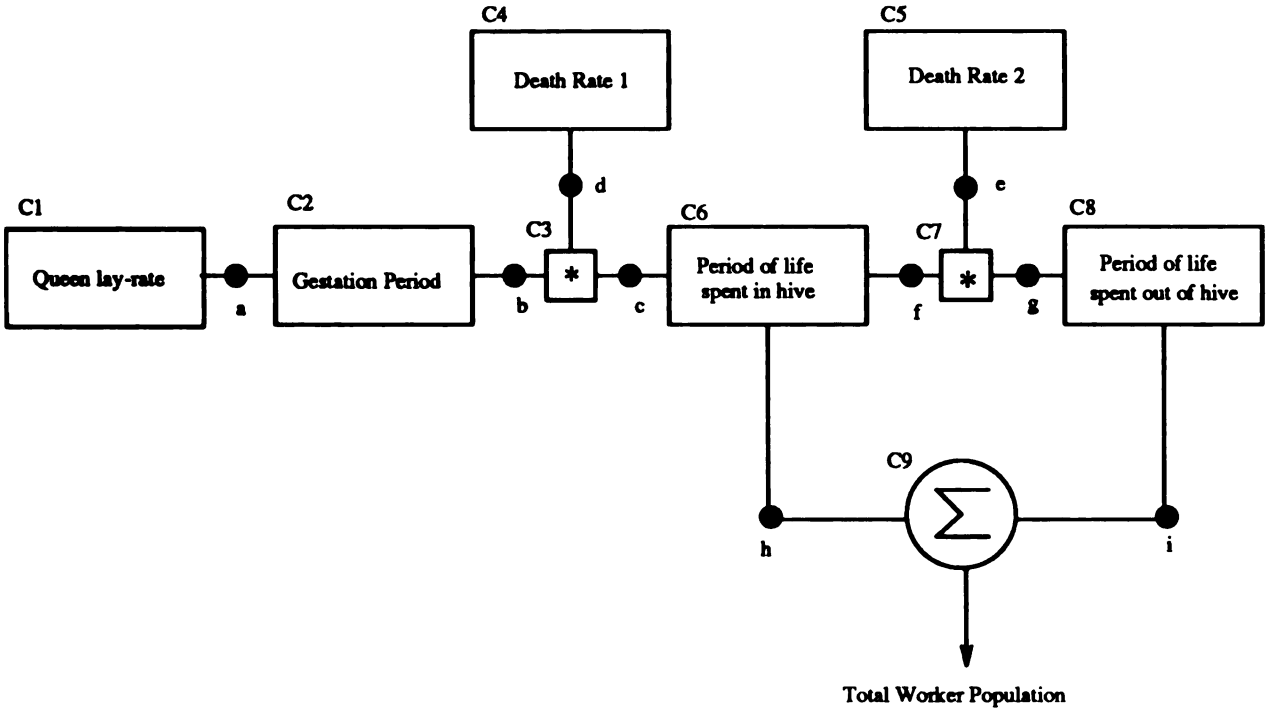

Figure 6.10. Model of worker bee population.

Component C_1 models the rate at which the queen lays eggs. It is assumed that the queen will lay at a peak rate of 2,000 eggs per day during the early spring, anticipating the main nectar flow. The lay rate will decrease as the summer progresses. The lay rate will be approximated with the single constitutive equation:

$$C_1.out(t) = layRate(t) \tag{6.17}$$

where $layRate(t)$ is modeled by the step function:

$$layRate(t) = 2000u(t) - 300u(t-25) - 200u(t-50) - 400u(t-75) - 300u(t-100) - 400u(t-125)$$

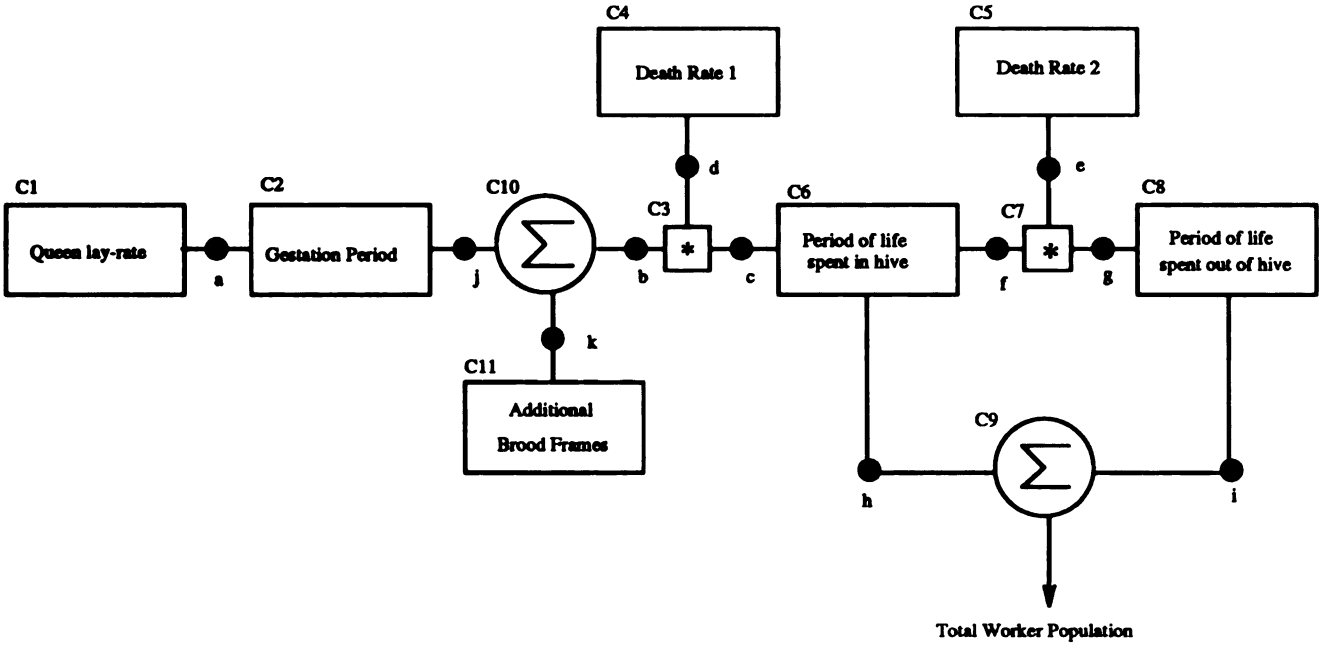

Figure 6.11. Model of worker bee population with additional brood frames being inserted.

where the unit step $u(t - t_0)$ is defined as

$$u(t - t_0) = \begin{cases} 1 & \text{if } t \geq t_0 \\ 0 & \text{if } t < t_0 \end{cases}$$

In reality, the lay rate is a function of the age of the queen, the availability of nectar and pollen, the temperature, and many other variables.

Component C_2 models the gestation period of the worker bees. This delay is modeled by a second order differential equation which is represented by two constitutive equations:

$$C_2.y_1'(t) = \frac{2}{del}(C_2.in(t) - C_2.y_1(t)) \tag{6.18}$$

$$C_2.out'(t) = \frac{2}{del}(C_2.y_1(t) - C_2.out(t)) \tag{6.19}$$

where *del* is the gestation delay in days. This type of component is referred to as a

distributed delay and is commonly used to model delays in aggregate models. Since the variance in gestation delay is quite small, a higher order equation would be more realistic, but the second order delay suffices for demonstration purposes.

Components C_6 and C_8 model the life times of the adult worker bees. Component C_6 models the period of time the worker bee spends carrying out its responsibilities within the hive, while component C_8 models the period of time the worker bee spends foraging for nectar and pollen. The constitutive equations are:

$$C_6.out'(t) = \frac{1}{d_{in}}(C_6.in(t) - C_6.out(t)) \tag{6.20}$$

$$C_8.out'(t) = \frac{1}{d_{out}}(C_8.in(t) - C_8.out(t)) \tag{6.21}$$

where d_{in} is the time in days the worker bee typically spends in the hive, and d_{out} is the time in days the worker bee spends foraging outside the hive. Since we are interested in the total adult worker population in the colony, the total number of bees in each of these two phases must be computed. The population in each stage can be computed with the equations:

$$C_6.pop(t) = d_{in}C_6.out(t) \tag{6.22}$$

$$C_8.pop(t) = d_{out}C_8.out(t). \tag{6.23}$$

Components C_4 and C_5 introduce the death rates of worker bees. The constitutive equations for these components are:

$$C_4.out(t) = dr_1(t) \tag{6.24}$$

$$C_5.out(t) = dr_2(t) \tag{6.25}$$

where $dr_1(t)$ and $dr_2(t)$ are step functions.

Components C_3 and C_7 are two input multipliers and have the constitutive equations:

$$C_3.out(t) = C_3.in_1(t) * C_3.in_2(t) \tag{6.26}$$

$$C_7.out(t) = C_7.in_1(t) * C_7.in_2(t). \tag{6.27}$$

The total worker population is computed by adding the number of adult bees in each of the two stages. This is accomplished by Component C_9 whose single constitutive equation is:

$$C_9.out(t) = C_9.in_1(t) + C_9.in_2(t). \tag{6.28}$$

The connectivity equations generated by the simulation framework from the nine connections labeled $a - i$ in Figure 6.10 are:

$$a : C_1.out(t) = C_2.in(t) \tag{6.29}$$

$$b : C_2.out(t) = C_3.in_1(t) \tag{6.30}$$

$$c : C_3.out(t) = C_6.in(t) \tag{6.31}$$

$$d : C_4.out(t) = C_3.in_2(t) \tag{6.32}$$

$$e : C_5.out(t) = C_7.in_2(t) \tag{6.33}$$

$$f : C_6.out(t) = C_7.in_1(t) \tag{6.34}$$

$$g : C_7.out(t) = C_8.in_1(t) \tag{6.35}$$

$$h : C_6.pop(t) = C_9.in_1(t) \tag{6.36}$$

$$i : C_7.pop(t) = C_9.in_2(t). \tag{6.37}$$

The insertion of brood frames into the hive can be modeled as a dynamic connection. At time $t = 30$ two additional components are spawned into the model, C_{10} and C_{11}, with constitutive equations:

$$C_{10}.out(t) = C_{10}.in_1(t) + C_{10}.in_2(t) \tag{6.38}$$

$$C_{11}.out(t) = numBees \tag{6.39}$$

where $numBees$ is a constant denoting the number of adult bees which will emerge from the brood frames being inserted into the hive. The connection b in Figure 6.10 is removed, and replaced with three new connections b, j, and k, as shown in Figure 6.11. The connectivity equations generated are:

$$b : C_{10}.out(t) = C_3.in_1(t) \tag{6.40}$$

$$j : C_2.out(t) = C_{10}.in_1(t) \tag{6.41}$$

$$k : C_{11}.out(t) = C_{10}.in_2(t). \tag{6.42}$$

At time $t = 31$ the connections b, j, and k, and components C_{10} and C_{11} can be removed and component C_2 can be reconnected to component C_3, restoring the model to its original topology as shown in Figure 6.10.

The framework accumulates the constitutive equations and the connectivity equations formulated from the connections, and solves them when variable queries are made. Figure 6.12 shows the values computed for the variable $C_9.out(t)$, the total worker population, for two different executions of the model: one where the hive was left undisturbed over the entire 130 day period, and another where brood frames were inserted into the hive on the 30th day. The solution of the equations was driven by

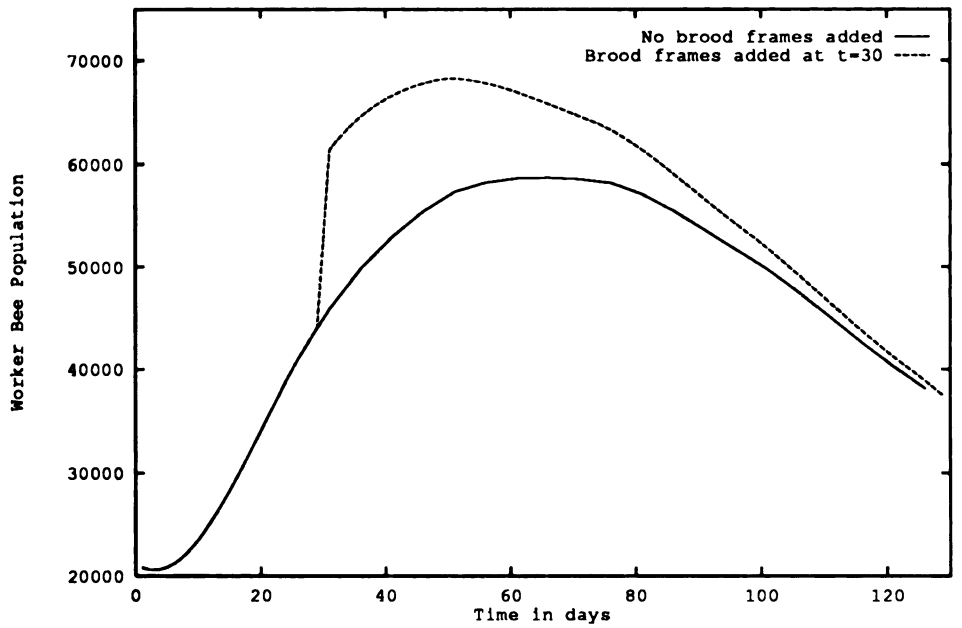

Figure 6.12. Output from the Beehive model.

the queries for the variable $C_9.out(t)$.

6.4.3 Discussion

Although many simplifying assumptions were made in this model, the results shown in Figure 6.12 are reasonable and correspond with the discussions of the honeybee population dynamics found elsewhere [76]. However, the objective in developing this model was not to produce new results pertaining to the population dynamics of honeybee colonies. The objective was to demonstrate the framework's applicability in a problem area that is not normally studied within the engineering disciplines.

Other phenomena in the beehive colony could be modeled with the framework. For example, if the queen were replaced with a more prolific queen (a common procedure), this could be modeled by removing component C_1 and replacing it with a component that accurately represented the new queen's laying rate. More complex phenomena

such as the effects of pesticides, marauding predators, etc., can be represented by dynamically spawning new components and interconnecting them with the existing components. Existing simulation tools can be used to model these topological changes by building a separate model for each possible topological configuration. For example, in the beehive model, two models would be constructed: one describing the topology before the addition of brood frames, and one describing the topology after the brood frames are inserted. The first model is executed until day 30. The output of this model is used as initial conditions in the second model. The second model is then executed from day 30 to day 130. While this approach may be reasonable for this particular example, in general, it is not practical for a number of reasons. First, if the topology is expected to change frequently, many different models would have to be constructed. Second, this approach is not applicable to interactive simulation where new components and connections can be introduced spontaneously, at the whim of the user. Finally, this approach assumes that the model developer somehow knows *a priori* all the possible topological configurations of the model. The FMDC framework can model dynamic components and connections within a single model, and makes no assumptions about the different topological configurations the model might evolve into.

6.5 Summary

This chapter has presented an implementation of the simulation framework called FMDC. Models from several problem domains were presented. A model of the common control channels in a cellular phone system demonstrated the framework's discrete modeling capabilities. A model of an electrical circuit demonstrated the framework's capability to mix both continuous and discrete components in one model. An aggregate model of the population dynamics of a beehive demonstrated the frame-

work's applicability to systems outside the realm of engineering.

CHAPTER 7

Summary, Contributions and Recommendations

7.1 Summary

Traditional computer-based simulation and modeling methodologies do not provide support for explicit modeling of components whose existence and connections vary over time. Many systems exist that contain components and connections which vary over time. Visual and interactive modeling tools also give rise to models whose components and connections vary over time by allowing users to interactively manipulate the model. This dissertation has presented an object-oriented framework for modeling systems whose components and connections vary with time.

The framework consists of four layers. The *applications layer* is the layer in which the framework is instantiated. This is accomplished by deriving classes from the abstract classes defined by the framework. The *interface* layer serves as an interface between the applications layer and the simulation layer, and consists of the abstract classes defined by the framework. The *simulation* layer consists of a set of concurrent objects which respond to changes in a model's component population and to the connections among components. The *computational* layer is responsible for the solution

of the systems of equations formulated by components and their connections.

Precise definitions were developed for components, terminals, connections, submodels, and other key abstractions. The *SubModel* abstraction is used to construct hierarchical models. The *SubModel* abstraction also solves the problem of mapping messages from the application to the appropriate concurrent object residing in the simulation layer. The concurrent objects in the simulation layer are transparent to the application layer.

The framework provides efficient mechanisms for handling the dynamic set of representative equations. Equation solution is demand driven; equations are not solved until a variable query from the application necessitates solution. Equations are organized and cached within the context of the *SubModel* with which they were registered. When a topological change occurs within the context of a particular *SubModel*, changes to the equation set are isolated to the subset of equations maintained by that *SubModel*.

7.2 Contributions

The primary contributions of this dissertation are

- **a modeling methodology that provides explicit support for modeling systems whose component population and interconnections may vary with time.** Unlike existing modeling methodologies that either assume a static population of components and interconnections or provide a very limited notion of a connection, the approach proposed by this dissertation allows a model's component population and interconnections to vary with time.

- **the application of object-oriented frameworks and object-oriented concurrency in the context of modeling systems with dynamic connections.** The framework defines a set of abstractions that can be used to

develop a family of simulation applications. The concurrent object-oriented approach allowed us to develop a framework that is amenable to a parallel or distributed implementation. Because the concurrent objects are transparent to the simulation application, applications derived from a parallel implementation of the framework are executed in parallel without the application's developer providing for it explicitly.

- **the unified treatment of components and connections in the context of continuous and discrete event simulation.** The same framework abstractions are used to develop both continuous and discrete components.

- **a C++ implementation of the framework.** The implementation has demonstrated the practicality of the ideas presented in this dissertation and can be used as a base for further research.

7.3 Recommendations

There are several areas in which this research could be carried further. First, an underlying assumption during the course of this research was that the application knows when new connections and components occur. If the application is an interactive simulator with which a user is interactively adding or removing components and connections from the running model, this assumption poses no problem. The user manipulations can be mapped directly onto the framework's component and connection registration methods. However, in an application where connections occur when certain conditions placed on component variables are satisfied, rather than when explicitly stated, a monitoring mechanism is needed. The mechanism must allow conditions such as "if $\vec{p_1} = \vec{p_2}$ then connect components X and Y" to be registered, where $\vec{p_1}$ and $\vec{p_2}$ might be position vectors defined and registered by components X

and Y. More general conditions, such as "if the position vectors of any components currently residing in region R are equal then connect those components" might also be registered. Efficient and accurate monitoring of such conditions in the context of large hierarchical models consisting of thousands of components is not a trivial problem.

Another area that needs further investigation is the application of results from parallel and distributed simulation to the simulation framework presented in this dissertation. In a parallel implementation, the concurrent objects need to be synchronized. Both the optimistic and conservative approaches are applicable here. Questions regarding which approach is best for this particular framework, and under what conditions remain to be answered.

Finally, from our experiences in implementing the computational layer of FMDC, we learned that more attention needs to be given to automatically manipulating mathematical equations into a form that can be readily solved with a computer. In the past simulation tools have forced the model developer to express equations in a form that lent itself to the underlying solution method. Robust implementations of the framework proposed here should allow equations to be registered in a form that the user finds acceptable.

APPENDICES

APPENDIX A

The Framework Classes

This appendix gives a detailed description of the data members and interface protocol for the abstract classes defined by the framework. An asterisk following a type name indicates that the data member is a pointer of that type.

A.1 The *Component* Class

The *Component* class is used to construct primitive components. The member data and member functions of this class are summarized in Figure A.1 and Figure A.2.

Name	Type
componentId	integer
terminals	Array of Terminal*
numberOfTerminals	integer
context	SubModel*

Figure A.1. Data members of the abstract class *Component*

Instantiating a component causes a unique number to be assigned to *componentId*. It is used by the framework for identification purposes. The member, *terminals*, is

Return value	Method name	Parameters
none	Component	SubModel*, integer
integer	registration	none
none	beforeConnecting	none
none	beforeUnConnecting	none
none	input	integer, Message
none	output	integer, Message

Figure A.2. The interface protocol of the abstract class *Component*

a dynamic array of pointers to the terminals of a component. The member, *numberOfTerminals*, indicates how many terminals are currently associated with the component. The *context* pointer is used to indicate where the component exists in the model hierarchy.

An instance of class *Component* is constructed via the component method *Component*. Its two parameters specify the context of the component, and how many terminals it will have. The methods *beforeConnecting* and *beforeUnConnecting* contain code to be executed before topological changes occur. The *input* and *output* methods are used for propagating messages among components. None of these methods, with the exception of the *Component* method, contain code in the *Component* class. They are defined in the *Component* class in order to establish the interface protocol among classes that will be derived from it

A.2 The *Terminal* Class

Instances of the *Terminal* class are used by a component to indicate the potential to connect to other components which have terminals of the same type. The member data and member functions of this class are summarized in Figure A.3 and Figure A.4.

The member *connectionId* is a unique identifier generated by the *ConnectionManager*

Name	Type
connectionId	integer
internalConnectionId	integer
partitionTable	2-dimensional array of integer
connectRule	Array of ConnectionRule*
parentContext	SubModel*
currentContext	SubModel*

Figure A.3. Data members of the abstract class *Terminal*

Return value	Method name	Parameters
none	Terminal	SubModel*, SubModel*
none	assignVariable	integer, integer
none	assignPartition	ConnectionRule*, integer

Figure A.4. The interface protocol of the abstract class *Terminal*

when the terminal is connected to other terminals. All the terminals in one connection will have the same *connectionId.* A terminal of a *SubModel* nested within another *SubModel* may have a reference to an internal connection. The unique identifier for this connection is stored in the member *internalConnectionId.* References to the applicable connection rules are stored in the array *connectRule.* The size of the array is equal to the number of variable partitions defined by the terminal. The *partitionTable* maintains a set of variable identifiers for each partition. Every variable is given a unique identifier by the *EquationAccumulator* when it is registered. The two *SubModel* pointers are used by terminals of nested *SubModels.* The *parentContext* member points to the parent *SubModel,* and the *currentContext* member points to the *SubModel* that instantiated the terminal.

An instance of class *Terminal* is constructed via the *Terminal* method. The two

SubModel pointers received as input parameters are stored in the *parentContext* and *currentContext* members. The *assignVariable* method is used to assign variables to the terminal's variable partitions. The first integer input parameter is the partition identifier, and the second integer input parameter is a variable identifier. The *assignPartition* method assigns a connection rule to a variable partition. The first input parameter is a pointer to the connection rule, and the second parameter is the partition identifier.

The constructors of classes derived from the *Terminal* class will use the *assignVariable* and *assignPartition* methods of the base class to set up the variable partition table as desired.

A.3 The *SubModel* Class

The *SubModel* class is used to construct hierarchical models. The member data and member functions of this class are summarized in Figure A.5 and Figure A.6. Each

Name	Type
accumulator	EquationAccumulator
conman	ConnectionManager
formulator	EquationFormulator
actions	EventHandler (shared by all instances)
solver	MathSolver (shared by all instances)

Figure A.5. Data members of the abstract class *SubModel*

SubModel instance will instantiate three active objects: the *accumulator*, the *conman*, and the *formulator*. The members, *actions* and *solver*, are references to the global *EventHandler* and *MathSolver* objects.

Return value	Method name	Parameters
none	SubModel	SubModel*, integer
none	schedule	Event*
integer	registerEquation	Equation*
integer	registerVariable	Variable*
integer	queryVariable	integer, integer
integer	rememberVariable	integer integer
integer	revokeConstitutiveEquation	integer
integer	registerComponent	Component*
integer	registerConnection	integer,integer,integer,integer
integer	removeComponent	integer
integer	removeConnection	integer, integer
none	propagateOutput	Variable*, Message
none	defineConstant	String, integer, real
none	undefineConstant	String, integer

Figure A.6. The interface protocol of the abstract class *SubModel*

The *SubModel* class supports a number of member functions which are used to manipulate the connections and components within the *SubModel.* These methods are as follows:

1. *SubModel* - used to create new *SubModel* instances. The *SubModel** parameter is a reference to the *SubModel* that the new *SubModel* instance will be registered with.

2. *schedule* - used to place an event in the priority queue maintained by the *EventHandler* object.

3. *registerEquation* - used by a component to define a constitutive equation. The method returns the unique identifier assigned to the equation.

4. *registerVariable* - used by a component to define a variable. The method returns the unique variable identifier assigned to the variable.

5. *queryVariable* - used to query for the value of a variable that has been registered previously. The input parameters are the variable identifier and the evaluation time.

6. *rememberVariable* - similar to the *queryVariable* method, except the result of the query is stored internally. This method is called to compute and store initial conditions for differential equations during topological changes.

7. *revokeConstitutiveEquation* - sends a message to the *EquationAccumulator* instructing it to revoke a constitutive equation. The equation is identified by the integer parameter.

8. *registerComponent* - send the *ConnectionManager* a message instructing it to register a new component.

9. *registerConnection* - send the *ConnectionManager* a message instructing it to register a connection. The four integer parameters identify the two components and their terminals to be connected.

10. *removeComponent* - sends the *ConnectionManager* a message instructing it to remove a component. The component is identified by the integer parameter.

11. *removeConnection* - sends the *ConnectionManager* a message instructing it to remove a component from a connection. The two integer parameters identify the component and terminal to be disconnected.

12. *propagateOutput* - used by discrete components to propagate an output message. The variable parameter indicates which terminal the message is to be propagated from.

13. *defineConstant* - initializes a symbolic constant with the *MathSolver*.

14. *undefineConstant* - removes a previously initialized symbolic constant.

A.4 Event Classes

Events are state changes which occur at discrete points in time. The framework defines an abstract class, *Event*, whose protocol and data members are given in Figures A.7 and A.8. The data member *submodel* indicates the context in which the event should be executed, and the member *time* indicates the time the event is to be executed. The member function, *eval*, is executed at the scheduled time, *time*. It returns an array of pointers to events which were generated during the execution of *eval*. The framework defines a set of standard events which are listed in Figure A.9. Each of these Events is inherited from the abstract class *Event*. New events can be derived from these events, or from *Event*.

Name	Type
submodel	SubModel*
time	real

Figure A.7. Data members of the abstract class *Event*

Return value	Method name	Parameters
Array of Event*	eval	none

Figure A.8. The interface protocol of the abstract class *Event*

Event	Description
CreateComponent	add a new component to the model
DeleteComponent	remove a component from the model
ConnectEvent	establish a connection between two or more components
UnConnectEvent	remove a connection between two components
SQueryEvent	synchronized query of a variable value
SRptQueryEvent	synchronized repetitive query of a variable value
PrintStateEvent	prints SubModel state for debug purposes
InputEvent	cause a component to read from an input terminal
OutputEvent	cause a component to output to an output terminal
TerminateEvent	cause simulation to terminate

Figure A.9. Events defined by the framework

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. Abell. *New Developments in perturbation analysis of discrete event systems with structural modifications.* PhD thesis, Oakland University, 1992.

[2] J. Abell and R. Judd. Object-oriented perturbation analysis of discrete event systems. In *Proceedings of the 1992 Summer Computer Simulation Conference*, pages 164–168, July 1993.

[3] A. Aggarwal, K. Gordon, et al. Animating simulations in RESQME. In *Proceedings of the 1989 Winter Simulation Conference*, pages 612–619, 1989.

[4] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, Sept. 1990.

[5] M. Bach. *The Design of the UNIX Operating System*, pages 383–388. Software Series. Prentice Hall, 1986.

[6] J. Banks and J. Carson. *Discrete-Event System Simulation.* Prentice-Hall Inc., 1984.

[7] C. Basnet, P. Farrington, et al. Experiences in developing an object-oriented modeling environment for manufacturing systems. In *Proceedings of the 1990 Winter Simulation Conference*, pages 477–481, 1990.

[8] E. Bensley, V. Giddings, J. Leivent, and R. Watro. A performance-based comparison of object-oriented simulation tools. In *Proceedings of Object-Oriented Simulation Conference, 1992 Western Simulation Multiconference*, pages 47–51, January 1992.

[9] J. Bezivin. Some experiments in object-oriented simulation. In *Proceedings 1987 OOPSLA*, pages 394–405, 1987.

[10] J. Bishop and O. Balci. General purpose visual simulation system: a functional description. In *Proceedings of the 1990 Winter Simulation Conference*, pages 504–512, 1990.

[11] E. Blair. DISC++: a C++ based library for object-oriented simulation. In *Proceedings of the 1989 Winter Simulation Conference*, pages 301–307, 1989.

[12] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings, 1991.

[13] D. Breen and V. Kühn. Message-based object-oriented interaction modeling. In *Eurographics'89*, pages 489–503. North-Holland, 1989.

[14] D. Brunner and J. Henriksen. A general purpose animator. In *Proceedings of the 1989 Winter Simulation Conference*, pages 155–163, 1989.

[15] O. Bryan. MODSIM II - an object-oriented simulation language for sequential and parallel processors. In *Proceedings of the 1989 Winter Simulation Conference*, pages 172–177, 1989.

[16] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A mixed-paradigm simulation/prototyping platform in C++ . In *1991 C++ at Work Conference*, 1991.

[17] J. Buck, S. Ha, E. Lee, and D. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *to appear in International Journal of Computer Simulation, special issue on Simulation Software Development*, 1993.

[18] R. Campbell, N. Islam, R. Johnson, P. Kougiouris, and M. P. Choices, *Frameworks and Refinement*, pages 9–15. IEEE Computer Society Press, Oct. 1991.

[19] F. Cellier. *Continuous System Modeling.* Springer-Verlag, 1991.

[20] F. Cellier and H. Elmqvist. The need for automated formula manipulation in object-oriented continuous-system modeling. In *Proceedings of CACSD92 - IEEE Computer-aided Control System Design Conference*, Mar. 1992.

[21] F. E. Cellier. Bond graphs – the right choice for educating students in modeling continuous-time physical systems. In *Proceedings of SCS Western Simulation MultiConference*, pages 123–127, Jan. 1992.

[22] F. E. Cellier, B. P. Zeigler, and A. H. Cutler. Object-oriented modeling: Tools and techniques for capturing properties of physical systems in computer code. In *Proceedings of CADCS91 - IFAC Symposium on Computer-aided Design in Control Systems*, pages 1–10, Jan. 1991.

[23] K. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering*, 5(9):440–452, Sept. 1979.

[24] K. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Communications of the ACM*, 24(4):198–205, 1981.

[25] G. Cooper and D. Zhao. OGST: an object-oriented graphical simulation environment. In *Proceedings of the Object-Oriented Simulation Conference, 1992 Western Simulation Multiconference*, pages 20–24, 1992.

[26] P. Corey and J. Clymer. Discrete event simulation of object movement and interactions. *Simulation*, 56(3):167–175, Mar. 1991.

[27] A. Corradi and L. Leonardi. PO: An object model to express parallelism. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 152–155, Apr. 1989.

[28] J. Cremer. *An Architecture for General Purpose Physical Simulation - Integrating Geometry, Dynamics, and Control.* PhD thesis, Cornell University, 1989.

[29] O. Dahl, B. Myhrhaug, and K. Nygaard. *SIMULA-67: Common Base Language.* Norwegian Computing Center, Oslo, Norway, 1982.

[30] W. Dally and A. Chien. Object-Oriented concurrent programming in CST. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 28–31, Apr. 1989.

[31] J. Derrick, O. Balci, and R. Nance. A comparison of selected conceptual frameworks for simulation modeling. In *Proceedings of the 1989 Winter Simulation Conference*, pages 711–718, 1989.

[32] J. Engelsma. A preliminary evaluation of the PRISM tool. Technical Report BC569-91-09, Cellular Infrastructure, Motorola Inc., Arlington Hghts, IL, 1992.

[33] J. Engelsma, M. Chung, and Y. Chung. Distributed token-driven logic simulation on a shared-memory multiprocessor. In *Proceedings of 6th Workshop in Parallel and Distributed Simulation*, pages 197–198, Jan. 1992.

[34] J. Engelsma and R. Reid. Modeling Dynamic Connections. In *Proceedings of the Summer Computer Simulation Conference*, pages 264–268, July 1992.

[35] J. Engelsma and R. Reid. A concurrent object-oriented framework for modeling dynamic connectivity. In *Proceedings of the 1993 Object-Oriented Simulation Conference, part of the Western Simulation Multiconference*, pages 143–148, Jan. 1993.

[36] J. Engelsma and R. Reid. Modeling dynamic connectivity with a hierarchy of co-operating concurrent objects. In *To appear in the 1993 European Simulation Multiconference*, 1993.

[37] J. Engelsma and P. Reilly. An extensible model of the GSM radio control channels. In *Proceedings of the Summer Computer Simulation Conference*, pages 496–500, July 1992.

[38] P. Ferrel and R. Meyer. VAMP: The Aldus Application Framework. In *Proceedings 1989 OOPSLA*, pages 185–189, 1989.

[39] R. Fujimoto. Parallel discrete event simulation. In *Proceedings of the 1989 Winter Simulation Conference*, pages 19–28, 1989.

[40] R. Fujimoto. Optimistic approaches to parallel discrete-event simulation. *Transactions of the Society for Computer Simulation*, 7(2):153–191, June 1990.

[41] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, Oct. 1990.

[42] B. Gilmore. *The Simulation of Mechanical Systems with a Changing Topology.* PhD thesis, Purdue University, 1986.

[43] R. Gimarc. Distributed simulation using hierarchical rollback. In *Proceedings of the 1989 Winter Simulation Conference*, pages 621–629, 1989.

[44] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[45] R. Gordon, E. MacNair, et al. Hierarchical modeling in a graphical simulation system. In *Proceedings of the 1990 Winter Simulation Conference*, pages 498–503, 1990.

[46] A. Guasch and R. Huntsinger. Object-oriented continuous system simulation. In *Proceedings of the 1989 Summer Simulation Conference*, pages 562–565, 1989.

[47] P. Jain and P. Newton. Putting structure into modeling. In *Proceedings of the 1989 Summer Simulation Conference*, pages 49–53, 1989.

[48] R. Jain. *The Art of Computer Systems Performance Analysis.* John Wiley and Sons, Inc., 1991.

[49] D. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7(3):404–425, July 1985.

[50] R. Johnson and V. Russo. Reusing Object-Oriented Designs. Technical Report UIUCDCS 91-1696, University of Illinois, Urbana, IL 61802, 1991.

[51] D. Jordan. Implementation benefits of C++ language mechanisms. *Communications of the ACM*, 33(9):61–64, Sept. 1990.

[52] J. Kearney, S. Hansen, and J. Cremer. Programming mechanical simulations. In *2nd Eurographics Workshop on Animation and Simulation*, pages 223–242, 1991.

[53] B. Kluth and C. Görg. Efficient simulation of network models in C++ . In *Proceedings of the 13th International Teletraffic Congress*, pages 601–608, 1991.

[54] T. Korson and J. McGregor. Understanding objected-oriented: a unifying paradigm. *Communications of the ACM*, 33(9):40–60, Sept. 1990.

[55] B. Krämer. Specifying concurrent objects. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 162–164, Apr. 1989.

[56] V. Kühn and W. Müller. Advanced object-oriented methods and concepts for simulations of multi-body systems. In *2nd Eurographics Workshop on Animation and Simulation*, pages 129–161, 1991.

[57] J. Lim and R. Johnson. The heart of object-oriented concurrent programming. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 165–167, Apr. 1989.

[58] M. Linton, J. Vlissides, and C. P.R. Composing User Interfaces with InterViews. *IEEE Computer*, 22(2):8–22, Feb. 1989.

[59] G. Lomow and D. Baezner. A tutorial introduction to object-oriented simulation and Sim++. In *Proceedings of the 1989 Winter Simulation Conference*, pages 140–146, 1989.

[60] B. Lubachevsky, A. Weiss, and A. Shwartz. An analysis of rollback-based simulation. *ACM Transactions on Modeling and Computer Simulation*, 1(2):154–193, Apr. 1991.

[61] O. Madsen and B. Moller-Pedersen. What object-oriented programming may be and what it does not have to be. In *Proceedings of the 1988 European Conference on Object-Oriented Programming*, pages 1–20, 1988.

[62] S. Mathewson. Simulation modelling support via network based concepts. In *Proceedings of the 1990 Winter Simulation Conference*, pages 459–467, 1990.

[63] K. Matwiczak. Interactive simulation: let the user beware! In *Proceedings of the 1990 Winter Simulation Conference*, pages 453–456, 1990.

[64] J. McAffer. Actor-based simulation. In *Proceedings of the 1989 Summer Simulation Conference*, pages 910–915, 1989.

[65] N. Meyrowitz. Intermedia: The architecture and construction of an object-oriented hypermedia system and applications framework. In *Proceedings 1986 OOPSLA*, pages 196–201, 1986.

[66] D. Monarchi and G. Puhr. A research typology for object-oriented analysis and design. *Communications of the ACM*, 35(9):35–47, June 1992.

[67] D. Newton and P. Vaughan. MCC PRISM version 1.4 research prototype release user's guide. Technical Report CAD-034-90, Microelectronics and Computer Technology Corporation, 1992.

[68] D. Newton, P. Vaughan, and R. Johns. PRISM: an object-oriented system modeling environment with an embedded symbolic spreadsheet. In *Proceedings of the 1991 Summer Computer Simulation Conference*, pages 81–86, 1991.

[69] J. O'Reilly and K. Nordlund. Introduction to SLAM II and SLAMSYSTEM. In *Proceedings of the 1989 Winter Simulation Conference*, pages 178–183, 1989.

[70] M. Ozden. Graphical programming of simulation models in an object-oriented environment. *Simulation*, 56(2):104–116, Feb. 1991.

[71] T. Page, S. Berson, W. Cheng, and R. Muntz. An object-oriented modeling environment. In *Proceedings 1989 OOPSLA*, pages 287–296, 1989.

[72] J. Pino, S. Ha, E. Lee, and J. Buck. Software synthesis for DSP using Ptolemy. *Invited paper to appear in the Journal on VLSI Signal Processing*, 1992.

[73] J. Poorte and D. Davis. Computer animation with cinema. In *Proceedings of the 1989 Winter Simulation Conference*, pages 147–154, 1989.

[74] A. A. B. Pritsker. Keynote address: why simulation works. In *Proceedings of the 1989 Winter Simulation Conference*, pages 1–6, 1989.

[75] R. Reid. Computer-aided engineering for computer architecture laboratories. *IEEE Transactions on Education*, 34(1):56–61, 1991.

[76] A. Root. *The ABC and XYZ of Bee Culture.* A.I. Root Company, 1983.

[77] P. Roth. Discrete, continuous and combined simulation. In *Proceedings of the 1987 Winter Simulation Conference*, pages 25–29, 1987.

[78] J. Rothenberg. Object-oriented simulation: Where do we go from here? In *Proceedings of the 1986 Winter Simulation Conference*, pages 464–469, 1986.

[79] A. Rutges and R. Vens. Object-oriented simulation of ecological systems using the beehive simulator. In *Proceedings of the 1993 Object-Oriented Simulation Conference, part of the Western Simulation Multiconferencer*, pages 157–162, Jan. 1993.

[80] M. Sakkinen. On the darker side of C++ . In *Proceedings of the 1988 European Conference on Object-Oriented Programming*, pages 162–176, 1988.

[81] L. Schruben and E. Yucesan. Simulation graph duality: a world view transformation for simple queuing models. In *Proceedings of the 1989 Winter Simulation Conference*, pages 738–745, 1989.

[82] Scientific and Engineering Software Inc., Austin, Texas. *SES/Workbench Reference Manual*, 1991.

[83] Scientific and Engineering Software Inc., Austin, Texas. *SES/Workbench Users Guide*, 1991.

[84] J. Shopiro. Extending the C++ task system for real-time control. In *C++ Workshop.* USENIX Association, 1987.

[85] S. Smith, M. Mercer, and B. Brock. Demand driven simulation: BACKSIM. In *Proceedings of the 24th ACM/IEEE Design Automation Conference*, pages 181–187, 1987.

[86] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1991.

[87] B. Stroustrup and S. J. A set of C++ classes for co-routine style programming. In *C++ Workshop*. USENIX Association, 1987.

[88] T. Thomasma and J. Madsen. Object-oriented programming languages for developing simulation-related software. In *Proceedings of the 1990 Winter Simulation Conference*, pages 482–489, 1990.

[89] K. Tonegawa. Simulating with feedforward information. In *Proceedings of the 1989 Summer Simulation Conference*, pages 1–4, 1989.

[90] B. Unger, J. Cleary, A. Dewar, and Z. Xiao. A multi-lingual optimistic distributed simulator. *Transactions of the Society for Computer Simulation*, 7(2):121–151, June 1990.

[91] J. van den Bos. PROCOL a protocol-constrained concurrent object-oriented language. In *Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming*, pages 149–151, Apr. 1989.

[92] P. van der Meulen. INSIST: Interactive simulation in Smalltalk. In *Proceedings of OOPSLA-87*, pages 366–376, Oct. 1987.

[93] P. Vaughan and D. Newton. Interactive Graphic MODeling environment (Igmod). In *Proceedings of the Object-Oriented Simulation Conference, 1992 Western Simulation Multiconference*, pages 25–30, 1992.

[94] P. Vaughan and D. Newton. PRISM: an object-oriented system modeling toolkit. *International Journal of Computer Simulation*, 1992.

[95] P. Vaughan, D. Newton, and R. Johns. PRISM: an object-oriented system modeling environment in C++ . Technical Report CAD-353-90, Microelectronics and Computer Technology Corporation, 1990.

[96] L. Velho and J. Gomes. A dynamics simulation environment for implicit objects using discrete models. In *2nd Eurographics Workshop on Animation and Simulation*, pages 183–189, 1991.

[97] R. Vujosevic. Object-oriented visual interactive simulation. In *Proceedings of the 1990 Winter Simulation Conference*, pages 490–497, 1990.

[98] A. Weinand, E. Gamma, and R. Marty. E++ - An Object-Oriented Application Framework in C++ . In *Proceedings 1988 OOPSLA*, pages 46–57, 1988.

[99] R. Wirfs-Brock and R. Johnson. Surveying Current Research in Object-oriented Design. *Communications of the ACM*, 33(9):104–124, Sept. 1990.

[100] S. Wolfram. *Mathematica, a system for doing mathematics by computer.* Addison-Wesley, second edition, 1991.

[101] Wolfram Research Inc. *MathLink External Communication in Mathematica*, Apr. 1992.

[102] D. Wyatt. A framework for reusability using graph-based models. In *Proceedings of the 1990 Winter Simulation Conference*, pages 472–476, 1990.

[103] A. Yonezawa and T. M. *Object-oriented Concurrent Programming.* Computer Systems Series. MIT Press, 1987.