Generic Flow Algorithm for Analysis of Interdependent Multi-Domain Distributed Network Systems

by

Lynn Ralph Feinauer

Dissertation submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Engineering

October 16, 2009
Blacksburg, Virginia

Keywords: Generic Algorithms, Graph Trace Analysis, Interdependent Systems, Multi-Domain Systems, Distributed Processing

Generic Flow Algorithm for Analysis of Interdependent Multi-Domain Distributed Network Systems

by

Lynn Ralph Feinauer

Dr. Robert P. Broadwater, Chair

Bradley Department of Electrical and Computer Engineering

(ABSTRACT)

Since the advent of the computer in the late 1950s, scientists and engineers have pushed the limits of the computing power available to them to solve physical problems via computational simulations. Early computer languages evaluated program logic in a sequential manner, thereby forcing the designer to think of the problem solution in terms of a sequential process.

Object-oriented analysis and design have introduced new concepts for solving systems of engineering problems. The term object-oriented was first introduced by Alan Kay [1] in the late 1960s; however, mainstream incorporation of object-oriented programming did not occur until the mid- to late 1990s. The principles and methods underlying object-oriented programming center around objects that communicate with one another and work together to model the physical system. Program functions and data are grouped together to represent the objects.

This dissertation extends object-oriented modeling concepts to model algorithms in a generic manner for solving interconnected, multi-domain problems. This work is based on an extension of Graph Trace Analysis (GTA) which was originally developed in the 1990's for power distribution system design. Because of GTA's ability to combine and restructure analysis methodologies from a variety of problem domains, it is now being used for integrated power distribution and transmission system design, operations and control. Over the last few years research has begun to formalize GTA into a multidiscipline approach that uses generic algorithms and a common model-based analysis framework. This dissertation provides an overview of the concepts used in GTA, and then discusses the main problems and potential generic algorithm based solutions associated with design and control of interdependent reconfigurable systems. These include:

- Decoupling analysis into distinct component and system level equations.
- Using iterator based topology management and algorithms instead of matrices.
- Using composition to implement polymorphism and simplify data management.
- Using dependency components to structure analysis across different systems types.
- Defining component level equations for power, gas and fluid systems in terms of across and though variables.

This dissertation presents a methodology for solving interdependent, multi-domain networks with generic algorithms. The methodology enables modeling of very large systems and the solution of the systems can be accomplished without the need for matrix solvers. The solution technique incorporates a binary search algorithm for accelerating the solution of looped systems. Introduction of generic algorithms enables the system solver to be written such that it is independent of the system type. Example fluid and electrical systems are solved to illustrate the generic nature of the approach.

Acknowledgements

I would like to thank Dr. Robert Broadwater for his encouragement, guidance and patience over the past five years. I am grateful for his friendship in addition to being my mentor.

I would like to thank Electric Distribution Design Inc. for providing the opportunity to work on the Distributed Engineering Workstation (DEW) for testing the algorithms described herein. Thanks also to Dr. Kwa-Sur Tam, Dr. William T. Baumann, Dr. A. Lynn Abbott, and Dr. James D. Arthur for serving on my committee.

I am grateful to my father, Earl, for instilling in me the desire to continue learning, and giving me encouragement along the long path that brought me to finish a doctoral program.

Special thanks to my wife, Patricia. My pursuit of a doctorate has required many sacrifices on her part. Although she may not have understood my need to finish this work, she never asked me to give up even when it meant time away from her and our children.

Contents

Figures

Tables

Chapter 1

Introduction

1.1 Introduction to Generic Analysis of Multi-Domain Systems

Physical systems of interest usually consist of more than one type with interdependencies between the systems, as shown in Figure 1.1. For example, modeling a shipboard system requires detailed knowledge of the electrical systems, the fluid delivery systems, hydraulics and other systems. The traditional approach has been to treat each system separately, then use solution values from one system as boundary or initial conditions in the solution of another system. Analysis equations are written for one specific type of system, assuming characteristics of the system. This results in software that is costly, rigid, and often good only for one problem or one category of problems.

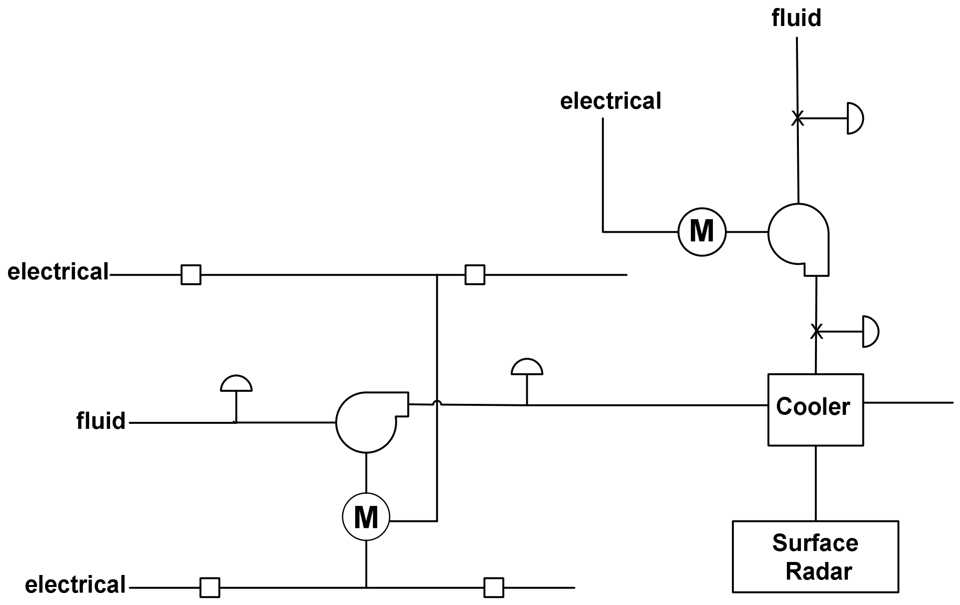

Figure 1.1 Interdependent electrical and fluid systems.

A widely used approach is to solve the systems of equations with sparse matrices [2]. Many sparse matrix solution solvers are available in the public domain, and many commercial solvers are available at very low cost.

A serious problem arises with approaches that incorporate matrix solution schemes, even if they rely on sparse matrix techniques. As the size of the model increases, the amount of storage required for the matrices and the solution time increase dramatically. For large models, on the order of millions or tens of millions of nodes, the required storage becomes costly and difficult to manage. Solution times become too slow to provide real-time analysis or practical “what-if” capabilities.

Graph Trace Analysis (GTA) [3] directly addresses these problems through a strategic combination of concepts from Physical Network Modeling [4] and Generic Programming [5]. GTA works off of iterator based system object models that are made to resemble standard operational diagrams such as Damage Control (DC) Plates [6] and engineering schematics. DC plates are analogous to a ship’s blueprints; they pertain to the ship’s watertight integrity.

GTA’s use of iterators is what makes it different from other approaches. GTA automatically generates and maintains the iterators it uses to define system connectivity as components are added, deleted, energized and switched. These iterators are used to structure other iterators and linked lists, which in turn are used to implement equation evaluations, manage data and coordinate analysis. Iterators use pointers, which “point to” specific locations in memory. This makes iterator based operations very fast. In addition, when a change occurs in the model, only the iterators that are directly affected by the change need to be updated. This combination speeds up analysis and makes it possible to analyze large models rapidly, even when a significant number of configuration changes occur during the course of analysis.

Generic Analysis and GTA are built around the ability they provide to decompose engineering equations and programming structures into distinct component and system levels. Generic Analysis takes this a step further and provides the additional capability to decompose equations and algorithms into things that are the same and things that are different, no matter what the type of system. Under this paradigm, the parts that are different are component terminal characteristics such as the voltage drop across a resistor when current is specified. For fluid components the across and through characteristics are pressure and flow, respectively.

The parts that are the same are:

- Components have across and through characteristics no matter what type they are.
- Information related status such as service loss and priority, and discrete behavior constructs such as on, off and failed are also the same.
- System level node and loop conservation equations are the same. This means that it is possible to develop a single algorithm called "flow" that will work with electrical, gas, fluid, thermal and other systems whose component models may be represented with across and through variables.

GTA is currently used commercially for design and planning of electric power systems and is also used as a model-based analysis engine for real-time control and operations, including control of distributed generation. Concept demonstration level power flow solutions have been performed on a model that was distributed across 8 processors, where the solutions with eight processors ran 6.2 times faster than the single processor based solution [7]. The largest model built to date using iterators and GTA is a 3 million node, multi-phase, unbalanced distribution system model that includes each major component in the system and detailed load information for each load.

Figure 1.2 shows a concept demonstration example of a GTA display for a notional ship model that was generated using line drawings from the Midway (CV-41). The model includes component models for ship service electrical system generation, 4160 volt and 450 volt distribution, vital loads, auxiliary system loads, fire main, chill water and mission priority driven interdependency connections between systems and loads. The model also includes logic controllers for ABT switches, an algorithm that graphically propagates and displays power and fluid flow loss, a component that marks dependency points between systems, and a priority management algorithm that propagates priority information across restoration paths and dependency points.

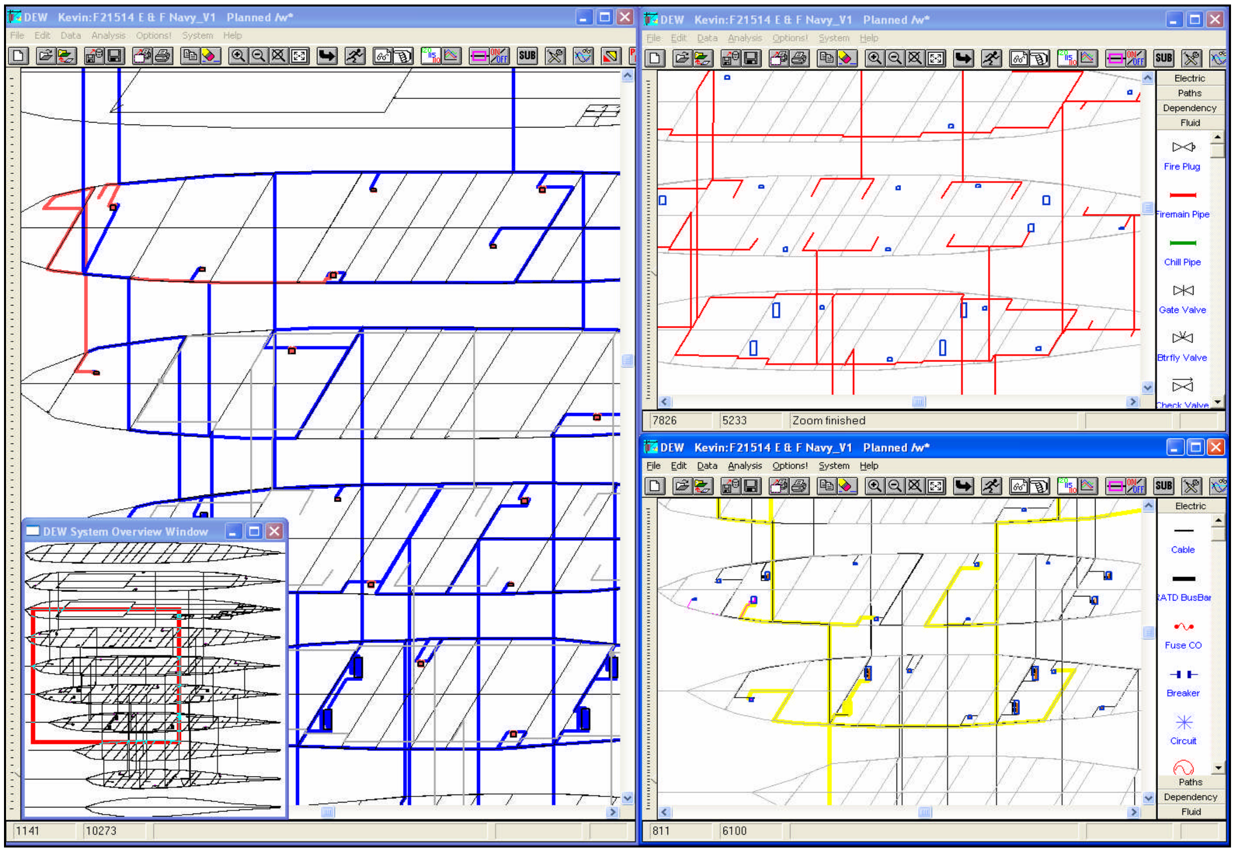

Figure 1.2 Prototype of integrated system decision aid display.

1.2 Challenges in Analysis of Interconnected Multi-Domain Systems

Modeling of physical systems requires large amounts of computer memory and CPU time. Many of the current solution techniques require inversion of matrices which are of order n^2, where n is the number of components. Even with recent increases in availability of cheap memory, the CPU time required to solve the matrices also increases quadratically with n. At the same time, detailed analysis of complex physical phenomena require more detailed models, increasing the number of required components in the model. This work implements a solution technique that eliminates the need for matrix solvers. It can handle models containing millions of components.

Current approaches used for analyzing interconnected multi-domain systems use specially developed software that is tailored to solving the particular problem. This results in single-use models and the software is expensive to build and maintain. Solution of interconnected systems often requires the use of multiple programs which are difficult to integrate with one another.

Most systems of interest involve interconnections between disparate system types. For example, fluid, gas and electric systems are interdependent on one another in a typical municipal distribution network. Current solutions typically treat the disparate systems separately and specify the interdependencies as explicit boundary conditions. The solution described in this work solves the systems simultaneously. This results in the ability to quickly determine the impact of one system on another, and can be used to analyze reconfiguration strategies when a component failure in one system causes outages in another system.

1.3 Research Objective and Contribution

The goal of this research is to develop a methodology by which multi-domain networks can be solved using generic algorithms. This builds on previous work [3] which laid a foundation for solving complex electrical systems using Graph Trace Analysis.

Polymorphic composition will be used to provide a flexible framework on which to base system components. This makes the architecture more amenable to changes that arise in late-stage development or implementation. Unless the problem is well-known in advance, getting an object hierarchy correct during the design phase is difficult or impossible. Most real-world problems entail surprises that cannot be foreseen at the design stage. Polymorphic composition also helps to avoid bloat that typically occurs as more and more behaviors and data are incorporated in lower levels of an inheritance hierarchy.

Another advantage of using polymorphic composition is that system simulation software can incorporate new types of components while executing. Thus, significant model updates, even to incorporating new types of components, can be performed without stopping and starting real-time analysis.

A system object will be abstracted independent of the underlying system type. This object will act as a container for the physical system components, with Graph Trace Analysis iterators suitably designed to traverse the component interconnections. The system object will be responsible for ensuring that the overall conservation equations are satisfied, i.e. that “across” variables around a closed loop sum to zero, and that “through” variables balance at a node (the flux into a node is equal to the flux out of the node).

The component objects are synonymous with the “edges” in the model. Connections between components of different physical types (such as electric to fluid) are modeled through a special dependency component as shown in Figure 1.3. The components are responsible for maintaining their own states through their "across" and "through" functions.

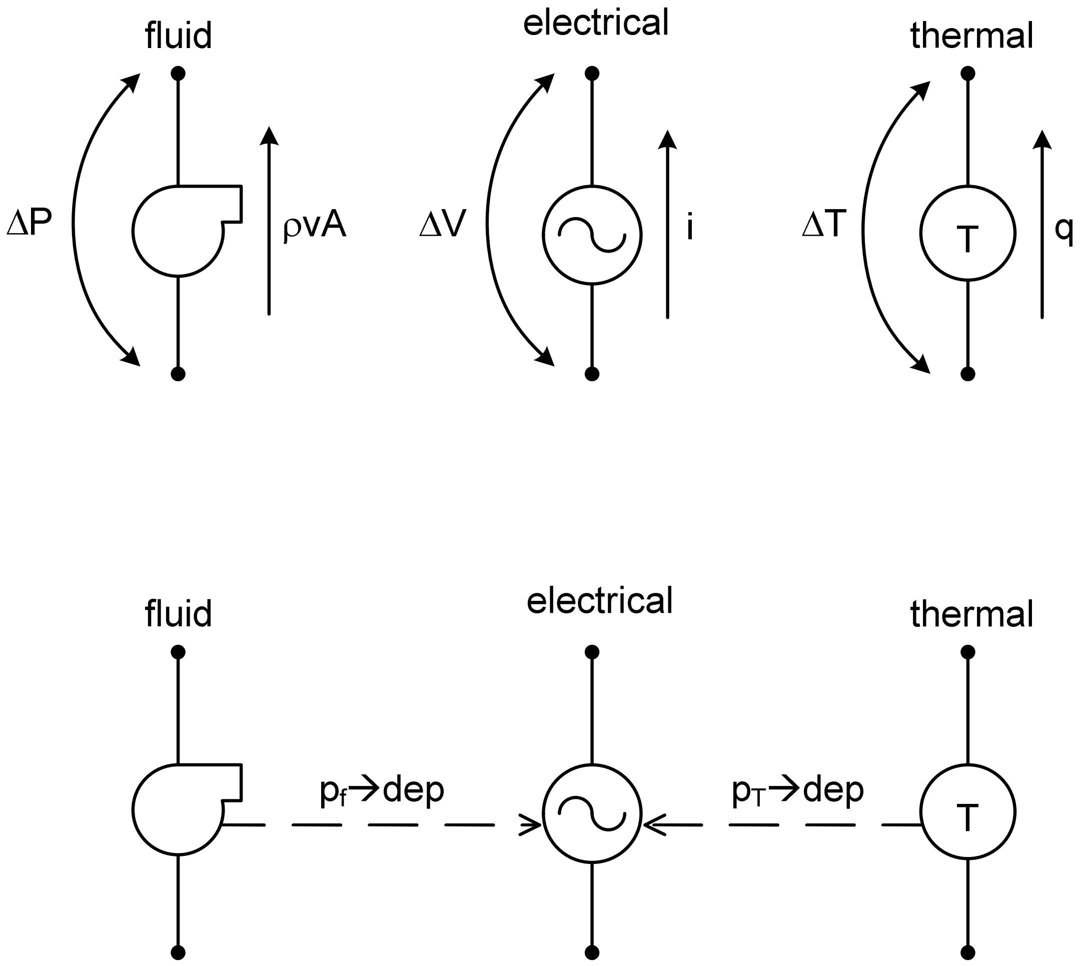

Figure 1.3 Interconnection of mixed physical system components.

A generic algorithm architecture will be developed. Individual components will supply their behaviors through standard interfaces such as "across( )", "through( )", "dialog( )", etc. In a manner similar to templates in C++ programming, generic algorithms will be written such that they are independent of the underlying type (in this case, the type of underlying physical system). The result will be a flexible, extensible, plug-and-play physical system simulation architecture.

Algorithms will be programmed to an interface that is exposed by the system object. Algorithms will use data from the system container, and supply their results to the container so that other algorithms can make use of the data. Access to the data will be accomplished via the GTA topology iterators.

This research extends previous research by creating a generic framework on which the programming abstraction is the network solution algorithms. Previous work in Object-Oriented programming and component programming has focused on abstraction in which the generic type is the function, for example Templates in C++.

1.4 Literature Review

A literature search has been performed in areas which impinge on the work herein described. These areas include:

- Analysis of multi-domain systems
- Critical Infrastructure Analysis
- Object-Oriented programming
- Generic programming

Whereas this dissertation is dependent on each of the above-mentioned fields, this study represents a novel approach to combining techniques from each, resulting in a dynamic new method of solving interconnected systems of disparate system types. The following subsections highlight differences in approaches between the existing literature and the approach taken by this research.

1.4.1 Multi-Domain System Analysis

Multi-domain system analysis is of vital concern to fields related to critical infrastructure, such as shipboard power. The maintainability of critical services and prioritizing these services throughout specific missions can be the determining factor between survival or loss of lives and ships. Reconfiguration of the system for restoration of services in shipboard distribution systems is detailed in reference [8]. This reference gives an overview of the interconnected nature of the shipboard systems and describes various approaches for solving the restoration problems, along with their shortcomings. It also proposes a general procedure for reconfiguration of service. The procedure uses linear optimization techniques to determine an optimal configuration without explicitly performing any load flow or power flow analyses. In contrast, the approach taken by this dissertation is to model the system components explicitly such that the component and system equations can be calculated without requiring computationally expensive matrix inversions.

Zivi [9] discusses complexity of interconnected systems and the increasing dependency on commercial off-the-shelf software in shipboard power systems. He points out that the state of existing software systems is insufficient to provide dependable control of these interconnected systems during attack conditions. The methodology described in this dissertation addresses the shortcomings of off-the-shelf software that Zivi discusses in his paper.

Wang et al. [10] examines the multi-domain analysis problem through the use of graphic modeling methods. Specifically, they examine the block diagram method, bond graph modeling, the functional model method, and the object-oriented modeling method. One of their conclusions is that the functional modeling method provides an efficient approach for building a hierarchical model for multidisciplinary systems. The approach applies an effort-flow structure to lumpedparameter system model. In contrast, this dissertation presents a methodology for analyzing large systems in which the components carry detailed physical properties. Algorithms can exchange their results with each other through the system object.

Weston et al. [11] describes the need for an integrated modeling environment that consists of a set of interconnected subsystem models to simulate an actual Naval ship system for the analysis of damage control and reduced manning scenarios. They propose an interrelationship mapping in which “the functional and physical system decompositions are performed in order to determine the interdependencies between subsystems, equipment, and components at several levels of detail”.

Sinha et al. [12] presents an overview of the state-of-the art in modeling and simulation. Their discussion includes an examination of graph-based modeling and briefly discusses the three graph-based paradigms: bond graphs, linear graphs, and block diagrams. Tam [13] presents an overview of approaches that have traditionally been used to model systems of systems, specifically for reconfiguration analysis. He discusses the characteristics of different approaches, the types of equations that are involved in the solution techniques and characterizes them as continuous, discrete time or s domain. The work described in this dissertation uses Graph Trace Analysis which is based on linear graph theory.

Tam and Broadwater [14] present a framework for modeling interdependent engineering systems. This dissertation extends their work and provides a generic flow algorithm that can be applied across different system types.

1.4.2 Critical Infrastructure Analysis

The analysis of critical infrastructures is a broad topic that includes such systems as the national electric grid and shipboard power and service systems. Sustainability of the national power grid is of vital importance to the nation and is a concern of the regional power companies and the department of Homeland Security. Shipboard power and service systems are vital to the mission with which the ship is tasked. In the case of a warship for example, the ship must maintain the ability to sustain the functionality of the onboard weapons regardless of other less vital systems.

Svendsen and Wolthusen [15] present a model for analyzing critical infrastructures with interactions and interdependencies. Their treatment does not treat the domain-specific analysis in detail; rather, it treats the interactions between systems statistically. In their conclusion, they make the following statement:

> “This necessarily is less accurate than domain-specific models, but it does allow for the modeling of interactions and interdependencies which such domain models cannot capture.”

In contrast, this dissertation provides the framework upon which domain models can be built which model and analyze such interactions and interdependencies through the use of physical network modeling and generic algorithms.

Rinaldi [16] reviews modeling and simulation techniques for critical infrastructure systems. He concludes that the “models and simulations of individual infrastructures are rather well developed today”; however, he concludes that there are significant challenges faced by developers of models and simulations of the interdependencies between critical infrastructure systems. The generic algorithm framework developed in this dissertation addresses many of these concerns.

1.4.3 Object-Oriented Programming

Interfacing with early computers was accomplished through direct input to the computer memory through manual setting of switches and output in the form of lights to represent “bits” of information. Computer access was by necessity close to the hardware and required the programmer to break problems into steps that the computer could “understand”, i.e. into the on/off switch sequences that represented data. Program functions were accomplished by wiring or rewiring computer units into configurations specific to the desired functionality.

As computer languages developed, abstraction was away from the hardware and more toward the way problems are manually solved; however, changes in abstraction have been slow to develop. A new methodology may take a decade or more to be implemented and widely accepted. For example, a programming paradigm that has been widely used is known as structured programming and was in use by 1968 or earlier [17]. Structured programs are comprised of blocks of procedural code which are split into subsections or subroutines. However, structured programming continued to require programmers to think in terms of the way a computer would approach solving a problem.

In the early 1970s the Smalltalk language [18] introduced object-oriented programming to a wide audience. Object-oriented programming is distinguished by the introduction of classes that encapsulate data and functions that operate on the data. Instantiations of the class are called objects and objects communicate with each other through messages.

Although object orientated constructs more closely resemble the physical world than procedural code, pure object orientation can be too restrictive when the system configurations and conditions change radically over time. This research presents a hybrid approach that combines features of object-oriented coding with other modeling techniques that provide the ability to discover and replace algorithms at run time.

Li and Broadwater [19] present the concept of object-oriented engineering frameworks for analysis of power distribution systems. The framework uses a layered architecture consisting of a component layer, an iterator layer, an algorithm layer and a distributed algorithm layer. By programming to interfaces the framework reduces coupling between software components. The work presented in this dissertation builds on some of these concepts to provide an object-oriented, generic algorithm framework.

Bonfé et al. [20] describes a modeling language that provides an object-oriented framework for multi-domain physical systems. Their modeling approach is based on Bond Graph theory and extends the Universal Modeling Language (UML) to include domain-specific concepts. Their work is focused on computer-controlled systems; specifically the design of industrial processes and mechatronic systems.

1.4.4 Generic Programming

Generic programming is a method by which algorithms can be written such that they can be applied to parameters of varying types. For example, a sort routine can be built that sorts integers, floating point numbers, character strings, etc. using one routine. Yewang, et al. [21] discusses the advantages and disadvantages of using generic programming as opposed to Object-Oriented code. Their approach is based on an inheritance hierarchy. As described later in this dissertation, implementation of polymorphic behaviors through containment is more flexible for solving problems in real world situations.

Bellman and Landauer [22] describe a model-based integration methodology for analyzing complex systems. Their approach is based on the use of wrapping knowledge bases to determine interfaces at run time. The wrappings describe the various ways the resources can be used for the solution of a particular problem. This is similar in some aspects to the approach taken in this dissertation; however, this research provides a flexible framework that is discoverable at runtime and that does not require the problem definition a priori.

Rysavy et al [23] present a generic library built on the Microsoft .NET framework which can be used to generate generic algorithms. It is built on top of the Standard Template Library (STL) and the Boost Graph Template Library (BGL). They define generic algorithms in a style similar to the STL; that is they define a generic algorithm to be computational methods that have parameterized inputs and outputs. The paper presents a basic framework for passing generalized classes as input and output parameters if they satisfy a preset interface. The framework presented in this dissertation also relies on the published interface to establish the contract between the algorithm and the object that invokes the algorithm. The generic algorithms can interact with each other through the system object, and different system types can be solved together through the use of system dependency objects which determine the flow of information between systems.

1.4.5 Other Literature Resources

The literature search also yielded other resources which did not fit into the four outlined categories. For example, Modelica [24] is a high-level object-oriented language that is used for modeling multi-domain systems. Chakrabarty and Mendonça [25] presents a methodology for defining a set of visualization tools for critical infrastructure systems. Voigt and Wachutka [26] describe a method for modeling micro-systems in which components interact with each other as constituent parts of a Kirchoffian network. The approach uses across and through variables to represent the physical quantities of driving forces and fluxes, respectively. Bachelet et al. [27] describes a methodology for producing extensible, generic algorithms, specifically for operations research. Their framework is based on polymorphism through inheritance, whereas this dissertation employs polymorphism through containment.

1.5 Dissertation Outline

The remainder of this dissertation consists of five chapters. Chapter 2 describes the Graph Trace Analysis methodology employed in the research. The groundwork is laid for using iterators to traverse the linear graph which represents the physical system components that are being analyzed. The Object Constraint Language (OCL)[28] is introduced to provide a precise manner in which model constraints can be written.

Chapter 3 extends the concepts of Object Oriented Programming and Generic Programming to produce a framework for the development of Generic Algorithms. The modeling of physical systems can be treated in a generic manner due to the similarities in the underlying system laws [4]. System-level balance equations can be written that are applicable to the system model regardless of system type. The components are responsible for calculating their internal states based on boundary and initial conditions.

Chapter 4 presents a generic flow algorithm that can be used to solve systems of different physical types. For example, the algorithm can be used to solve electric, fluid and gas systems. Interdependencies between different physical systems can be modeled, allowing for the simultaneous solution of interconnected heterogeneous systems. Since the solution scheme does not require inversion of matrices, scaling of the solution to large network systems is feasible with moderate memory requirements.

Chapter 5 presents results of applying the flow algorithm outlined in Chapter 4 to the solution of various problems. The computational results of two fluid flow problems are compared with analytical results to demonstrate the accuracy of the algorithm.

Chapter 6 presents conclusions drawn from the development and application of the generic flow algorithm developed in this research effort and outlines possible future research that may build thereon.

Chapter 2
Physical System Modeling

2.1 Procedural versus Object-Oriented Languages

Early computer architectures were severely restricted in the amount of available memory (both run-time core and external storage), and limited by slow execution speeds. To accommodate the modeling and analysis of physical systems, applications were therefore restricted to very coarse modeling, or extremely long execution times.

Computer modeling frequently used assembly language in order to get the fastest processing available. Complicated mathematical algorithms were written in terms of the problem being solved, and therefore could not readily be used in the solution of other problems. This tight coupling of the solution to the problem space became commonplace in programs written with procedural languages such as Fortran, PL/1 and ALGOL. Programs were designed to execute in the same manner as a person would process the evaluations by hand.

Each of the procedural languages incorporated a method for branching and for jumping from one coding section to another. This gave rise to the GOTO statement which often led to what is familiarly referred to as spaghetti code. In a letter to the editor of Communications of the ACM [29], Edsger Dijkstra enumerated some of the reasons that such coding constructs were harmful. He also pointed out that we are "geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed". Structured programming largely eliminated the GOTO statement through the judicious use of more structured constructs such as looping (for, while, do-while, etc.), "if-then-else" blocks, and switch/case statements.

However, structured programming suffered from some of the same problems as those previously encountered. One reason for this is that the coding artifacts were still designed with the computer software solution space in mind.

2.2 Graph Trace Analysis

Computational modeling of physical systems can be facilitated through the use of graph theory [30]. A graph is a simple geometrical figure that consists of two sets of objects which are generally referred to as nodes and edges. Nodes are linked together by edges or lines. Since lines are used as connectors between nodes, the graphs are generally known as linear graphs.

Graph Trace Analysis [31] uses the concept of a directed graph, i.e., a graph in which there is a direction associated with the edges between the nodes. Figure 2.1 shows a simple directed graph with nodes 1-6.

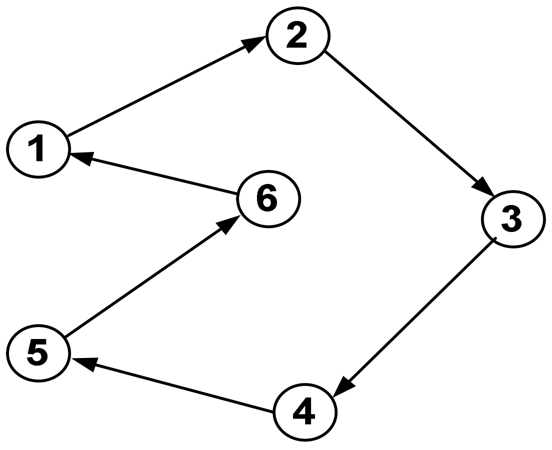

Figure 2.1 Example directed graph.

The graph objects represent physical objects, such as transformers, capacitors, pumps, etc. Whereas many other systems employ both nodes and edges, GTA uses a one-to-one correspondence between the edges in a “system container” and the components in the physical network model. The resulting edge-edge model refers to only one type of object, “system” component. In other object-oriented approaches, node-edge models are employed, resulting in added complexity from an iteration point of view because two types of objects must be manipulated, nodal objects and edge objects. With GTA, only one type of object must be manipulated.

For radial systems four topology iterators are defined for tracing through the graph: forward (f), backward (b), feeder-path (fp) and brother (br). All components that are related by these four iterators have one-and-only-one reference source. In a system with loops, each component may be fed from multiple sources, but only one of the sources is its reference source. The positive direction of the through variable (such as current or flow) of a component is assumed to be away from its reference source. Likewise, the positive direction for the across variable (such as voltage or pressure drop) of a component is assumed to be away from its reference source.

Forward and backward iterators may be used to trace through every component in a system, where in a single trace each component is encountered once and only once. Assume a trace is positioned at a given component referred to as the “current” component. The forward iterator may be used to return a component that is directly fed from (i.e. receive flow from) the current component but is further removed from the reference source, or a component that is not directly fed from the current component. In a forward trace from a given component, all components that receive flow from the given component (relative to the reference source for the components) must be traced prior to encountering any component that does not receive flow from the given component.

The feeder path iterator returns the previous component that is physically connected in the direction leading back to the current component's reference source. Note that components returned in the forward or backward traversals are not necessarily physically connected to the current component, whereas components returned in the feeder-path traces are physically connected. The feeder path iterator may be used to determine the other three iterators, forward, backward, and brother.

The brother iterator returns the first component in the forward trace direction that does not carry any flow from the reference source which also goes through the given component. When the forward iterator is equal to the brother iterator, a physical "jump" takes place in the system topology.

Figure 2.2 illustrates topology iterators for a system consisting of four components and its corresponding graph containing four edges. In the figure integers 1-4 are used as unique identifiers for the edges in the graph, and the integer 0 is used to indicate the end of a trace. If the starting component for a forward trace is not specified, the first component in the forward trace is always a reference source, as represented by component 1 in the figure. In terms of iterators, component 1's backward (b) and feeder-path (fp) iterators do not point to another component. Their value is zero (or null).

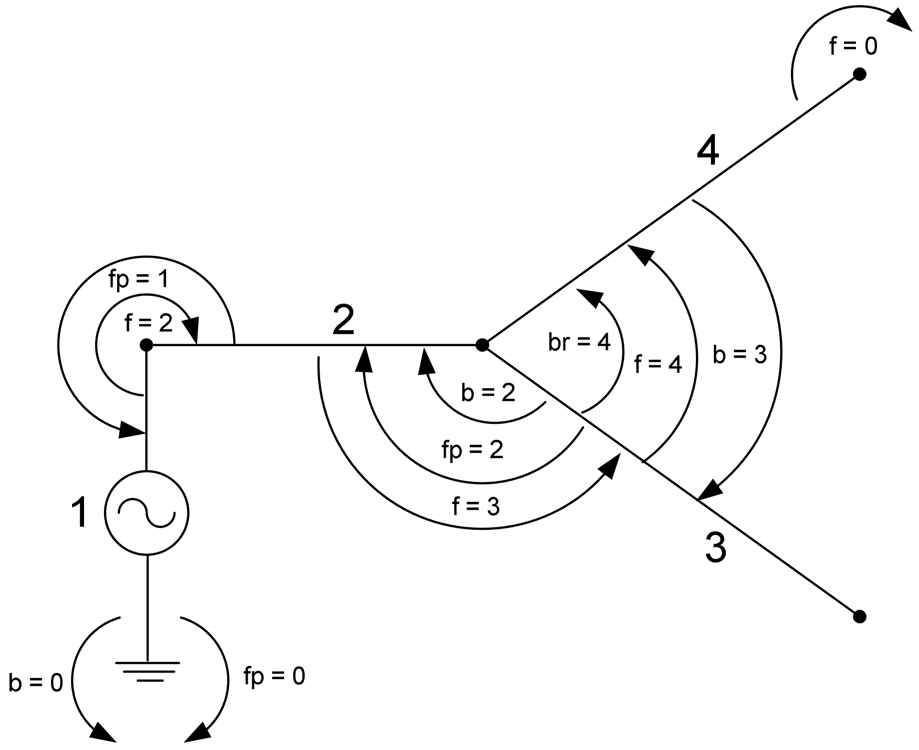

Figure 2.2 Using iterators to trace through system.

Adjacent components in the forward or backward traversals are not necessarily physically connected, whereas adjacent components in the feeder-path trace are physically connected. Each component may be fed from multiple sources, but each component must have only one reference source. When the forward iterator is equal to the brother iterator, this indicates a physical “jump” in the system topology (for example, going from component 3 to component 4 in Figure 2.2). For radial systems, traces form branches of a tree, each having a single beginning and a single end. However, when a switch is closed, a loop may be formed, and if a loop is formed the edge that represents the closed switch is referred to as a cotree element [32].

When a loop is formed an additional iterator is used: the adjacent (adj) iterator. Relative to a current component in a given trace, if an adjacent iterator returns a component that is referenced to a different source, an independent loop involving two different sources has been created. If the adjacent iterator returns a component that references the same source as the current component, an independent loop within the system has been created, the elements of which have the same reference source.

Automated traces through the system implemented with iterators are used to generate "trace" sets. A forward trace beginning at a component p is created by successively applying the forward iterators from component p through the system and yields the set FT_P. Similarly, a backward trace uses the backward iterators yielding the set BT_P, a feeder-path trace uses the feeder-path iterators yielding the set FPT_P, and the brother trace uses the brother iterators yielding the set BRT_P.

The use of topology iterators to perform traces through the system graph is what distinguishes GTA from other analysis approaches. Using GTA, system equations and analysis algorithms can be written in terms of topology iterators and trace sets. In GTA iterators may be implemented using C++ pointers, and successive applications of an iterator involves repeatedly dereferencing the pointer. For example, starting at p, the first application of the forward iterator is $p \to f$, the second application is $p \to f \to f$, and so forth, as long as the successive iterations do not yield a zero value.

GTA iterators are used for managing system equations from physical network modeling. GTA iterators may be used to formulate and manage system node or loop equations. Because GTA tracks loop connection points, or cotrees, iteration can be used to solve both radial and looped equations. A large electric power system model would require the use of approximately a 9 million $\times$ 9 million node-voltage-matrix. This same system is solved with GTA using forward and backward iterations combined with a 3125 $\times$ 3125 loop current matrix. This reduction in matrix size is significant. Although the nodal matrices typically used in solution approaches are sparse and sparse matrix solution schemes are available, this type of solution process does not scale well. Managing computer memory for solving large system matrices is problematic and is also not well suited for use with distributed processing.

Trace sets are continually updated when something happens to change the topology such as when a component is added to the system or a switching event occurs. Examples of basic trace sets that are used in different types of analysis are feeder paths and segments [3][32]. Segments are groups of components that are connected together through a common set of sectionalizing devices such as switches or circuit breakers, or in the case of a fluid system, isolation valves. If any one component in a segment fails, all the other components in that segment must be isolated together with the failed component from the rest of the system.

GTA algorithms can be written such that the solution of the system equations is independent of the type of system being solved. For example, an algorithm that solves for system flows can be used to solve for electrical flows, fluid flows or gas flows. As in object-oriented programming, the container does not require any knowledge of what the contained objects are. The system object is responsible for ensuring that system-wide constraints are met, such as the requirement that the sum of all "across" variables (those that apply across the component terminals such as voltage or pressure drops) around a closed loop must equal zero.

2.3 Interdependent Systems

Interdependencies between systems of different types, such as fluid and electrical systems can be modeled by attaching a dependency component at the junction point between the systems. The format for GTA dependency components is structured after the "dependency" format used in the Unified Modeling Language (UML)[33]. Figure 2.3 shows a simplified shipboard system model that contains electrical distribution, chill water, vital mission loads and abstract loads that represent missions. Dependency components used to structure relationships between different system types may transfer energy, priority information and status across system junction points such as pump motors and heat exchangers.

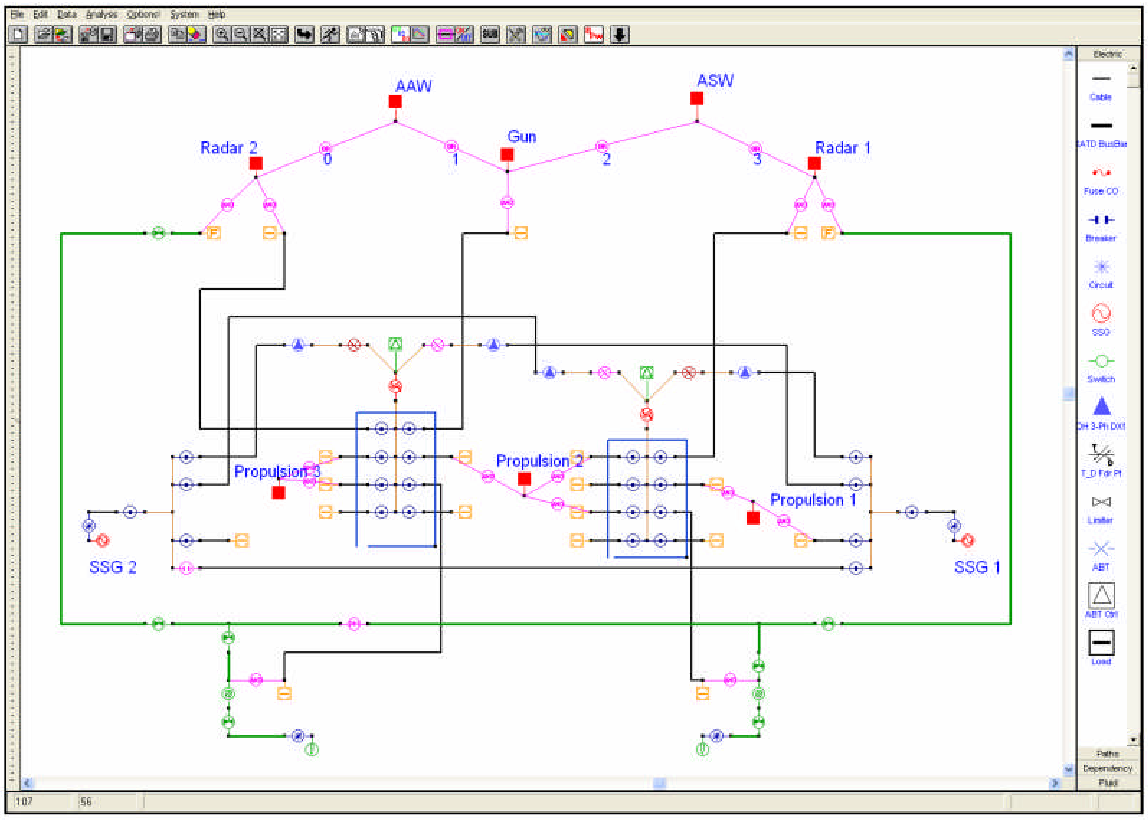

Figure 2.3 Simplified integrated ship system model.

Using GTA with a detailed system drawing and user-specified information (mission priorities) for Anti-air Warfare (AAW) and Anti-surface Warfare (ASW), the following analyses can be performed:

- Damage isolation that identifies devices to be operated to isolate damage effects to the smallest number of system components possible.
- Flow analysis that evaluates system operating constraints, such as pressure or voltage limits for every component in the system. The flow analysis is used to evaluate the viability of proposed configurations.
- Mission priority management that propagates priority information from mission logical loads, down through reconfigurable electric and fluid service system runs, to vital and auxiliary system service loads.
- Reconfiguration for restoration analysis that evaluates possible restoration paths and switching operations according to component status, operating constraints and service load priorities.

Chapter 3 Generic Algorithms

3.1 Use of a Common Algorithm Interface

Analysis algorithms can be written with a common interface such that they can be run on the system model regardless of the system type that is being solved. This provides a method for solving multi-domain system problems where dependencies among the systems are involved. This methodology is referred to as Generic Analysis because of its resemblance to the object-oriented principles of generic programming [34].

Inheritance has been considered the standard method for providing code reuse in object-oriented programming. It is used to provide a hierarchical lineage for objects that share common attributes (variables) and behaviors (functions). A parent or base class defines those characteristics that apply to all classes that inherit from it. Subclasses modify or add to the inherited characteristics to implement their own distinct attributes and behaviors.

Polymorphism is one of the distinguishing characteristics of an object-oriented programming language, referring to the ability to call a function through a reference to the base class. For example, if the base class is ChessPiece and one of the defined behaviors is Move(), each of the chess pieces moves in a different way. In this case, each subclass would provide its own Move() functionality. Polymorphism allows an object of any of the chess pieces to be declared as the base type and have the appropriate behavior determined at run time.

The implementation of polymorphism through inheritance is fragile if the analysis software design changes, which is most often the case when solving real world emergent problems. Rather than implement polymorphism with inheritance, as illustrated in Figure 3.1a, the approach taken here is to implement polymorphic behaviors using containment and an interface, as illustrated in Figure 3.1b. This results in a flexible solution and a framework for solving interdependent, multi-domain systems. In C++ with the inheritance approach, the polymorphic function f() is accessed through a component pointer as p→f(); using containment, it is accessed as f(p), where p may be a component from an electrical, fluid, gas, or other system type. Again, note that GTA only manipulates components, where each component knows what type it is.

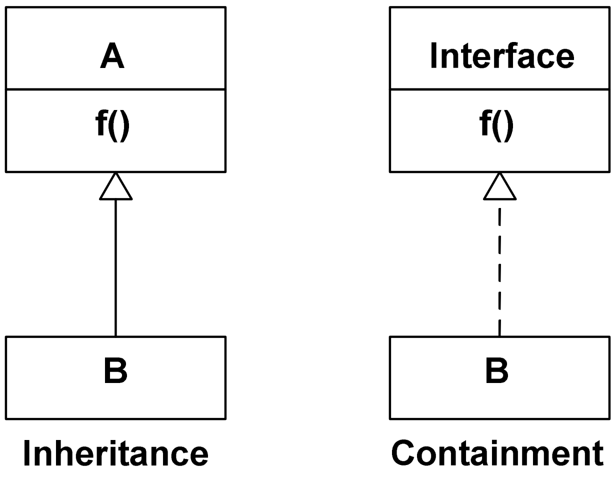

Figure 3.1 Polymorphism through inheritance versus containment.

Regardless of the system type, there are two system connection equations that involve through (e.g. flows) and across (e.g. voltage drops) variables which must be satisfied. One system connection equation requires that the flow into a node must equal the flow out of the node. In electrical systems this is Kirchhoff’s first law or conservation of current; in fluid systems this is conservation of mass. Additionally, the across variables around a closed loop must sum to zero. In electrical systems the voltage drops around the loop must sum to zero; in fluid systems the pressure drops around the loop must sum to zero. Since the form of the equations is the same regardless of the system type, a single system solver may be used for all system types.

Individual components are responsible for calculating their internal state and response to external stimuli. Generic “through” and “across” functions depict the calculation of variables that flow through components (currents, mass flow, etc.) and those that apply across components (voltage, pressure drop, etc.), respectively. Calculation of the through variable for component p is expressed as

through(p),

and calculation of the across variable for component p is expressed as

across(p).

Generally, due to nonlinearities, even in radial systems numerical iteration is required to reach a converged solution. In generic programming a container object provides iterators to access its contained objects. In the approach presented here, the system of components is the container, and the GTA trace iterators traverse the components within the system container. Algorithms use the iterators to access the contained objects. An iterator may also be used to return an object that represents a system. Thus, p may be an element of a system S, and S may be an element of a system-of-systems, SS, as expressed by

$p \in S \in SS.$

Using this process iteration may be performed over all of the components in a system-of-systems.

GTA algorithms are programmed to an interface which is exposed by the system container, as illustrated in Figure 3.2. Algorithms relate calculated results to objects in the container so that their results may be accessed by other algorithms. Thus algorithms work together through the container. For example, if data required by an algorithm already exists, the container merely provides it to the algorithm. However, if it has not yet been calculated, the container can request that another registered algorithm be invoked to generate the missing data. The container is used to exchange data among algorithms.

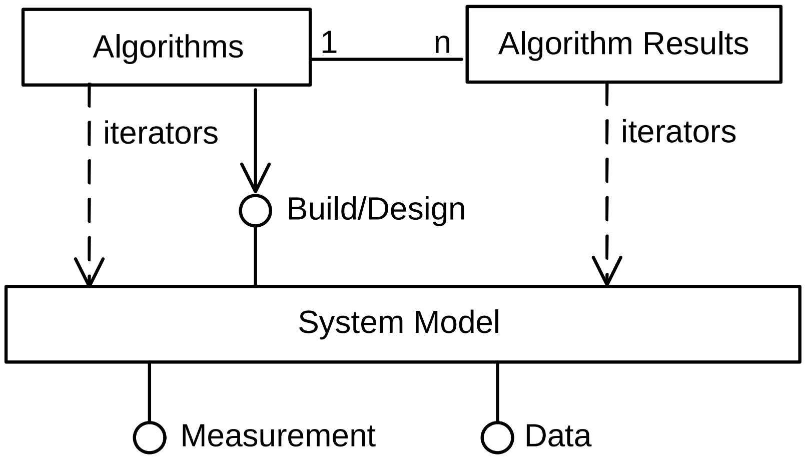

Figure 3.2 Algorithms programmed to an interface.

3.2 Polymorphism through Inheritance

One of the primary characteristics of object oriented programming is polymorphism. It is most commonly implemented using an inheritance tree, where derived classes behave differently in response to the same message. . Using inheritance, the basic attributes (variables) and behaviors (functions) that are shared by related classes are defined in the base or root class from which the other classes are derived. As classes grow outward from the base class, functionality becomes more specialized.

Figure 3.3 shows an electrical system that will be used in this and subsequent sections to illustrate the differences in the method for solving systems using inheritance, static containment and dynamic containment.

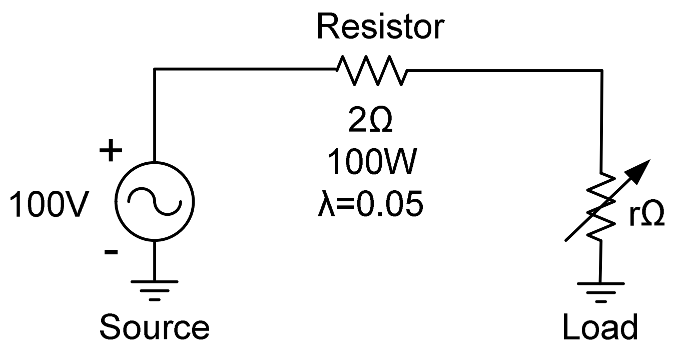

Figure 3.3 System for analysis.

System components share basic characteristics, therefore a base class named Cmp can be defined from which the Source, Resistor and Load components are derived as shown in Figure 3.4. Inheritance is typically used to describe “is a” relationships; for example in this case Sources, Resistors and Loads are all types of “Cmp”.

The base class defines the attributes and behaviors that are applicable to all of the derived components. Attributes and behaviors that are specific to derived components are defined only at the derived level. All components have an across and a through behavior; however since each component evaluates these functions differently, the base class contains only a very basic default behavior.

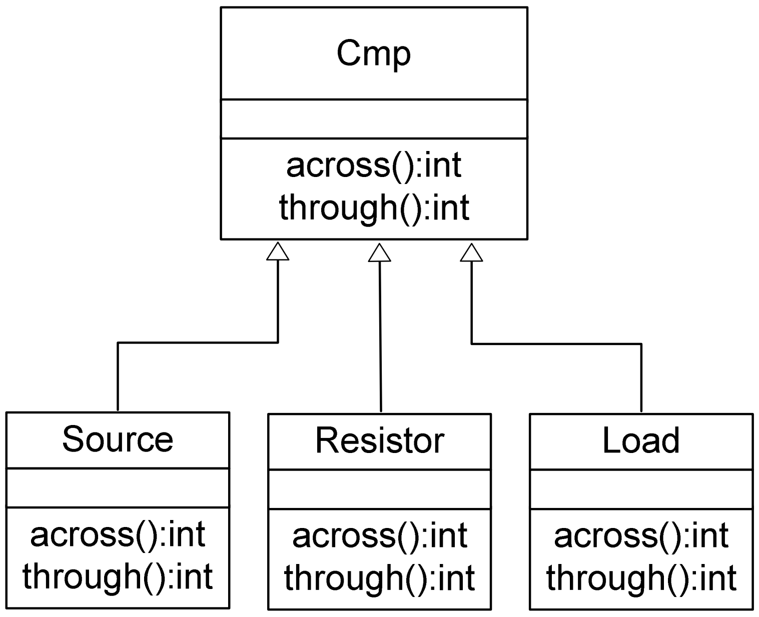

Figure 3.4 Component class inheritance.

Figure 3.5 shows an example C++ class definition for the component hierarchy resulting from the use of inheritance. The base class contains virtual methods for across() and through(). If a derived class does not override these methods, the default behaviors implemented by the base class are used.

For this example, the constructor for the Source class takes one argument, the voltage. The Resistor class constructor requires three arguments: the resistance in ohms, the power rating in watts and the failure rate. The constructor for the Load class requires three arguments: the resistance in ohms, the lower resistance limit in ohms and the upper resistance limit in ohms.

```
class Cmp
{
    public:
        Cmp();
        ~Cmp();
        virtual int across();
        virtual int through();
        ...

    private:
        string m_strName;
        double m_dAcross;
        double m_dThrough;
        ...
};

class Source : public Cmp
{
    public:
        Source(double dVoltage=0.0);
        ~Source();
        int across();
        int through();
        ...
};

class Resistor : public Cmp
{
    public:
        Resistor(double dResistance=0.0, dCurrent=0.0,
            dEfficiency=0.0);
        ~Resistor();
        int across();
        int through();
        ...

    private:
        double m_dEfficiency;
        ...
};

class Load : public Cmp
{
    public:
        Load(double dResistance=0.0,
            double dLowResistance=0.0,
            double dHighResistance=0.0);
        ~Load();
        int across();
        int through();
        ...

    private:
        double m_dLowResistance;
        double m_dHighResistance;
};
```

Figure 3.5 Class definition for polymorphism through inheritance.

Modeling this system using inheritance based polymorphism involves instantiating one each of the derived classes within the system as follows:

```
Source* pSrc = new Source(100.0);
Resistor* pFixedR = new Resistor(2.0, 100.0, 0.05);
Load* pVarR = new Load(3.0, 1.0, 10.0);
```

A pointer of type Cmp* can point to any of these derived classes, and the across() and through() methods can be accessed through the base class pointer. This provides a convenient method for storing arrays of derived components and enables the container to iterate over the components without the container requiring knowledge of the individual derived types. For example, the following code illustrates storing an assortment of derived components in an STL vector.

```
typedef  vector<Cmp*>  CmpArray;
CmpArray aCmps;
Source* pSrc1 = new Source(125.0);
aCmps.push_back(pSrc1);
Resistor* pRes1 = new Resistor(5.0, 150.0, 0.04);
aCmps.push_back(pRes1);
Source* pSrc2 = new Source(110.0);
aCmps.push_back(pSrc2);
Load* pLoad1 = new Load(5.2, 1.0, 10.0);
aCmps.push_back(pLoad1);
Resistor* pRes2 = new Resistor(6.5, 125.0, 0.01);
aCmps.push_back(pRes2);
...
```

In this example, the system container can iterate over the collection of components, and call their across() and through() methods through the generic Cmp pointer, as follows:

```
CmpArray::iterator iter;
Cmp* pCmp = NULL;
for (iter = aCmps.begin(); iter != aCmps.end(); iter++)
{
    pCmp = *iter;
    pCmp->across();
    pCmp->through();
}
```

Polymorphism is implemented by C++ and many other computer languages through virtual tables commonly referred to as "vtables". When a method is declared with the "virtual" keyword, a pointer, called a vptr ("virtual table pointer") is added to the lookup table. When the method is called on the object, the object's vptr is used to look up the appropriate function based on the object's type. This allows the program to dynamically determine which function to call; that is the linkage is done at runtime rather than at compile time.

If a derived class does not override a virtual function defined in the base class, it inherits the base class implementation of the function. In the case of a pure virtual function (one for which the base class does not provide an implementation), all derived classes must provide an implementation for that virtual function. As described in [35], a typical virtual function call executes as follows:

- The compiler determines if the call is being made via a base-class pointer and that the function is virtual.
- It then locates the entry in the vtable using the offset or displacement.
- The compiler generates code that performs the following operations:
  - Select the pointer being used in the function call from the third level of pointers in the triple indirection used for virtual functions.
  - Dereference that pointer to retrieve the underlying object.
  - Dereference the object's vtable pointer to get to the vtable.
  - Skip the offset to select the correct function pointer.
  - Dereference the function pointer to form the name of the actual function to execute and use the function call operator to execute the appropriate function.

Consider a CmpArray (vector<Cmp*>) object that contains a Source, a Resistor and a Load. Figure 3.6 shows the flow of virtual function call baseClassPtr→through() when baseClassPtr points to object Resistor.

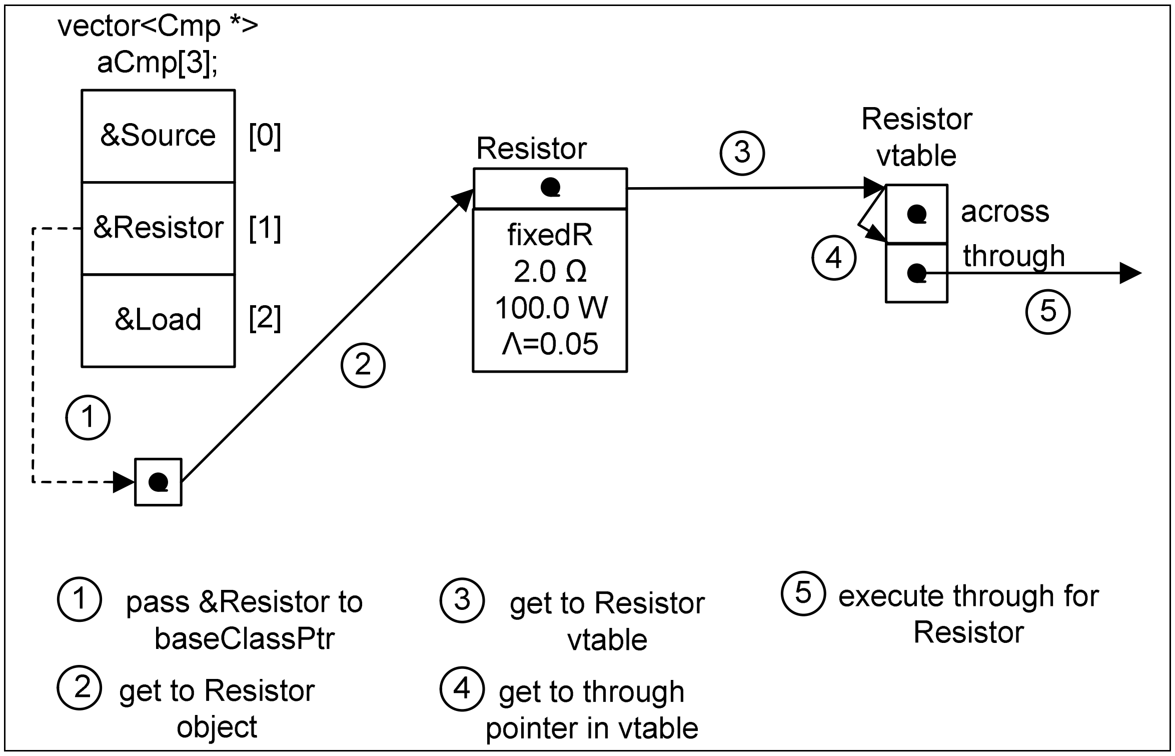

Figure 3.6 Flow of virtual function through base class pointer.

The example circuit shown in Figure 3.3 can be solved using GTA forward and backward iterators. Given a setting of 3Ω for the load (i.e. r=3), the solution proceeds as follows:

1. Assume an initial current, for example, let the current be 0.0A.
2. Using a forward trace, at the resistor, the across value (voltage) will be 100V.
3. The through value (current) at the load will be I = E / R which gives 100V / 3Ω = 33.33A
4. Now, following a backward trace, the through value at the resistor is also 33.33A.
5. The across value for the resistor is then E = 100V – (2Ω * 33.33A) = 33.33V.
6. The forward trace iterator is then used to advance to the load, for which the through value is calculated to be I = 33.33V / 3Ω = 11.11A.

7. After applying the forward and backward traces for a few iterations, the solution converges to a voltage drop across the resistor of 60V and a current of 20A.

3.3 Polymorphism through Static Containment

Inheritance suffers from many potential problems, especially when the hierarchy is deep. As more classes are added to the hierarchy, a subclass may inadvertently inherit from more than one parent class. Also, as changes are made to derived classes, the base class may require changes, additions or deletions. This requires rebuilding all modules that depend on the class definitions.

Polymorphic behavior can also be achieved through containment. Containment is used to describe “has a” relationships; in this case, the system has Source, Resistor and Load components. Figure 3.7 shows an example C++ class definition for the component hierarchy resulting from the use of static containment.

In this approach, individual components each contain their own specific parts information. The generic information that applies to all components is contained in the Cmp class and specific information about the part is stored in class Part. For example, centrifugal pumps require characteristic curves that express the head (delta pressure) of the pump versus the flow rate, whereas electrical transformers require much different kinds of information. The base class does not, however, contain virtual functions for the across and through behaviors. Instead, these functions require a pointer to the object, which is cast to a pointer to the base class type Cmp.

```
class Part
{
    public:
        Part();
       ~Part();
        ...
};

class Cmp
{
    public:
        Cmp();
        ~Cmp()
        getPart() { return m_pPart; }
        setPart(Part* pPart) { m_pPart = pPart; }
        ...

    private:
        Part* m_pPart;
        ...
};

typedef  vector<Cmp*>  CmpList;

class Source
{
    public:
        Source();
        ~Source();
        int across(Cmp* pCmp);
        int through(Cmp* pCmp);
        ...

    private:
        Cmp* m_pCmp;
        double m_dVoltage;
        ...
} ;

class Resistor
{
    public:
        Resistor();
        ~Resistor();
        int across(Cmp* pCmp);
        int through(Cmp* pCmp);
        ...

    private:
        Cmp* m_pCmp;
        double m_dVoltage;
        double m_dCurrent;
        ...
} ;

class Load
{
    public:
        Load();
        int across(Cmp* pCmp);
        int through(Cmp* pCmp);
        ...

    private:
        double m_dLowResistance;
        double m_dHighResistance;
        ...
} ;

class Circuit
{
    public:
        Circuit();
        ...

    private:
        CmpList m_cmpList;
        ...
} ;

typedef  vector<Circuit*>  CircuitList;
```

Figure 3.7 Polymorphism through static containment

Static containment is a compile-time coding construct; that is, the class definitions are processed during the code compilation stage. Use of containment allows for changes to individual components without the "ripple effect" of a change in one class requiring changes in other classes within a class hierarchy. Solution of real-world engineering problems seldom fits a standard pattern that can be predicted and programmed beforehand. Containment helps solve this problem by making the software more amenable to change without completely restructuring class definitions.

Applying polymorphism through containment to the sample problem, an object of type Circuit, can be defined that contains a vector of pointers to components. In this case, the vector contains pointers to one Source, one Resistor and one Load component. Although the components are different types, each component can be accessed through a pointer to the base class Cmp. In this case, the Circuit class contains separate implementations for each of the individual classes Source, Resistor and Load. For example, the Cmp class contains an across method with function signature:

int across(Source* pSource);

As each of the components is retrieved from the component list, the across and through methods can be invoked without explicit knowledge of the underlying type as was the case with inheritance. However, in this case the specific method is determined through the use of a parameterized function. This provides the indirection necessary to invoke the proper class implementation as shown in Figure 3.8.

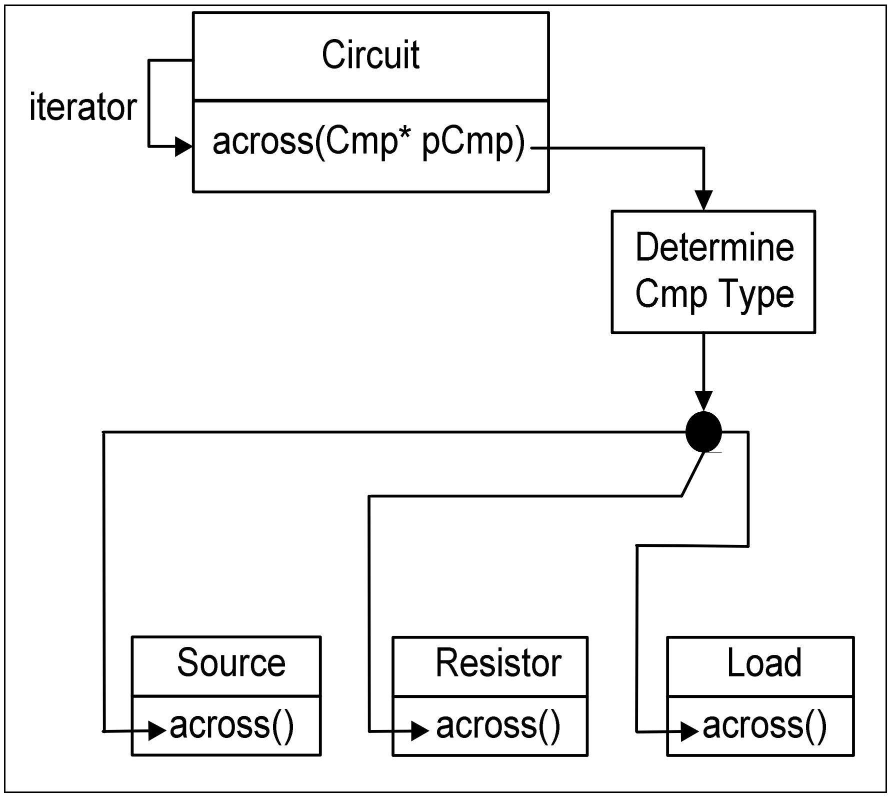

Figure 3.8 Container iterating over components in the list.

The following shows an example of how to programmatically call the across method through the respective component pointer.

```
Circuit* pCircuit = new Circuit();
if (pCircuit != NULL)
{
    // Do error handling
    …
    return false;
}

Source* src = new Source(100.0);
pCircuit->GetCmpList()->push_back(src);
Resistor* pFixedR = new Resistor(2.0, 100.0, 0.05);
pCircuit->GetCmpList()->push_back(pFixedR);
Load* pVarR = new Load(3.0, 1.0, 10.0);
pCircuit->GetCmpList()->push_back(pVarR);
...

// Iterate over the container, invoking the across behavior for each component
CmpList::iterator iter;
for (iter = pCircuit->GetCmpList()->begin(); iter != pCircuit->GetCmpList()->end(); iter++)
{
    pCmp = *iter;
    pCircuit->across(pCmp);
}
```

Since the Circuit component does not know the underlying types of the components, a helper function is supplied. It uses the component's type number to determine which of the specific component across functions to call. For example:

```
int Circuit::across(Cmp* pCmp)
{
    Source* pSrc=NULL;
    Resistor* pRes=NULL;
    Load* pLoad=NULL;
    try
    {
        switch (pCmp->iType)
        {
            case CMPTYPE_SOURCE:
                pSrc = (Source*)pCmp;
                return pSrc->across();
            case CMPTYPE_RESISTOR:
                pRes = (Resistor*)pCmp;
                return pRes->across();
            case CMPTYPE_LOAD:
                pLoad = (Load*)pLoad;
                return pLoad->across();
            ...
            default:
                return -1;  // Error condition
        }
    }
    catch (...)
    {
        // Log an error and return error condition
        char szErr[MAX_ERR];
        if (pCmp)
        {
            sprintf_s(szErr, MAX_ERR, “Unknown type ‘%d’ encountered in across function!”,
                pCmp->iType);
        }
        else
        {
            sprintf_s(szErr, MAX_ERR, “Null pointer encountered in across function!”,
                pCmp->iType);
        }
        DisplayErrorMsg(szErr);
        return -1;
    }
}
```

In the static containment case, the container class Circuit defines a parameterized function whose responsibility it is to make sure the proper component class across() and through() functions get called, based on the component type. The determination can be made by using an element of the base Cmp class that uniquely identifies the component type. In the coding presented herein, an integer element named iType is used.

This methodology is static since all of the constructs need to be defined at compile time; however, the framework is more flexible than in the case with polymorphism via inheritance and less prone to problems associated with inheritance trees. The base Cmp class now has no virtual functions for the across and through behaviors; these behaviors are required elements of the specific component classes. Although this approach is more flexible than the inheritance hierarchy approach, changes to class definitions, associations and constraints require the recompilation and linking not only of the changed class but potentially of the entire application.

3.4 Polymorphism through Dynamic Containment

With static containment, the coding constructs are determined at compile time. This means that whenever the class definitions need to be revised, the code must to be recompiled. This makes replacement of the algorithms inconvenient and time-consuming when solving large, interconnected systems that rely on real-time SCADA information. If a new algorithm becomes available using static containment, the program must be stopped, the new algorithm must be installed on the machine or machines, and the program must be restarted. In order to overcome these problems, dynamic linking or late binding can be combined with GTA to provide polymorphic behavior through dynamic containment.

Dynamic linking has been used for decades; mainly to provide system functionality through application programming interface (API) functions. In Microsoft Windows the functions are exposed by Dynamic Link Libraries (DLLs); in Unix and many other systems the functions are exposed by shared libraries. Both types of libraries provide a method for packaging executable program chunks that can be used by many programs simultaneously. In essence, DLLs make a program ***runtime modular***. DLLs can be used to implement polymorphism using composition, referred to as dynamic polymorphic containment. Dynamic polymorphic containment provides flexibility that is difficult or impossible to achieve through the static polymorphic containment.

The class layout for the dynamic containment class is similar to the static containment case; however, dynamic containment requires a mechanism for examining the modules and providing the equivalent of a virtual table for the polymorphic functions. Figure 3.9 shows an example C++ class definition for the component hierarchy resulting from the use of dynamic containment and coding to create and use a run-time function map that is used to select the proper polymorphic functions based on the component type.

```
#define DllExport __declspec(dllexport)
#define DllImport __declspec(dllimport)

class Part
{
    public:
        Part();
       ~Part();
        ...
};

class Cmp
{
    public:
        Cmp();
        ~Cmp()
        getPart() { return m_pPart; }
        setPart(Part* pPart) { m_pPart = pPart; }
        ...

    private:
        Part* m_pPart;
        ...
};

typedef  vector<Cmp*>  CmpList;

class Source
{
    public:
        Source();
        ~Source();
        int across(Cmp* pCmp);
        int through(Cmp* pCmp);
        ...

    private:
        Cmp* m_pCmp;
        double m_dVoltage;
        ...
} ;

class Resistor
{
    public:
        Resistor();
        ~Resistor();
        int across(Cmp* pCmp);
        int through(Cmp* pCmp);
        ...

    private:
        Cmp* m_pCmp;
        double m_dVoltage;
        double m_dCurrent;
        ...
} ;

class Load
{
    public:
        Load();
        int across(Cmp* pCmp);
        int through(Cmp* pCmp);
        ...

    private:
        double m_dLowResistance;
        double m_dHighResistance;
        ...
} ;

class Circuit
{
    public:
        Circuit();
        ...

    private:
        CmpList m_cmpList;
        ...
} ;

typedef  vector<Circuit*>  CircuitList;

...
// Type definitions for dynamic class access
typedef int (*DIALOG_FCN)(HWND, Cmp*);
typedef int (*ACROSS_FCN)(Cmp*);
typedef int (*THROUGH_FCN)(Cmp*);
typedef BOOL (*CMPEXT_INITDLL)();

typedef struct cmpext
{
    DIALOG_FCN lpfnDialog;
    ACROSS_FCN lpfnAcross;
    THROUGH_FCN lpfnThrough;
} CMPEXT, *PCMPEXT;

...
// Functions used internally by the main CmpExt DLL
int dlg(HWND hwnd, CMP* pCmp);
int across(Cmp* pCmp);
int through(Cmp* pCmp);
```

```
// Functions exported by the CmpExt DLL                          106
extern “C” DllExport int cmpExtDlg(HWND hwnd,                    107
    Cmp* pCmp);                                                  108
extern “C” DllExport int cmpExtAcross(Cmp* pCmp);                109
extern “C” DllExport int cmpExtThrough(Cmp* pCmp);               110
                                                                 111
typedef map<int, CMPEXT*> CMPEXTMAPTYPE;                         112
                                                                 113
```

Figure 3.9 Polymorphism through dynamic containment.

A standard template library (STL) map object can be used to produce an efficient mapping between components and their implementations. As shown in line 112 of Figure 3.9, the component ID is used as the key to an entry that maintains pointers to the across, through and dialog functions for the specific component type. The function pointers are obtained by doing a runtime examination of the DLL specified for the component type in the configuration file. This is initially done at program startup; however, new components can be added “on-the-fly” as discussed in the Section 3.4.1. If a component type is not specified in the configuration file, the across, through and dialog functions are set to default implementations. For example, the default across function could be to set the outlet property to the same value as the inlet value.

The functions are exported from the DLL as C-style functions in order to avoid problems associated with name-mangling which is done differently between C++ vendors. Although the exported function is C-style, the DLL can implement C++ classes and methods internally; only the interface function is specified using C binding. This is similar to Component Object Model (COM) programming, though the implementation and access are less restrictive than in COM.

As an example, consider the following lines in a CmpExt configuration file:

```
<dll name="CmpExtFluid.dll">
    <across name="CentPumpAcross" cmp="158" />
    <through name="CentPumpThrough" cmp="158" />
    <dialog name="CentPumpDialog" cmp="158" />
</dll>
```

The “dll” tag declares the name of the DLL from which functions will be taken until the next “dll” tag is encountered. For each component, an “across”, “though” and “dialog” tag are specified. As mentioned previously, if a definition is not specified, the default implementation. During program startup, the configuration file is read and the map is created for the components. The “name” attributes of the “across”, “through” and “dialog” tags specify the entry point names for the respective functions. Figure 3.10 shows selected portions of the CCmpExt class that illustrate how the functions are loaded into the map:

```
int dialog(HWND hwnd, Cmp* pCmp)
{
    CallMainDialog(goWH.hwndDraw, pCmp);
    return 0;
}

int across(Cmp* pCmp)
{
    Cmp* pFp = pCmp->pFp;
    if (pFp)
    {
        pCmp->adAcross_re[0] = pFp->adAcross_re[0];
        pCmp->adAcross_re[1] = pFp->adAcross_re[1];
        ...
    }
    return 0;
}

int through(Cmp* pCmp)
{
    return 0;
}

bool CCmpExt::ProcessFunctionNode(CComPtr<IXmlReader> pReader,
    int type, string& strDLL)
{
    ...
    // Insert the function entry into the map
    ACROSS_FCN lpfnAcross;
    THROUGH_FCN lpfnThrough;
    DIALOG_FCN lpfnDialog;
    int cmpID = atoi(strCmp.c_str());
    CMPEXT* pItem = m_cmpExtMap[cmpID];
    if (pItem == NULL)
    {
        pItem = new CMPEXT;
        if (pItem == NULL)
```

```
            {                                                            38
                // Log an error and bail out                             39
                LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_MEM, "");         40
                return false;                                            41
            }                                                            42
            m_cmpExtMap[cmpID] = pItem;                                  43
            pItem->lpfnAcross = across;                                  44
            pItem->lpfnThrough = through;                                45
            pItem->lpfnDialog = dialog;                                  46
        }                                                                47
                                                                         48
        HMODULE hLibrary = LoadLibrary(LPCTSTR)strDLL.c_str());          49
        if (hLibrary == NULL)                                            50
        {                                                                51
            // Log the error and continue with the next entry            52
            LogMsg(ERR_MSG, ghInstCmpExt, ERR_LOADLIB_FAILURE, "t",      53
                strDLL.c_str());                                         54
            return true;                                                 55
        }                                                                56
        ...                                                              57
        switch (type)                                                    58
        {                                                                59
            case NODE_TYPE_ACROSS:                                       60
                if (strName.size() > 0)                                  61
                {                                                        62
                    lpfnAcross = (ACROSS_FCN)GetProcAddress(hLibrary,    63
                    strName.c_str());                                    64
                    if (lpfnAcross == NULL)                              65
                    {                                                    66
                        LogMsg(ERR_MSG, ghInstCmpExt,                    67
                            ERR_NULL_ACROSS_ENTRY,  “tt”, strName.c_str(),   68
                            strDLL.c_str());                             69
                        lpfnAcross = across;                             70
                    }                                                    71
                }                                                        72
                pItem->lpfnAcross = lpfnAcross;                          73
                break;                                                   74
                ...                                                      75
        }                                                                76
        return true;                                                     77
    }                                                                    78
```

Figure 3.10 Loading the function entry points into the CmpExt map

If a component already has an entry in the map, the statement at line 33 retrieves a pointer to the CmpExt item that currently exists. If the map does not have an entry, a CmpExt item is allocated for it and it is inserted into the map.

The following shows an example of how to programmatically call the across method through the respective component pointer. Note that this is the same as for the static containment case. The differences are handled by the system and are transparent to the application programmer.

```
Circuit* pCircuit = new Circuit();
if (pCircuit != NULL)
{
    // Do error handling
    …
    return false;
}

Source* src = new Source(100.0);
pCircuit->GetCmpList()->push_back(src);
Resistor* pFixedR = new Resistor(2.0, 100.0, 0.05);
pCircuit->GetCmpList()->push_back(pFixedR);
Load* pVarR = new Load(3.0, 1.0, 10.0);
pCircuit->GetCmpList()->push_back(pVarR);
...

// Iterate over the container, invoking the across behavior for each component
CmpList::iterator iter;
for (iter = pCircuit->GetCmpList()->begin(); iter != pCircuit->GetCmpList()->end(); iter++)
{
    pCmp = *iter;
    pCircuit->across(pCmp);
}
```

3.4.1 Hot-Swappable Analysis Modules

Programming techniques often take their inspiration from hardware counterparts. For example, modular programming breaks a program into smaller, well-defined components, thus allowing for reuse of common code and aggregation of the pieces together to solve a specific problem.

One method for implementing the polymorphic behavior is similar in concept to the virtual tables used by the C++ compiler when virtual functions are declared in the base class of a class inheritance tree. The implementation for a particular component is specified in a DLL; several may coexist in the same DLL. A configuration file is then used to set up a map associating the behaviors (function interfaces) for the component to their respective entry points within the DLL.

A container class, Circuit, acts as a container for the components as in the previous section. The container has no knowledge of the types of components it contains. It sends messages to (invokes functions on) each component and ensures that the general conservation laws are satisfied for the subsystem which the Circuit represents.

Figure 3.11 shows a separate execution thread that monitors for changes in the executable module directory. When a new module is found, it is examined to determine if it is an extended component definition. If it is, the information it contains is added to the function pointer map.

Capabilities provided by DLLs can be dynamically determined and loaded on demand. By combining this capability with the generic algorithm approach presented previously, a hot-swappable algorithm mechanism can be provided. The convenience of this feature is analogous to the convenience provided by hot-swappable hard drives. Normally, if a hard drive fails, or if another drive containing auxiliary data needs to be added to the system, the machine must be shut down, the new hard drive installed and the machine must be powered on again. With hot-swappable drives, one simply removes the old drive and snaps in the new one.

In a similar fashion, by employing dynamic discovery of analysis algorithms, or model components that have been programmed to a defined interface, the system object shown in Figure 3.2 may become immediately aware of new algorithms or components whose DLLs are placed into the executable directory. Because the program does not have to be shut down and restarted, this allows computations to continue uninterrupted, creating a seamless transition with algorithm or model component additions. This capability is necessary when running large models under real-time conditions.

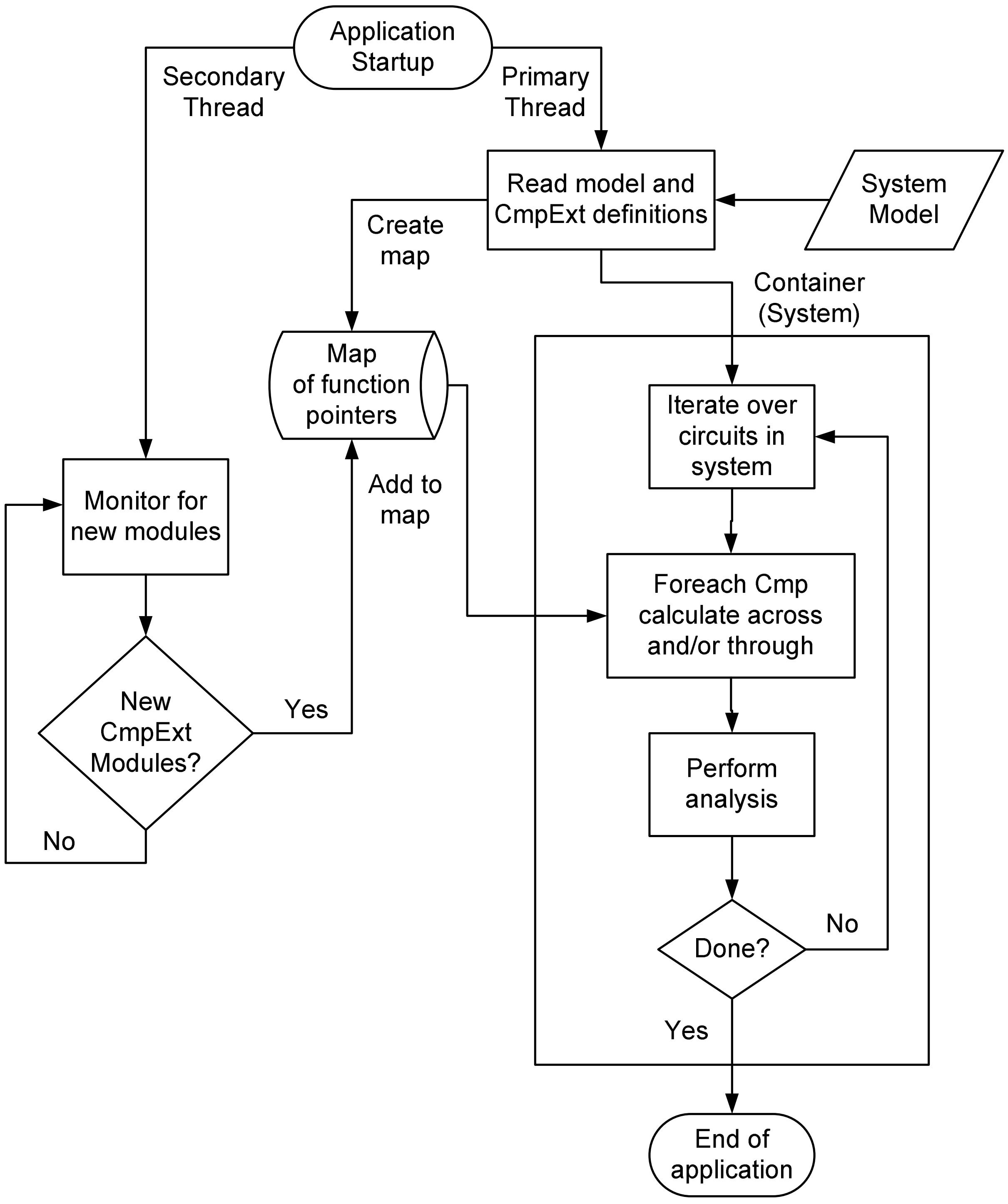

Figure 3.11 Analysis application using hot-swappable modules.

Since all the algorithms operate on the same components in the model and the components have unique identifiers, the algorithms are able to exchange data through the system-object-interface. For example, a flow algorithm can share its output with a reliability algorithm. If an updated version of the algorithm becomes available, it can be replaced while the system continues to run. Furthermore, fluid and electric systems and their interdependencies can be analyzed using shared analysis algorithms [36].

3.4.2 Generic System Analysis Modules

C++ provides "template" functions and classes; that is functions and classes that can operate on a variety of data types. For example, a template function might be used to provide a generic "sort" algorithm that operates on integer data, floating point data and for character strings. When coding is encountered that calls a template function, the compiler deduces the data type(s). In order to match a template function signature, the type of each parameter must exactly match the type of the template arguments.

Generic modules do not need to be limited to simple functions. The modules that have been described in previous sections can be complex combinations of functions and/or classes that work together to solve problems as long as they comply with the contract (interface definitions). Through the use of the generic across and through interfaces, algorithms can be created that solve interdependent systems which are composed of multiple physical system types. For example, the dependence of a centrifugal pump on the electrical power input is one example of the interconnection between fluid and electrical systems.

Generic component modeling for physical networks was outlined by Blackwell [4] in 1968. Physical measurement of properties across and through components using appropriate meters is modeled by the use of generic across and through functions as described in Section 3.1 of this dissertation. Similarities between the conservation equations for different systems allow for the solution of systems of components described in this manner. For example, the conservation of energy equation for fluid systems corresponds to Kirchhoff’s Voltage Law for electric systems, and the fluid conservation of mass equation corresponds to Kirchhoff’s Current Law.

The approach described in this dissertation is to model system components which are responsible for calculating their across and through properties based with the inputs defined through their GTA connections. Since the component interface is generic, the container (circuit or system) does not require knowledge of the system type. The container invokes the across and through functions through the generic interface and ensures that the conservation laws are satisfied. The container iterates until convergence is obtained.

This approach is different from others that are currently used in network system analysis. It makes no a priori assumptions regarding the system type. This allows for the combination of electrical, fluid, gas and other systems together in one computer model. Algorithms written to the generic interface then operate on the components through the container object.

This approach also avoids the need to solve large systems of matrices by iterating the equations explicitly; matrices are only required for nodes which represent the cotrees of the system. Section 4.2 describes an alternative technique of obtaining convergence at the cotrees using the bisection method. Thus, no matrices are required with a search approach. By eliminating or reducing the need for matrix solutions, analysis of very large models can be done at or near real-time speeds.

Chapter 4 Generic Flow Algorithm

4.1 Generic Algorithms Using GTA

In a fluid system the across variable represents the pressure drop across the component and the through variable represents the mass flow through the component. Let us consider the calculation of pressure drops and fluid flows for a radial system that has only one pressure source. The extension to a system with multiple sources builds on the solution to this problem and incorporates cotree flows [4].

The algorithm is written in terms of the GTA iterators described in Section II using a concise modeling notation which borrows from the Unified Modeling Language (UML) Object Constraint Language (OCL) [28]. OCL is a declarative language that sets forth precisely how objects are required to behave; it specifies completely the pre- and post-conditions that must be satisfied for each object and also specifies how objects interact with each other. In OCL the "***forall***" operator iterates over all elements of the collection; the "***select***" operator specifies a Boolean expression which is true for each of the elements of the resulting subset of the collection; and the "***collect***" operator applies a calculation to each of the elements in the collection and places the resulting elements into a new collection.

Using GTA notation, given the flows (or an estimate of the flows) through all components, and starting at a pressure source referenced by the iterator ps, component pressure drops are calculated by:

FTps→collect(pf[p]→pd = across(p)) (1)

where

FTps = set of all ordered components in forward trace from component ps

pf[p]→pd = variable which stores pressure drop across component p for pressure-flow algorithm pf

across(p) = polymorphic function which calculates the pressure drop across component p

The arrow, →, is a polymorphic operator. The first arrow (i.e. →) in (1) operates on the set FTps. It is the set member operator and causes the collect() operator to be applied to every p in the ordered set FTps. The second arrow in (1) is used to access the member variable pd associated with the algorithm pf and the component p.

Given the pressure drops across all components (or an estimate of the pressure drops), in order to obtain the flow through all components, two reverse traces from the first component in the reverse trace, pe, are performed as given by

RTpe→collect(pf[p]→q = 0) (2)

RTpe→collect(pf[p]→q + = through(p),

pf[p→fp]→q + = pf[p]→q) (3)

where

RTpe = set of all ordered components in reverse trace from component pe

pf[p]→q = variable which stores flow through component p

Although the across and through behaviors have been described for a fluid system, the equations can be applied to any system type. Since the across and through functions are polymorphic, generic flows through the system can be determined using the following algorithm:

Initialization

FT $\rightarrow$ collect (p | pf[p]$\rightarrow$pr = nominal pressure) (4)

FT $\rightarrow$ collect (p | pf[p]$\rightarrow$q = 0) (5)

CT = FT $\rightarrow$ select (p | p$\rightarrow$cs = = COTREE) (6)

Equations 4 and 5 are used to set nominal pressure and flow at the start of the calculation. Equation 6 builds a set of all cotrees in the system which marks the points where loops have been created. [4]

Following the initialization, the following forward pressure trace and reverse flow trace are performed. The pressure trace starts at a source. The flow trace starts at the first component in the reverse trace.

Iteration

BT$\rightarrow$collect (pf[p]$\rightarrow$q + = pf[p]$\rightarrow$load,

pf[p$\rightarrow$fp]$\rightarrow$q + = flow[p]$\rightarrow$q) (7)

FT$\rightarrow$collect (pf[p]$\rightarrow$pr = pf[p$\rightarrow$fp]$\rightarrow$pr + across(p)) (8)

The iteration is continued until

$$\Delta pf[ps] \rightarrow q < \varepsilon,$$

where ps is the source component.

4.2 Treatment of Looped Flow

In a radial system with only one source, the fluid flow can always be calculated directly back to the source through a reverse trace. In a system with one or many loops, the cotree element that marks each loop behaves as an ending component with no intrinsic flow. Consequently, a single reverse trace will not assign any flow through the cotree element. After the first reverse trace, the pressure at a cotree will generally be unequal to the pressure at its adjacent component (components A and B in Figure 4.1). Here a bisection algorithm uses the cotree pressure difference to determine how the fluid flow through the cotree should be adjusted. When the system of equations has converged, the pressure difference between the cotree and its adjacent should be sufficiently close to zero.

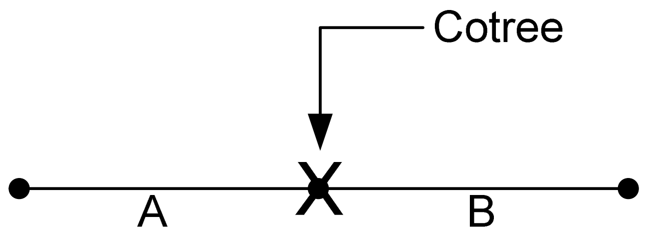

Figure 4.1 Components connected at a cotree.

For the systems considered here, a bisection algorithm was used to manipulate cotree flows. At each iteration the pressures at the cotree element and its adjacent component are compared and a mass flow rate is assigned to flow from the component of higher pressure to the component of lower pressure, as given by

```
if ( pf[A]→pr > pf[B]→pr )
    through (A) = through(A) + FlowStep
    through (B) = -through(A)
```

where FlowStep = mass flow rate bisection step

This flow rate is increased for each cotree flow until a sign change occurs between the compared pressures. The bisection step flow rate is then decreased as

FlowStep = FlowStep/k

where k > 1.0

Convergence is reached when the change in flow at all sources and pressure difference at each cotree is sufficiently small. This is expressed by

S→forall(Δpf[p]→q < ε1)

CT→forall(| pf[p]→pr – pf[p→adj]→pr | < ε2)

where S = set of all sources

CT = set of all cotrees.

For systems studied here, convergence was reached at a pressure difference of 0.0001 psi.

In a system with multiple cotrees, changing one cotree may affect the convergence of another so that a minimum step size is necessary to prevent premature convergence. The flow at each cotree is adjusted once per iteration, and the test for step size is determined individually for each cotree. Although the difference in pressure at any given cotree ideally should decrease with each iteration, the ongoing adjustments to other cotrees in the system may prevent its immediate convergence. An oscillation is identified when the pressure difference value for a cotree repeats itself at every other iteration. At that cotree, the step size is reset to its original value and the cotree must converge again. The larger step size allows the flow values at the cotree to quickly bypass the oscillation point, while the other cotrees are freed from their dependence on the oscillating cotree.

4.3 Distribution of System Model

A method of improving speed and efficiency of calculations is to distribute tasks across multiple processors. In the past, this required large mainframe or supercomputers; however, low-cost desktop machines now come with multiple processors on the same chip and make distribution of the solution more readily accessible.

One of the benefits of the GTA method is that it provides natural locations at which the model can be split; namely at cotrees. Consider Figure 4.2 which shows a simple system consisting of two sources, two loads, and a few additional miscellaneous components. A cotree is formed between component 4 and component 7, where current i_4 is the current that flows from component 4 to the right-hand load. This must match the current from component 7 to the load. In this example, the system can be split apart at the cotree to form the two subsystems shown in Figure 4.3. The solution of the left-hand subsystem can be handled by one processor while the right-hand subsystem is handled by a second processor, exchanging only the current i_4 at the boundary.

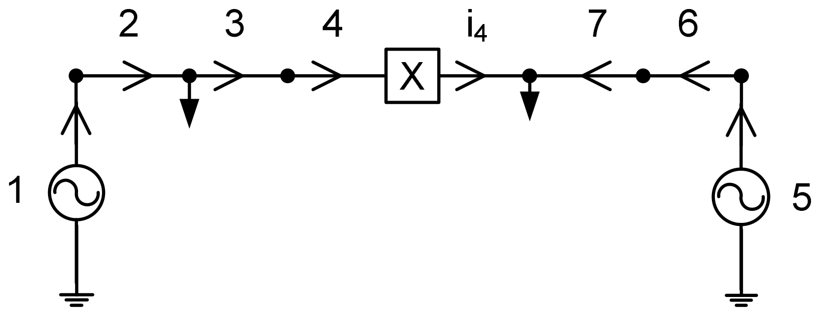

Figure 4.2 Model with two sources and a cotree.

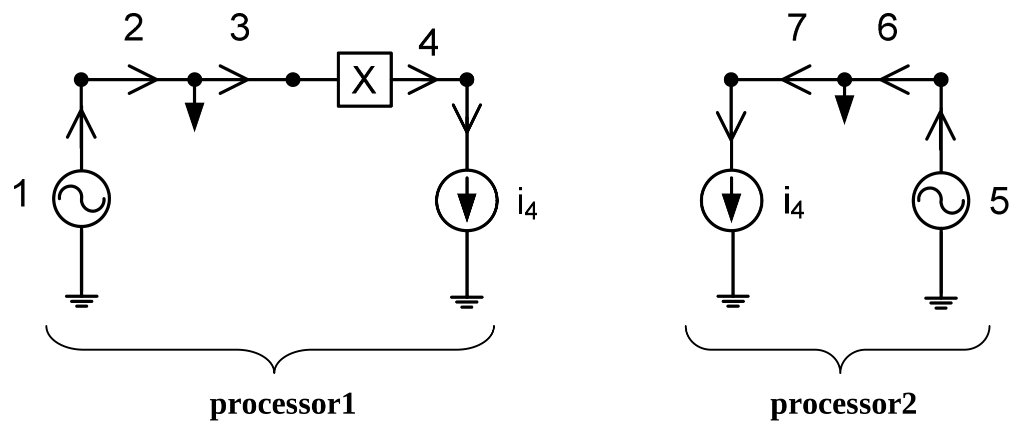

Figure 4.3 Model split across processors at cotree.

Li and Broadwater [7] present a distributed algorithm for looped load flow. Their environment consisted of eight machines connected in a 10BASE-T Ethernet LAN, and the test involved solution of a 40,000 element system. This configuration resulted in a speedup of about 6 times relative to the base case run on one processor. The test results described in their paper demonstrate the feasibility of distributing GTA analysis over multiple processors, specifically the generic algorithm framework presented in this dissertation.

Due to the difficulties encountered in producing computer chips with higher clock speeds, the trend has been for computer manufacturers to produce multi-core processors. Currently for applications to take full advantage of the multiple cores, they must be written using multiple threads of execution. Much of this can be addressed through the judicious use of multithreading in the core of the generic algorithm framework. This is outside the scope of the current dissertation; it is a potential field of further research.

Chapter 5
Application of the Generic Flow Algorithm

5.1 Single-loop Fluid System

A one-loop system is shown in Figure 5.1. In this system all pipes have a diameter of D = 10 inches (0.833 ft), the density of the water is $\rho = 62.428$ lb/ft^3, the kinematic viscosity is $\nu = 1.076\text{x}10^{-5}$ ft^2/s, and the pipe roughness is specified as $\varepsilon = 0.006$. The upper path of the loop is 163 feet long and the lower path is 65 feet. The discharge rate at the sink is specified to be 500 gal/min.

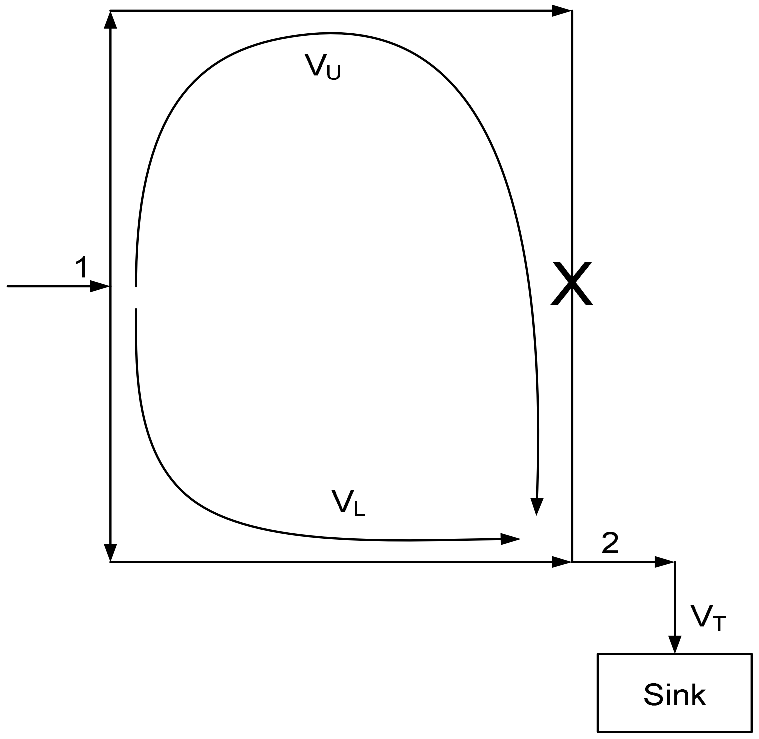

Figure 5.1 One loop system.

The mass flow rate at point 2 must equal the mass flow rate through the top section ($\rho V_U A$) plus the mass flow rate through the bottom section ($\rho V_L A$). This gives the relationship:

$$V_U = V_T - V_L \qquad (1)$$

The pressure difference across the loop is given by:

$$\Delta P = P_1 - P_2 = \frac{f_U L_U}{D} \frac{\rho V_U^{\ 2}}{2g} - \frac{f_L L_L}{D} \frac{\rho V_L^{\ 2}}{2g} \qquad (2)$$

where the friction loss is given by the Swamee-Jain [37] equation:

$$f = \frac{0.25}{\left(\log_{10}\left[\frac{\varepsilon}{3.7D} + \frac{5.74}{\mathrm{Re}^{0.9}} \right] \right)^2} \qquad (3)$$

The Reynolds number, Re, for circular pipes is given by:

$$\mathrm{Re} = \frac{\rho V D}{\mu} = \frac{VD}{\nu} \qquad (4)$$

where μ is the dynamic viscosity and ν is the kinematic viscosity, V is the fluid velocity and D is the pipe diameter.

The pressure drop from point 1 to point 2 is a single value and must be the same whether the path followed is along the top leg or the bottom leg. Therefore, equation (2) can be written as:

$$\frac{f_U L_U}{D} \frac{\rho V_U^{\ 2}}{2g} = \frac{f_L L_L}{D} \frac{\rho V_L^{\ 2}}{2g} \qquad (5)$$

Since the flow split between the upper and lower legs of the loop is not known a priori and the friction losses are dependent on the velocity, the solution requires iteration with an initial guess. If the initial flow split is assumed to be 50% through the upper leg and 50% through the lower leg, convergence is achieved after two iterations. The hand-calculated results are:

$w_L = \rho V_L A = 307.3$ gpm

and

$w_U = \rho V_U A = 192.7$ gpm

The results obtained using the algorithm described in this paper are within 0.8% of the analytical results.

5.2 Multi-loop System using GTA and EPANET

A multi-loop system is shown as modeled in EPANET (Figure 5.2) and GTA (Figure 5.3). It consists of one constant-head source, R-1 at 200 feet and another constant-head source, R-2 at 150 feet. All pipes have a 12” diameter with constant pipe roughness of 0.006 feet.

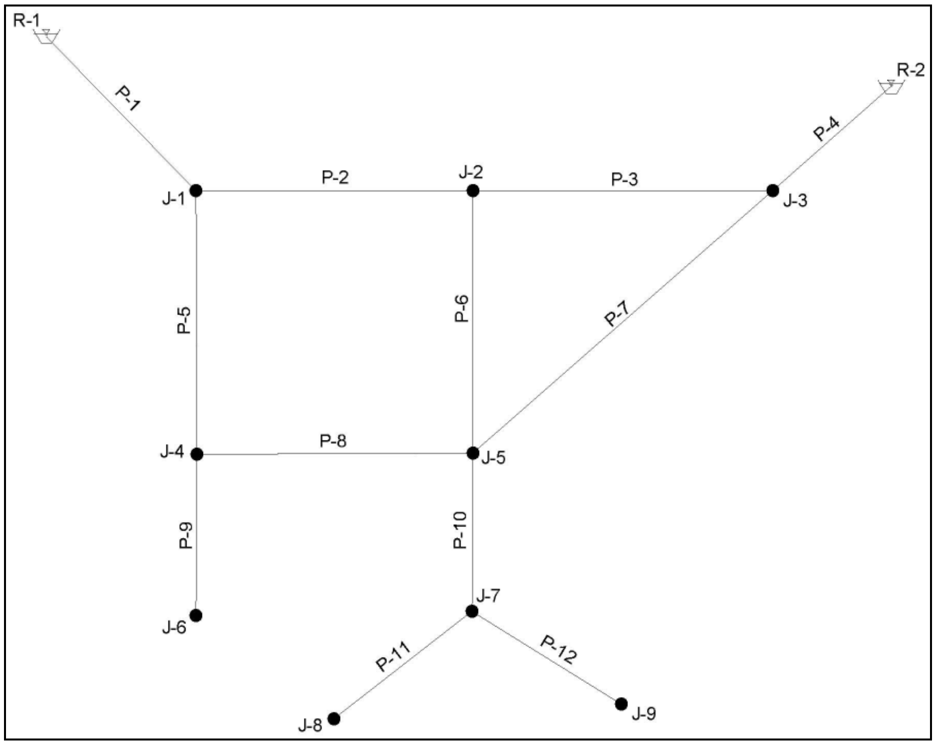

Figure 5.2 Example multi-loop EPANET test system.

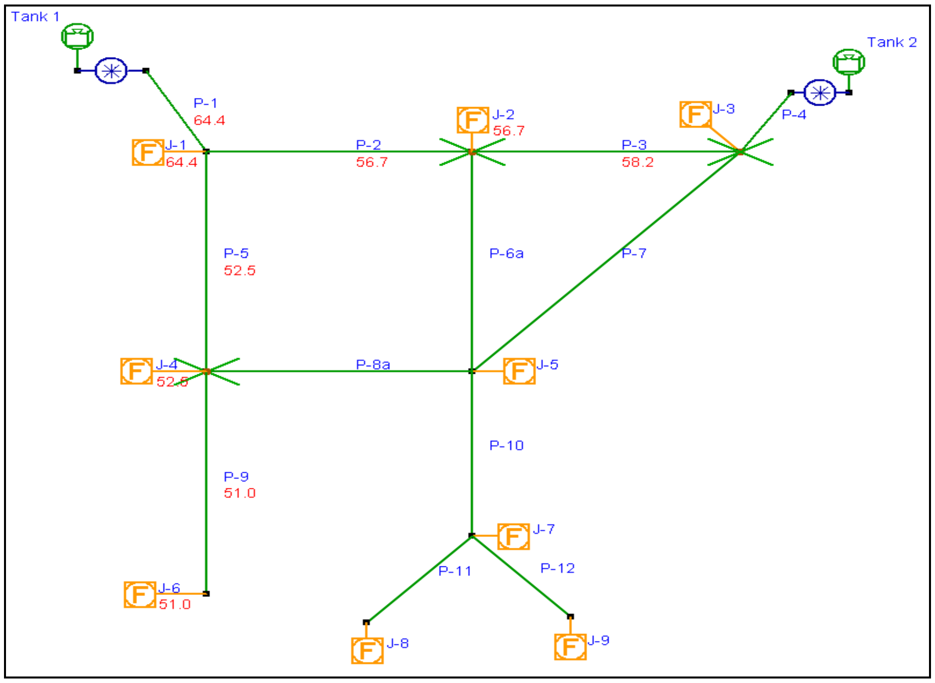

Figure 5.3 Example multi-loop GTA model test system.

The example system was compared with a manual solution using the Hardy-Cross iterative method, and with a reference model using EPANET [38]. All of the calculations used the Darcy-Weisbach equation with the Swamee-Jain approximation for obtaining the friction factor. Table 5.1 lists the pipe lengths and the demand loads at specified junctions.

Table 5.1 List of pipe lengths and demand loads.

Pipe ID	Pipe Length (Feet)	Junction ID	Junction Demand (GPM)
P-1	2000	J-1	-
P-2	3000	J-2	900
P-3	3000	J-3	-
P-4	2000		
P-5	4000	J-4	450
P-6	4000	J-5	900
P-7	5000		
P-8	3000		
P-9	4000	J-6	450
P-10	3000	J-7	300
P-11	2000	J-8	600
P-12	2000	J-9	300

Table 5.2 shows the pipe flows obtained from the three calculations, and Table 5.3 shows the junction pressures. As seen from the tables, all three calculations are in close agreement.

Table 5.2 Pipe flows for example network system.

	Hardy Cross	Reference Model	% Difference	GTA Result	% Difference
Pipe#	GPM	GPM	(Reference vs. HC)	GPM	(GTA vs. HC)
1	2514.1	2514.6	0.02%	2512.4	0.07%
2	1213	1213.3	0.02%	1211.9	0.09%
3	528.3	528	0.06%	529.5	0.22%
4	1385.9	1385.4	0.03%	1387.6	0.13%
5	1301.1	1301.3	0.02%	1300.5	0.05%
6	841.4	841.3	0.01%	841.4	0.01%
7	857.5	857.4	0.02%	858.1	0.06%
8	401.1	401.3	0.05%	400.5	0.15%
9	450	450	n/a	450	n/a
10	1200	1200	n/a	1200	n/a
11	600	600	n/a	600	n/a
12	300	300	n/a	300	n/a
	Average % Difference =		0.03%		0.10%

Table 5.3 Junction pressures for example system.

	Hardy Cross	Reference Model	% Difference	GTA Result	% Difference
Junction#	PSI	PSI	(Reference vs. HC)	PSI	(GTA vs. HC)
J-1	64.54	64.4	0.22%	64.4	0.22%
J-2	56.77	56.8	0.06%	56.7	0.11%
J-3	58.26	58.2	0.10%	58.1	0.27%
J-4	52.62	52.5	0.23%	52.5	0.23%
J-5	51.75	51.7	0.10%	51.6	0.30%
J-6	51.17	51.1	0.13%	51	0.33%
J-7	44.14	44.1	0.09%	44	0.32%
J-8	42.86	42.8	0.13%	42.7	0.37%
J-9	43.81	43.8	0.03%	43.7	0.26%
	Average % Difference =		0.12%		0.27%

A large system-of-systems model was solved which consisted of a water system and an electrical system. The water system contained 17,750 components and 180 independent loops. The electrical model contained 9,850 components and was operated radially.

The electrical and fluid components can each specify their own across and through behaviors. The system object is responsible for ensuring that the overall conservation equations are satisfied.

Chapter 6
Conclusions and Further Research

6.1 Conclusions

This dissertation presents a framework for solving interconnected systems of different types through the implementation of generic algorithms and the use of Graph Trace Analysis. This provides one common foundation for different system types. One advantage of using GTA is that it does not require large matrices.

Characteristics of topology iterators include:

- Topology iterators naturally support distributed processing.
- Only topology iterators that directly relate to a component affected by topology changes (such as component failure or sectionalizing device operation) need to be updated when a topology change occurs.
- Topology iterators provide for efficient use of memory, which is important in managing large models.

Chapter 4 presents a generic flow algorithm which can be applied to different system types. Chapter 5 illustrated its use in the solution of two example incompressible fluid systems and one example electrical system.

6.2 Contributions

The work described in this dissertation provides the following contributions to the field of Computer Engineering:

- It provides a generic, object-oriented framework in which the object is the engineering algorithm. System components are programmed to a standard interface, allowing for different domains to be solved using the same system solver.
- Generic algorithms can be packaged into dynamic program units (DLLs) to provide hot-swappable algorithms to the main program. The DLLs are run-time design artifacts whereas most existing design approaches employ compile-time artifacts. Using this new framework, algorithms can be discovered and replaced while the program continues to run without requiring changes to any other software. This is particularly important in distributed processing systems, where many analyses may be working concurrently with different parts of the system model over different processors.
- The polymorphic behavior of the across and through variables is implemented using containment rather than inheritance. Inheritance-based polymorphism has often proven to be rigid and difficult to adapt as new requirements are discovered. Implementing polymorphism with containment provides flexibility in the software framework for responding to emergent problems.

6.2 Further Research

This work presents one generic algorithm and a framework through which interconnected systems of different domains can be solved. This framework is flexible and can be further enhanced through a more complete modeling of dependencies between components and systems.

This dissertation presented a generic flow algorithm; based on this framework, many other generic algorithms can be formulated. For example, generic reconfiguration for restoration algorithms could be written to evaluate various scenarios for restoring critical services in response to component failures.

Another example generic algorithm is one for evaluating mission priorities for naval ships. The coordination of many interconnected systems is required to maintain seaworthiness of the ship while also providing sufficient resources to accomplish the overall mission. Priorities assigned to systems and subsystems could be propagated from logical loads, down through reconfigurable electric and fluid systems, to vital and auxiliary system service loads.

Further investigation into the use of this framework in distributed computing environments is also recommended. With the rapid declines in computer costs, array or blade computers are now cost-effective. The combination of generic algorithms that run over multi-domain networks with effective use of distributed processing can provide an environment in which large systems can be run in real- or near real-time.

References

[1] Alan Curtis Kay, The Reactive Engine, Doctoral Thesis, University of Utah, Salt Lake City, 1969.

[2] F. Zhang, C.S. Cheng, “A Modified Newton Method for Radial Distribution System Power Flow Analysis”, IEEE Transactions on Power Systems, Vol. 12, No. 1, February 1997, pp. 389-397.

[3] R.P. Broadwater, J.C. Thompson, T.E. McDermott, “Pointers and Linked Lists in Electric Power Distribution Circuit Analysis”, *Proceedings of 1991 IEEE PICA Conference*, Baltimore, Maryland, 1991, 16-21.

[4] Blackwell, William A., *Mathematical Modeling of Physical Networks*. New York: The MacMillan Company, 1968.

[5] Lynn R. Feinauer, Kevin J. Russell, Robert P Broadwater, “Graph Trace Analysis and Generic Algorithms for Interdependent Reconfigurable System Design and Control”, Naval Engineers Journal, Volume 120 Issue 1, 2008.

[6] Vadim V. Bulitki, David C. Wilkins, “Automated Instructor Assistant for Ship Damage Control”, in: *Proceedings of the 11th Innovative Applications of Artificial Intelligence ’99 conference*, Orlando, USA, July 1999, pp. 778-785.

[7] Fangxing Li, Robert P. Broadwater, “Distributed Algorithms with Theoretic Scalability Analysis of Radial and Looped Load Flows for Power Distribution Systems”, Electric Power Systems Research 65 (2003), pp. 169-177.

[8] K.L. Butler, N.D.R. Sarma, V. R. Prasad, “Network Reconfiguration for Service Restoration in Shipboard Power Distribution Systems”, IEEE Transactions on Power Systems, Vol. 16, No. 4, November 2001, pp. 653-661.

[9] E.L.Zivi, “Integrated Shipboard Power and Automation Control Challenge Problem”, IEEE Power Engineering Society Summer Meeting, 2002, Vol. 1, pp. 325-330.

[10] Lirong Wang, Jiacai Wang, Ichiro Hagiwara, “Modeling Approach of Functional Model for Multidomain System, JSME International Journal”, Series C, Vol. 48, No. 1, 2005, pp. 70-80.

[11] N.R. Weston, M.G. Balchanos, M.R. Koepp, D.N. Mavris, “Strategies for Integrating Models of Interdependent Subsystems of Complex System-of-System Products”, Proceedings of the 38th Southeastern Symposium on System Theory, Tennessee Technological University, Cookeville, Tennessee, March 2006, pp. 181-185.

[12] R.. Sinha, C.J.J. Paredis, V.C. Liang, P.K. Khosa, “Modeling and Simulation Methods for Design of Engineering Systems”, ASME Journal of Computing and Information Science in Engineering, vol. 1, March 2001, pp. 84-91.

[13] Kwa-Sur Tam, “Modeling Approaches for Large-Scale Reconfigurable Engineering Systems”, Transactions on Engineering, Computing and Technology, Volume 17, December 2006, pp. 135-140.

[14] K.S. Tam, R. Broadwater, "A Framework to Model Interdependent Engineering Systems", Proceedings of the 24th IASTED International Conference, MODELING, IDENTIFICATION, AND CONTROL, February, 2005, Innsbruck, Austria, pp. 558-563.

[15] Nils K. Svendsen, Stephen D. Wolthusen, "Analysis and Statistical Properties of Critical Infrastructure Interdependency Multiflow Models", Proceedings of the 2007 IEEE Workshop on Information Assurance, United States Military Academy, West Point, NY, June 2007, pp. 247-254.

[16] S.M. Rinaldi, "Modeling and Simulating Critical Infrastructures and Their Interdependencies", Proceedings of the 37th Hawaii International Conference on System Sciences, 2004.

[17] Edsger W. Dijkstra, "Notes on Structured Programming", T.H. Report 70-WSK-03, Technological University Eindhoven, The Netherlands, Department of Mathematics, April 1970.

[18] Alan C. Kay, "The Early History of Smalltalk", ACM SIGPLAN Notices, Volume 28, No. 3, March 1993, pp. 69-95.

[19] F. Li, R.P. Broadwater, "Software Framework Concepts for Power Distribution System Analysis", IEEE Transactions on Power Systems, Vol. 19, No. 2, May 2004, pp. 948-956.

[20] M. Bonfé, C. Fantuzzi, C. Secchi, "Object-Oriented Modeling of Multi-Domain Systems", Proceedings of the 2005 IEEE International Conference on Automation Science and Engineering, Edmonton, Canada, August 2005, pp. 363-368.

[21] Chen Yewang, Jiang Zhixiong, Zhao Wenyun, Peng Xin, "Generic Component: A Generic Programming Approach", Seventh International Conference on Computer and Information Technology, University of Aizu, Fukushima, Japan, Oct. 2007, pp. 87-92.

[22] Kirstie L. Bellman, Christopher Landauer, "Integration Science: More Than Putting Pieces Together", IEEE Aerospace Conference Proceedings, Vol. 4, 2000, pp. 397-409.

[23] Ondrej Ryasvy, Frantisek Scuglik, Miroslav Sveda, "Designing Algorithm-Oriented Generic Library on .NET Framework", Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems, 0-7695-2546-6/06, 2006.

[24] *Modelica^{TM} – A Unified Object-Oriented Language for Physical Systems Modeling*: *Tutorial, Version 1.4*, December 2000, Modelica Association.

[25] M. Chakrabarty, D. Mendonça, "Integrating Visual and Mathematical Models for the Management of Interdependent Critical Infrastructures", IEEE International Conference on Systems, Man and Cybernetics, 2004, pp. 1179-1184.

[26] P. Voigt, G. Wachutka, "Electro-fluidic Microsystem Modeling Based on Kirchhoffian Network Theory", IEEE International Conference on Solid-State Sensors and Actuators, Chicago, Illinois, June 1997, pp. 1019-1022.

[27] B. Bachelet, A. Mahul, L. Yon, "Designing Generic Algorithms for Operations Research", Université Blaise-Pascal, BPm Aubière, France, Research Report LIMOS/RR03-20, 2003.

[28] Jos B. Warmer, Anneke G. Kleppe, *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[29] Edsger Dijkstra, "Go To Statement Considered Harmful", Communications of the ACM, Volume 11, Number 3, March, 1968, pp. 147-148.

[30] Wai-Kai Chen, "Graph Theory and Its Engineering Applications", Advanced Series in Electrical and Computer Engineering: Volume 5, University of Illinois at Chicago, World Scientific Publishing Company, 1997.

[31] Lynn R. Feinauer, Molly E. Ison, Robert P. Broadwater, "Generic Algorithms for Analysis of Interdependent Multi-Domain Network Systems", International Journal of Critical Infrastructures (IJCIS), Vol. 6, No.1, pp. 81-95, 2010.

[32] R. Broadwater, J. Thompson, M. Ellis, H. Ng, N. Singh, D. Loyd, "Application Programmer Interface for the EPRI Distribution Engineering Workstation", IEEE Transactions on Power Systems, Vol. 10, No.1, February 1995, 499-505.

[33] P. Steven, R. Pooley, *Using UML: Software Engineering with Objects and Components*, Addison-Wesley, Reading, MA, 2006.

[34] J. Järvi, A. Lumsdaine, D.P. Gregor, M. Kulkarni, D.R. Musser, S. Schupp, "Generic Programming and High-Performance Libraries", IEEE Proceedings of the 18th International Parallel and Distributed Processing Symposium, April 26-30, 2004.

[35] H.M. Deitel, *C++ How To Program: Fifth Edition*, Prentice Hall, 2006.

[36] David L. Kleppinger, Kevin J. Russell, Robert P. Broadwater, "Graph Trace Analysis Based Shipboard HM&E System Priority Management and Recovery Analysis", IEEE Electric Ship Technologies Symposium, May 2007, pp. 109-114.

[37] P.K. Swamee, A.K. Jain, "Explicit Equations for Pipe-Flow Problems", Journal of the Hydraulics Division, ASCE 102(5), pp. 657-664, 1976.

[38] L. Rossman, *EPANET 2: Technical Report EPA/600/R-00/057*, Washington, D.C., 2000.

Appendix: An Implementation of Polymorphism using Dynamic Containment

This Appendix presents an implementation of polymorphism using dynamic containment as defined in the body of this dissertation. The implementation shown is written in C++, and is specifically targeted at the Windows XP and Vista operating systems. At startup, the program initializes the CmpExt (extended component) module. If no configuration file is found, all of the “across”, “through” and “dlg” functions are defaulted to the normal component behavior.

Example Configuration File (CmpExt.cfg)

```
<?xml version="1.0"?>
<cmpext>
    <!—
    ****        Fluid Component definitions          ****
    ****                                             ****
    **** Note that component across, through and  ****
    **** dialog functions can be in different DLLs    ****
    -->
    <dll name="CmpExtFluid.dll">
        <!—
        **** Component “across” function definitions  ****
        -->
        <across name="CentPumpAcross" cmp="158" />
        <across name="CheckValveAcross" cmp="155" />
        <across name="ChillWaterPipeAcross" cmp="206" />
        <across name="FlowInjectorAcross" cmp="214" />
        <across name="FluidCircuitAcross" cmp="162" />
        <across name="FluidLoadAcross" cmp="181" />
        <across name="TankAcross" cmp="200" />
        <!—
        **** Component “through” function definitions  ****
        -->
        <through name="CentPumpThrough" cmp="158" />
        <through name="CheckValveThrough" cmp="155" />
        <through name="ChillWaterPipeThrough" cmp="206" />
        <through name="FlowInjectorThrough" cmp="214" />
        <through name="FluidCircuitThrough" cmp="162" />
        <through name="FluidLoadThrough" cmp="181" />
        <through name="TankThrough" cmp="200" />
        <!—
        **** Component “dialog” function definitions  ****
        -->
        <dialog name="CentPumpDialog" cmp="158" />
        <dialog name="CheckValveDialog" cmp="155" />
        <dialog name="ChillWaterPipeDialog" cmp="206" />
        <dialog name="FlowInjectorDialog" cmp="214" />
        <dialog name="FluidCircuitDialog" cmp="162" />
        <dialog name="FluidLoadDialog" cmp="181" />
        <dialog name="TankDialog" cmp="200" />
    </dll>
    <!—
    **** Electrical Component definitions ****
    -->
    <dll name="CmpExtElectrical.dll">
        <!—
        **** Component “across” function definitions  ****
        -->
        < across name="CableLineAcross" cmp="128" />
        < across name="DistTransformerAcross" cmp="20" />
        < across name="FuseCutoutAcross" cmp="182" />
        < across name="RecloserAcross" cmp="9" />
        < across name="SectionalizerAcross" cmp="12" />
        < across name="SubstationAcross" cmp="0" />
        < across name="SwitchAcross" cmp="183" />
        < across name="VoltageRegulatorAcross" cmp="22" />
        <!—
        **** Component “through” function definitions  ****
        -->
        <through name="CableLineThrough" cmp="128" />
        <through name="DistTransformerThrough" cmp="20" />
        <through name="FuseCutoutThrough" cmp="182" />
        <through name="RecloserThrough" cmp="9" />
        <through name="SectionalizerThrough" cmp="12" />
        <through name="SubstationThrough" cmp="0" />
        <through name="SwitchThrough" cmp="183" />
        <through name="VoltageRegulatorThrough" cmp="22" />
        <!—
        **** Component “dialog” function definitions  ****
        -->
        <dialog name="CableLineDialog" cmp="128" />
        <dialog name="DistTransformerDialog" cmp="20" />
        <dialog name="FuseCutoutDialog" cmp="182" />
        <dialog name="RecloserDialog" cmp="9" />
        <dialog name="SectionalizerDialog" cmp="12" />
        <dialog name="SubstationDialog" cmp="0" />
        <dialog name="SwitchDialog" cmp="183" />
        <dialog name="VoltageRegulatorDialog" cmp="22" />
    </dll>
</cmpext>
```

Header File with Component IDs (CmpIDs.h)

```
#pragma once
#define FLD_ START                      10000
#define FLD_CENTPUMP                    FLD_START
#define FLD_CHECKVALVE                  FLD_START + 10
#define FLD_CHILLWATERPIPE              FLD_START + 20
#define FLD_FLOWINJECTOR                FLD_START + 30
#define FLD_FLUIDCIRCUIT                FLD_START + 40
#define FLD_FLUIDLOAD                   FLD_START + 50
#define FLD_TYPE_TANK                   FLD_START + 60
```

```
#define ELEC_START                          20000
#define ELEC_CABLELINE                      ELEC_START
#define ELEC_DISTTRANSFORMER                ELEC_START + 10
#define ELEC_FUSECUTOUT                     ELEC_START + 20
#define ELEC_RECLOSER                       ELEC_START + 30
#define ELEC_SECTIONALIZER                  ELEC_START + 40
#define ELEC_SUBSTATION                     ELEC_START + 50
#define ELEC_SWITCH                         ELEC_START + 60
#define ELEC_VOLTAGEREGULATOR               ELEC_START + 70
```

Main Program

```
static bool bInitCmpExt = false;
…

if ( !bInitCmpExt )
{
    if ( cmpExtInitDLL( ) != 1)
    {
        return false;
    }
}
```

CmpExt.h

```
#pragma once

struct Cmp
{
    Cmp();
    ~Cmp();
    int iTra;              // Trace index for component
    int iCkt;              // Trace index for circuit
    int iSub;              // Trace index for substation
    int iSys;              // Trace index for system
    int iCmp;              // Index for row in Component table
    int iPtRow;            // Index for row in Parts table
    int iNumPh;            // Number of phases present
    int iYr;               // Year component was installed
    int iYrManufact;       // Year of manufacture
    int iEnDeg;            // End node degree of component
    char szMapZon[4];      // Map zone identifier
    int aiXy[2];           // xy workstation coordinates
    int iOldCkt;           // Saved circuit trace index
    int iOldTra;           // Saved component trace index
    BOOL fSectDev;         // 0 => open sectionalizing component
                           // 1 => closed sectionalizing component
    int bCmpTyp;           // Component Type
    int asPh[3];           // Logic for phases present
                           // asPh[0] =  0   => phase A present
                           //            1   => phase B, but not A
                           //            2   => phase C, not A or B
                           // asPh[1] =  1   => phases A and B
                           //            2   => phases C and A or B
                           //           -1   => single phase present
                           //           -1   => 3 phases not present
                           // asPh[2] =  2   => phases A, B, and C
    int bPh;             // 1 => phase A
                           // 2 => phase B
                           // 3 => phase AB
                           // 4 => phase C
                           // 5 => phase AC
                           // 6 => phase BC
                           // 7 => phase ABC
    char chTyp;          // Generic Component Type
                           // 'e' electrical
                           // 'f' fluid
                           // 'g' gas
    int iSBX;            // X Starting Screen Coord
    int iSBY;            // Y Starting Screen Coord
    int iSEX;            // X Ending Screen Coord
    int iSEY;            // Y Ending Screen Coord
    int iAdjCkt;         // Trace index for adjacent circuit
    int iAdjTra;         // Trace index for adjacent component
    int iAdjCtConTra;     // Trace index for cotree component
    int iBTra;           // Trace index for backware component
    int iBrTra;          // Trace index for brother component
    int iFTra;           // Trace index for forward component
    int iFpTra;          // Trace index for feeder path component
    double adAcross_Im[3];  // Across real values for 3 phases
    double adAcross_Re[3];  // Across imaginary values for 3 phases
    double adThrough_Im[3]; // Through real values for 3 phases
    double adThrough_Re[3]; // Through imaginary values for 3 phases
    struct Cmp* pAdjCmp;    // Pointer to adjacent component
    struct Cmp* pB;       // Pointer to backward component
    struct Cmp* pBr;      // Pointer to brother component
    struct Cmp* pCtCmp;    // Pointer to cotree component
    struct Cmp* pF;       // Pointer to forward component
    struct Cmo* pFp;      // Pointer to feeder path component
    ...
} CMP, *PCMP;

typedef int (*DIALOG_FCN)(HWND hwnd, struct Cmp*);
typedef int (*ACROSS_FCN)(struct Cmp*);
typedef int (*THROUGH_FCN)(struct Cmp*);
typedef BOOL (*CMPEXT_INITDLL)();

typedef struct cmpext {
  DIALOG_FCN lpfnDialog;
  ACROSS_FCN lpfnAcross;
  THROUGH_FCN lpfnThrough;
} CMPEXT, *PCMPEXT;

typedef map<int, CMPEXT*> CMPEXTMAPTYPE;

int dlg(HWND hwnd, PCMP pCmp);
int across(PCMP pCmp);
int through(PCMP pCmp);

// Functions exported by the CmpExt
extern "C" DllExport int cmpExtDlg(HWND hwnd, PCMP pCmp);
extern "C" DllExport BOOL cmpExtInitDLL();
extern "C" DllExport int cmpExtAcross(PCMP pCmp);
extern "C" DllExport int cmpExtThrough(PCMP pCmp);

// Classes used internally by CCmpExt
class CLine
{
  public:
    CLine(string& strLine);
    BOOL GetInt(int& iVal);
    BOOL GetString(string& strVal);

  private:
    string m_strLine;
    int m_endPos;
    int m_curPos;
};

class CCmpExt
{
  public:
    CCmpExt();
    ~CCmpExt();
    int DoCmpExtDlg(HWND hwnd, PCMP pCmp);
    int DoCmpExtAcross(PCMP pCmp);
    int DoCmpExtThrough(PCMP pCmp);
    BOOL ReadConfigFile();
    bool ProcessDLLNode(CComPtr<IXmlReader> pReader,
      string& strPath);
    bool ProcessFunctionNode(CComPtr<IXmlReader> pReader, int type,
      string& strPath);

  private:
    CMPEXTMAPTYPE m_cmpExtMap;
    DLG_FCN m_lpfnDlgDefault;
    ACROSS_FCN m_lpfnAcrossDefault;
    THROUGH_FCN m_lpfnThroughDefault;
    string m_strConfigFile;
    string m_strAcrossDefault;
    string m_strDlgDefault;
    string m_strThroughDefault;
};
```

CmpExt.cpp

```
#include "stdafx.h"
#include "CmpExt.h"
#include "FileStream.h"

HINSTANCE ghInstCmpExt=NULL;
CCmpExt* gpCmpExt=NULL;
string gstrExecDir="";

BOOL APIENTRY DllMain( HANDLE hModule, DWORD dwReasonForCall,
     LPVOID lpReserved)
{
     char szFileName[_MAX_PATH];
     char szDrive[_MAX_DRIVE];
     char szDir[_MAX_DIR];
```

```
    char szExecDir[_MAX_PATH];
    switch (dwReasonForCall)
     {
        case DLL_PROCESS_ATTACH:
            ghInstCmpExt = reinterpret_cast<HINSTANCE>(hModule);
            GetModuleFileName(NULL, (LPTSTR)szFileName, _MAX_PATH);
            _splitpath_s(szFileName, szDrive, _MAX_DRIVE, szDir, _MAX_DIR,
                NULL, 0, NULL, 0);
            _makepath_s(szExecDir, _MAX_PATH, szDrive, szDir, NULL, NULL);
            gstrExecDir = szExecDir;
            break;

        case DLL_PROCESS_DETACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
        break;
    }
    return TRUE;
}

int DllExport cmpExtDlg(HWND hwnd, PCMP pCmp)
{
    if (gpCmpExt == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NO_CMPEXT_INST, "");
        return 1;
    }
    return (gpCmpExt->DoCmpExtDlg(hwnd, pCmp));
}

BOOL DllExport cmpExtInitDLL()
{
    gpCmpExt = new CCmpExt();
    if (gpCmpExt == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_CANNOT_INIT_CMPEXT, "");
        return FALSE;
    }

    BOOL bStat = gpCmpExt->ReadConfigFile();
    if (!bStat)
    {
        delete gpCmpExt;
        gpCmpExt = NULL;
    }
    return bStat;
}

int DllExport cmpExtAcross(PCMP pCmp)
{
    if (gpCmpExt == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NO_CMPEXT_INST, "");
        return 1;
    }
    return (gpCmpExt->DoCmpExtAcross(pCmp));
}

int DllExport cmpExtThrough(PCMP pCmp)
{
    if (pCmp == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_PCMP, "");
        return 1;
    }

    if (gpCmpExt == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NO_CMPEXT_INST, "");
        return 1;
    }
    return (gpCmpExt->DoCmpExtThrough(pCmp));
}

int dlg(HWND hwnd, PCMP pCmp)
{
    DoMainDialog(ghMainWnd, pCmp);
    return 0;
}

int across(PCMP pCmp)
{
    int iPhase;
    PCMP pFp = pCmp->pFp;
    if (pFp)
    {
        for (iPhase = 0; iPhase < MAX_PHASES; iPhase++)
        {
            pCmp->adAcross_re[iPhase] = pFp->adAcross_re[iPhase];
            pCmp->adAcross_re[iPhase] = pFp->adAcross_im[iPhase];
            pCmp->adThrough_re[iPhase] = pFp->adThrough_re[iPhase];
            pCmp->adThrough_im[iPhase] = pFp->adThrough_im[iPhase];
        }
    }
    return 0;
}

int through(PCMP pCmp)
{
    return 0;
}

CCmpExt::CCmpExt( )
{
    m_strConfigFile = "CmpExt.cfg";
    m_strAcrossDefault = "across";
    m_strThroughDefault = "through";
    m_strDlgDefault = "dlg";
}

CCmpExt::~CCmpExt()
{
    gpCmpExt = NULL;
}

int CCmpExt::DoCmpExtDlg(HWND hwnd, PCMP pCmp)
{
    if (pCmp == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_PCMP, "");
        return 1;
    }

    PCMPEXT pCmpExt = m_cmpExtMap[pCmp->iCmp];
    if (pCmpExt == NULL)
    {
        // Use the default implementation
        return (m_lpfnDlgDefault(hwnd, pCmp));
    }

    if (pCmpExt->lpfnDlg == NULL)
    {
        // Use the default implementation
        return (m_lpfnDlgDefault(hwnd, pCmp));
    }
    return (pCmpExt->lpfnDlg(hwnd, pCmp));
}

int CCmpExt::DoCmpExtAcross(PCMP pCmp)
{
    if (pCmp == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_PCMP, "");
        return 1;
    }

    PCMPEXT pCmpExt = m_cmpExtMap[pCmp->iCmp];
    if (pCmpExt == NULL)
    {
        // Use the default implementation
        return (m_lpfnAcrossDefault(pCmp));
    }

    if (pCmpExt->lpfnAcross == NULL)
    {
        // Use the default implementation
        return (m_lpfnAcrossDefault(pCmp));
    }
    return (pCmpExt->lpfnAcross(pCmp));
}
```

```
int CCmpExt::DoCmpExtThrough(PCMP pCmp)
{
    if (pCmp == NULL)
    {
        // Log error
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_PCMP, "");
        return 1;
    }

    PCMPEXT pCmpExt = m_cmpExtMap[pCmp->iCmp];
    if (pCmpExt == NULL)
    {
        // Use the default implementation
        return (m_lpfnThroughDefault(pCmp));
    }

    if (pCmpExt->lpfnThrough == NULL)
    {
        // Use the default implementation
        return (m_lpfnThroughDefault(pCmp));
  }

  return (pCmpExt->lpfnThrough(pCmp));
}

BOOL CCmpExt::ReadConfigFile()
{
    // Set the default behaviors for components that don't have an
    // extension declaration in the CmpExt.cfg file.
    m_lpfnAcrossDefault = across;
    m_lpfnThroughDefault = through;
    m_lpfnDlgDefault = dlg;

    HRESULT hr;
    CComPtr<IStream> pFileStream;
    CComPtr<IXmlReader> pReader;
    CComPtr<IXmlReaderInput> pReaderInput;

    // Open the configuration file. This file is named 'CmpExt.cfg'
    // and must be in the same directory as the main executable.
    if (FAILED(hr = FileStream::OpenFile(L"CmpExt.cfg", &pFileStream, FALSE)))
    {
        // This does not indicate an error. It simply means that the user
        // does not have a configuration file with extended component
        // information defined.
        return TRUE;
    }

    if (FAILED(hr = CreateXmlReader(__uuidof(IXmlReader), (void**)&pReader, NULL)))
    {
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_READCONFIG_RDERR, "");
        return FALSE;
    }

    pReader->SetProperty(XmlReaderProperty_DtdProcessing, DtdProcessing_Prohibit);
    if (FAILED(hr = CreateXmlReaderInputWithEncodingCodePage(pFileStream, NULL, 65001,
        FALSE, L"c:", &pReaderInput)))
    {
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_READER_SETPROPERTY, "");
        return FALSE;
    }

    if (FAILED(hr = pReader->SetInput(pReaderInput)))
    {
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_READER_SETINPUT, "");
        return FALSE;
    }

    XmlNodeType nodeType;
    const WCHAR* pQName;
    wchar_t szDLL[] = L"dll";
    wchar_t szDialog[] = L"dialog";
    wchar_t szAcross[] = L"across";
    wchar_t szThrough[] = L"through";
    string strDLL;

    // Read until there are no more nodes
    while (S_OK == (hr = pReader->Read(&nodeType)))
    {
        switch (nodeType)
        {
            case XmlNodeType_Element:
                if (FAILED(hr = pReader->GetQualifiedName(&pQName, NULL)))
                {
                    LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_GETNAME, "");
                    return FALSE;
                }

                if (wcscmp(szDLL, pQName) == 0)
                {
                    if (!ProcessDLLNode(pReader, strDLL))
                    {
                        return FALSE;
                    }
                }
                else if (wcscmp(szDialog, pQName) == 0)
                {
                    if (!ProcessFunctionNode(pReader, NODE_TYPE_DIALOG, strDLL))
                    {
                        return FALSE;
                    }
                }
                else if (wcscmp(szAcross, pQName) == 0)
                {
                    if (!ProcessFunctionNode(pReader, NODE_TYPE_ACROSS, strDLL))
                    {
                        return FALSE;
                    }
                }
                else if (wcscmp(szThrough, pQName) == 0)
                {
                    if (!ProcessFunctionNode(pReader, NODE_TYPE_THROUGH, strDLL))
```

```
                    {
                        return FALSE;
                    }
                }
                break;

            case XmlNodeType_Text:
                break;

            case XmlNodeType_EndElement:
                break;

            default:
                break;
        }
    }
    return TRUE;
}

bool CCmpExt::ProcessDLLNode(CComPtr<IXmlReader> pReader, string& strDLL)
{
    HRESULT hr;
    const WCHAR* pName;
    wchar_t szName[] = L"name";

    if (FAILED(hr = pReader->MoveToAttributeByName(szName, NULL)))
    {
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_DLLCMPATTR, "");
        return false;
    }

    if (FAILED(hr = pReader->GetValue(&pName, NULL)))
    {
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_DLLCMPVALUE, "");
        return false;
    }

    // Convert the file name from wide char to multi-byte
    size_t i;
    char* pBuffer = new char[BUFFER_SIZE];
    if (pBuffer == NULL)
    {
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_MEM, "");
        return false;
    }

    wcstombs_s(&i, pBuffer, (size_t)BUFFER_SIZE, pName, (size_t)BUFFER_SIZE);
    strDLL = pBuffer;
    delete pBuffer;
    return true;
}

bool CCmpExt::ProcessFunctionNode(CComPtr<IXmlReader> pReader, int type, string& strDLL)
{
    HRESULT hr;
    const WCHAR* pName;
const WCHAR* pCmp;
wchar_t szName[] = L"name";
wchar_t szCmp[] = L"cmp";

if (FAILED(hr = pReader->MoveToAttributeByName(szName, NULL)))
{
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_FCNNAMEATTR, "");
    return false;
}

if (FAILED(hr = pReader->GetValue(&pName, NULL)))
{
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_FCNNAMEVALUE, "");
    return false;
}

if (FAILED(hr = pReader->MoveToAttributeByName(szCmp, NULL)))
{
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_FCNCMPATTR, "");
    return false;
}

if (FAILED(hr = pReader->GetValue(&pCmp, NULL)))
{
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_XMLREAD_FCNCMPVALUE, "");
    return false;
}

// Convert the function name and the cmpID from wide char to multi-byte
size_t i;
string strName;
string strCmp;
char* pBuffer = new char[BUFFER_SIZE];
if (pBuffer == NULL)
{
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_MEM, "");
    return false;
}

wcstombs_s(&i, pBuffer, (size_t)BUFFER_SIZE, pName, (size_t)BUFFER_SIZE);
strName = pBuffer;
delete pBuffer;

pBuffer = new char[BUFFER_SIZE];
if (pBuffer == NULL)
{
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_MEM, "");
    return false;
}

wcstombs_s(&i, pBuffer, (size_t)BUFFER_SIZE, pCmp, (size_t)BUFFER_SIZE);
strCmp = pBuffer;
delete pBuffer;

// Insert the function entry into the map
ACROSS_FCN lpfnAcross;
```

```
THROUGH_FCN lpfnThrough;
DLG_FCN lpfnDlg;
int cmpID = atoi(strCmp.c_str());
CMPEXT* pItem = m_cmpExtMap[cmpID];
if (pItem == NULL)
{
    pItem = new CMPEXT;
    if (pItem == NULL)
    {
        // Log an error and bail out - if we can't allocate memory
        // there is a real problem
        LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_MEM, "");
        return false;
    }
    m_cmpExtMap[cmpID] = pItem;
    pItem->lpfnAcross = across;
    pItem->lpfnThrough = through;
    pItem->lpfnDlg = dlg;
}

HMODULE hLibrary = LoadLibrary((LPCTSTR)strDLL.c_str());
if (hLibrary == NULL)
{
    // Log the error and continue with the next entry
    LogMsg(ERR_MSG, ghInstCmpExt, ERR_LOADLIB_FAILURE, "t",
        strDLL.c_str());
    return true;
}

switch (type)
{
    case NODE_TYPE_ACROSS:
        if (strName.size() > 0)
        {
            lpfnAcross = (ACROSS_FCN)GetProcAddress(hLibrary,
                strName.c_str());
            if (lpfnAcross == NULL)
            {
                // Log the error and use the default implementation
                LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_ACROSS_ENTRYPT,
                    "tt", strName.c_str(), strDLL.c_str());
                lpfnAcross = across;
            }
        }
        pItem->lpfnAcross = lpfnAcross;
        break;

    case NODE_TYPE_THROUGH:
        if (strName.size() > 0)
        {
            lpfnThrough = (THROUGH_FCN)GetProcAddress(hLibrary,
                strName.c_str());
            if (lpfnThrough == NULL)
            {
                // Log the error and use the default implementation
                LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_THROUGH_ENTRYPT,
                        "tt", strName.c_str(), strDLL.c_str());
                    lpfnThrough = through;
                }
            }
            pItem->lpfnThrough = lpfnThrough;
            break;

        case NODE_TYPE_DIALOG:
            if (strName.size() > 0)
            {
                lpfnDlg = (DLG_FCN)GetProcAddress(hLibrary, strName.c_str());
                if (lpfnDlg == NULL)
                {
                    // Log the error and use the default implementation
                    LogMsg(ERR_MSG, ghInstCmpExt, ERR_NULL_DLG_ENTRYPT,
                        "tt", strName.c_str(), strDLL.c_str());
                    lpfnDlg = dlg;
                }
            }
            pItem->lpfnDlg = lpfnDlg;
            break;
    }
    return true;
}
```

CmpExtFluid.cpp

```
#include "stdafx.h"
#include "Dialogs.h"

int iAppId=APPID_FLUID_FLOW_ANALYSIS;
static double dGrav=32.17;                  // Acceleration due to gravity
double ddiam, ddiam1, ddiam2;   // Internal diameter of pipe
double dArea, dArea1, dArea2;   // Internal Area
static double dRho=62.428;                  // Density of water in lb/ft^3
static double dAreaConv=144.0; // Conversion factor in^2/ft^2
static double dFt3OverM3=35.3134;       // Conversion factor ft^3/m^3
static double dPiFour=PI*.25;     // PI / 4
static double dKinVis=0.00001076;       // Kinematic viscosity of water in ft^2/s
static double dInToFt=0.08333;  // Conversion factor ft/in
static double dAtm=14.7;                    // Atmospheric pressure psia
static double dHeadToPsi=2.31;            // Feet of head per psi for water
static double epsilon=0.00015;   // Darcy Weisbach pipe roughness coefficient
static double dN=0.011;             // Manning pipe roughness coefficient
static double dGpmToFps=0.13911;        // Conversion from gal/min to lb/s for water
static double dCfManning=4.66;  // Manning conversion factor to English units
static double dCfHazenWilliams=4.73; // Hazen Williams conv factor to English units
BOOL bPressUpdByMeasDlg=FALSE;

BOOL APIENTRY DllMain(HANDLE hModule, DWORD dwReasonForCall,
    LPVOID lpReserved)
{
    switch (dwReasonForCall)
    {
        case DLL_PROCESS_ATTACH:
            hModCmp = (HINSTANCE)hModule;
            break;
        case DLL_PROCESS_DETACH:
        case DLL_THREAD_ATTACH:
        case DLL_THREAD_DETACH:
            break;
    }
    return TRUE;
 }

int DllExport CentPumpDialog(HWND hwnd, PCMP pCmp)
{
    part=pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_PUMPDLG),
        hwnd, PumpDlg);
    return 0;
}

int DllExport CentPumpAcross(PCMP pCmp)
{
    // Example centrifugal pump will maintain a specified pressure at its
    // outlet. Head curves should be used  to more closely model the
    // physics of the pump.
    double dPressure;
    BOOL bMeas;
    if (pCmp->pOper)
    {
        if (pCmp->pOper->fLostPrw == 1)
        {
            PCMP pFp = pCmp->pFp;
            if (pFp)
            {
                pCmp->adAcross_re[0] = pFp->adAcrosse_re[0];
            }
            return 0;
        }
    }

    if ((pCmp->fRevAmps == FALSE &&
        pCmp->adThrough_re[0] > 0.0)||(pCmp->adThrough_re[0] == 0))
    {
        bMeas = GetMeasurement(pCmp, BOUNDARY_PRESSURE, &dPressure);
    }
    else if (pCmp->fRevAmps == TRUE && pCmp->adThrough_re[0] < 0.0)
    {
        dPressure = pCmp->pF->adAcross_re[0];
        bMeas = TRUE;
    }
    else
    {
        SwitchCmp(&pCmp, FALSE);
        pCmp->pOper->fLostPrw=TRUE;
        dPressure = pCmp->pFp->adAcross_re[0];
        bMeas = TRUE;
    }

    if (!bMeas)
    {
        // This may be the case if the dialog box for the pump has never been
        // opened. Take the normal operating value from the database table
        if (gapPtFldPump && gapPtFldPump[pCmp->iPtRow])
        {
            dPressure = gapPtFldPump[pCmp->iPtRow]->dNormOpHead / dHeadToPsi;
        }
    }
    pCmp->adAcross_re[0] = dPressure;
    return 0;
}

int DllExport CentPumpThrough(PCMP pCmp)
{
    PCMP pFp = pCmp->pFp;
    if (pFp == NULL)
    {
        return 0;
    }

    // Set the flow rate through the pump to be the inlet flow rate
    pCmp->adThrough_re[0] = pFp->adAcross_re[0];
    return 0;
}

int DllExport CheckValveDialog(HWND hwnd, PCMP pCmp)
{
    part=pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_VALVEDLG),
        hwnd, ValveDlg);
    return 0;
}

int DllExport CheckValveAcross(PCMP pCmp)
{
    // If the valve is shut (fSect==OPEN), there is no flow or pressure drop
    int iRet=0;
    if (pCmp->fSectDev == CLOSED)
    {
        iRet = ValveDeltaP(pCmp);
    }
    else
    {
        PCMP pFp = pCmp->pFp;
        if (pFp)
        {
            pCmp->adAcross_re[0] = pFp->adAcross_re[0];
        }
    }
    return iRet;
}

int DllExport CheckValveThrough(PCMP pCmp)
{
    int iRet=0;
    if (pCmp->fSectDev == CLOSED)
    {
        iRet = ValveFlowRate(pCmp);
    }
    else
    {
        pCmp->adThrough_re[0] = 0.0;
    }
    return iRet;
}

int DllExport ChillWaterPipeDialog(HWND hwnd, PCMP pCmp)
{
    part = pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_PIPEDLG), hwnd,
        PipeDlg);
    return 0;
}

int DllExport ChillWaterPipeAcross(PCMP pCmp)
{
    int iRet = DeltaP(pCmp);
    return iRet;
}

int DllExport ChillWaterPipeThrough(PCMP pCmp)
{
    int iRet = MassFlowRate(pCmp);
    return iRet;
}

int DllExport FlowInjectorDialog(HWND hwnd, PCMP pCmp)
{
    part = pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_FLOWINJECTOR),
        hwnd, InjectorDlg);
    return 0;
}

int DllExport FlowInjectorAcross(PCMP pCmp)
{
    // The fluid injector specifies the mass flow rate in lb/s.
    double dFlowRate;
    if (GetMeasurement(pCmp, BOUNDARY_FLOWRATE, &dFlowRate))
    {
        pCmp->adThrough_re[0] = -dFlowRate;
        DeltaP(pCmp);
    }
    return 0;
}

int DllExport FlowInjectorThrough(PCMP pCmp)
{
    return 0;
}

int DllExport FluidCircuitDialog(HWND hwnd, PCMP pCmp)
```

```
{
    part = pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_FLUIDCIRCUITDLG),
        hwnd, FluidCircuitDlg);
    return 0;
}

int DllExport FluidCircuitAcross(PCMP pCmp)
{
    // Get the Pressure for the fluid circuit from the Source component.
    // Find it by doing a feeder path trace.
    PCMP pFp = pCmp->pFp;
    while (pFp != NULL)
    {
        if (pFp->bCmpTyp == CMPTYPE_SUB)
        {
            cmpExtAcross(pFp);
            pCmp->adAcross_re[0] = pFp->adAcross_re[0];
            break;
        }
        pFp = pFp->pFp;
    }
    return 0;
}

int DllExport FluidCircuitThrough(PCMP pCmp)
{
    int iRet = MassFlowRate(pCmp);
    return iRet;
}

int DllExport FluidLoadDialog(HWND hwnd, PCMP pCmp)
{
    part = pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_FLUIDLOADDLG),
        hwnd, FluidLoadDlg);
    return 0;
}

int DllExport FluidLoadAcross(PCMP pCmp)
{
    // The fluid load specifies the mass flow rate in lb/s.
    double dFlowRate;
    double dLoadScale = goApiFldWorkDef.dLoadScaleFactor;
    if (GetMeasurement(pCmp, BOUNDARY_FLOWRATE, &dFlowRate))
    {
        pCmp->adThrough_re[0] = dFlowRate * dLoadScale;
        DeltaP(pCmp);
    }
    return 0;
}

int DllExport FluidLoadThrough(PCMP pCmp)
{
    return 0;
}
```

```
int DllExport TankDialog(HWND hwnd, PCMP pCmp)
{
    part = pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_TANKDLG),
        hwnd, TankDlg);
    return 0;
}

int DllExport TankAcross(PCMP pCmp)
{
    // The pressure for the tank suction component is specified on the dialog box.
    // If it has not been changed through the dialog box, set it to atmospheric
    // pressure (14.7 psi)
    double dLevel;
    GetMeasurement(pCmp, BOUNDARY_PRESSURE, &dLevel);
    pCmp->adAcross_re[0] = dLevel;
    return 0;
}

int DllExport TankThrough(PCMP pCmp)
{
    int iRet = MassFlowRate(pCmp);
    return iRet;
}

int DllExport PressureSourceDialog(HWND hwnd, PCMP pCmp)
{
    part = pCmp;
    DialogBox(hModCmp, MAKEINTRESOURCE(IDD_PSOURCEDLG),
        hwnd, PressureSourceDlg);
    return 0;
}

int DeltaP(PCMP pCmp)
{
    // Calculate Delta Pressure across pipes using the Bernoulli
    // equation for a streamline along the flow path.
    double mdot;
    double P1;
    double P2;
    double z1;
    double z2;
    double v;
    double dElevationTerm;
    double Re;
    double f = 0.0;
    double dLength;
    double dPipeFactor;
    double dCFactor = 0.0;
    double hLoss;
    double vSign;
    double dQ;
    double dHWCFactor;
    PCMP pFp = pCmp->pFp;
    if (pFp == NULL)
```

```
{
    return 1;
}

// Get the internal diameter of the component. If this is a sink, use the
// diameter of the feeder-path component
if (IsPipeCmp(pCmp))
{
    ddiam2 = gapPtFldLine[pCmp->iPtRow]->dInDiam;
}
else
{
    ddiam2 = gapPtFldLine[pCmp->pFp->iPtRow]->dInDiam;
}

if (IsPipeCmp(pCmp->pFp))
{
ddiam1 = gapPtFldLine[pCmp->pFp->iPtRow]->dInDiam;
}
else
{
    ddiam1 = gapPtFldLine[pCmp->iPtRow]->dInDiam;
}

// Calculate area, mass flow, pressure, elevation, velocity, sign
dArea2 = (dPiFour*ddiam2*ddiam2)/dAreaConv;
dArea1 = (dPiFour*ddiam1*ddiam1)/dAreaConv;
mdot = pCmp->adThrough_re[0];
P1 = pFp->adAcross_re[0];
z1 = pFp->dZCoord;
z2 = pCmp->dZCoord;
v = mdot / (dRho * dArea2);
vSign = (v < 0.0) ? -1.0 : 1.0;

// If this is a pipe, get the length in feet; otherwise, the component has no length
if (IsPipeCmp(pCmp))
{
    dLength = pCmp->pChn->dCmpLngthLul * LULINMULS;
}
else
{
    dLength = 0;
}

// dPipeFactor is pipe loss computed from the friction factor, length and diameter
if (goApiFldWorkDef.fUseFrictionTerm)
{
    if (goApiFldWorkDef.iFrictionCalc == 0)
    {
        // Hazen Williams Friction Calculation
        dQ = (fabs(v) * pow(ddiam2*dInToFt,2) * PI)/4;
        dHWCFactor = gapPtFldLine[pCmp->iPtRow]->dCFactor;
        hLoss = (dCfHazenWilliams * dLength * pow(dQ, 1.852)) /
            (pow(dHWCFactor, 1.852)*pow(ddiam2*dInToFt, 4.87));
    }
    else if (goApiFldWorkDef.iFrictionCalc == 1)
    {
        // Darcy-Weisbach Friction Calculation
        // Calculate the Reynolds number. If the Reynold's number is
        // zero (zero velocity), set the friction factor to a nominal
        // value of 0.001. For turbulent flow (Re > 2100), use the
        // Swamee-Jain approximation to the Moody curve
        if (v == 0.0)
        {
            f = 0.001;
        }
        else
        {
            Re = fabs((v * ddiam2 * dInToFt) / dKinVis);
            if ((Re > 0.0) && (Re < 2100.0))
            {
                f = 64.0 / Re;
            }
            else
            {
                f = .25 / pow((log10((epsilon / (3.7 * ddiam2 * dInToFt)) +
                    (5.74 / pow(Re, 0.9)))),2);
            }
        }

        dPipeFactor = (f*dLength)/(ddiam2*dInToFt);
        hLoss = (dPipeFactor * dRho * v * v)/(2*dGrav);
    }
    else if (goApiFldWorkDef.iFrictionCalc == 2)
    {
        // Manning Friction Calculation
        // Q is the pipeline flow rate in m^3/s
        dQ = (fabs(v) * pow(ddiam2*dInToFt, 2) * PI) / 4;
        hLoss = (dCfManning * dLength * pow(dN*dQ, 2)) /
            pow(ddiam2*dInToFt, 5.33);
    }
}

// Do the calculation in units of lb/ft^2 and then convert to psi. Note:
// pressure decreases from friction loss in the direction of the flow
if (goApiFldWorkDef.fUseElevationTerm)
{
    dElevationTerm = dRho * (z1 - z2);
}
else
{
    dElevationTerm = 0.0;
}

P2 = ((P1 * dAreaConv) - ((hLoss ) * vSign) + dElevationTerm) / dAreaConv;
if (P2 < dAtm)
{
    P2 = dAtm;
}

pCmp->adAcross_re[0] = P2;
return 0;
```

```
}

BOOL IsPipeCmp(PCMP pCmp)
{
    if (    pCmp->iCmp == ICMP_PIPE                         ||
            pCmp->iCmp == ICMP_CHILLWATER_PIPE      )
    {
            return TRUE;
    }
    else
    {
            return FALSE;
    }
}

BOOL IsValveCmp(PCMP pCmp)
{
    if (    pCmp->iCmp == ICMP_GATE_VALVE   ||
            pCmp->iCmp == ICMP_CHECK_VALVE       )
    {
            return TRUE;
    }
    else
    {
            return FALSE;
    }
}

int MassFlowRate(PCMP pCmp)
{
    // Calculate the Mass Flow Rate through the component
    double P1;
    double P2;
    double z1;
    double z2;
    double Re;
    double v;
    double vSign;
    double mdot;
    double vsquared;
    double dHeadTerm=0.0;
    double dFrictionTerm=0.0;
    double dElevationTerm=0.0;
    double dCFactor=0.0;
    double hLoss=0.0;
    double f=0;
    double dConst;
    double dBeta;
    double dLength;

    PCMP pFp = pCmp->pFp;
    if (pFp == NULL)
    {
            // No upstream component. This is taken to imply that the
            // downstream pressure and elevation are the same as those
            // in the current component, so there is no flow.
    pCmp->adThrough_re[0] = 0.0;
    return 0;
}

if (pCmp->iCmp == ICMP_FLUID_LOADBUS)
{
    ddiam=gapPtFldLine[pCmp->pFp->iPtRow]->dInDiam;
}
else
{
    ddiam=gapPtFldLine[pCmp->iPtRow]->dInDiam;
}

dArea = (dPiFour * ddiam * ddiam) / dAreaConv;
P1 = pFp->adAcross_re[0];
P2 = pCmp->adAcross_re[0];
z1 = pCmp->pFp->dZCoord;
z2 = pCmp->dZCoord;
mdot = pCmp->adThrough_re[0];
v = mdot / (dRho * dArea);
vSign = (v < 0.0) ? -1.0 : 1.0;

if (IsPipeCmp(pCmp))
{
    dLength = pCmp->pChn->dCmpLngthLul * LULINMULS;
}
else
{
    dLength = 0.0;
}

// Calculate the Reynolds number. If the Reynold's number is
// zero (zero velocity), set the friction factor to a nominal
// value of 0.001. For turbulent flow (Re > 2100), use the
// Swamee-Jain approximation to the Moody curve.
if (v == 0.0)
{
    Re = 0.0;
    f = 0.001;
}
else
{
    Re = fabs((v * ddiam2 * dInToFt) / dKinVis);
    if ((Re > 0.0) && (Re < 2100.0))
    {
        f = 64.0 / Re;
    }
    else
    {
        f = .25 / pow((log10((epsilon/(3.7 * ddiam2 * dInToFt)) +
            (5.74 / pow(Re, 0.9)))), 2);
    }
}

// dCFactor accounts for expansion and contraction
// dPipeFactor is pipe loss computed from the friction factor, length and diameter
```

```
    // dControl is total head loss at a control valve from the valve curve database
    if (ddiam2 > ddiam1)
    {
        dBeta = dArea1 / dArea2;
        dCFactor = ((1.0/dBeta) - 1.0) * ((1.0/dBeta) - 1.0);
    }
    else if (ddiam1 > ddiam2)
    {
        dBeta = dArea2 / dArea1;
        dCFactor = 0.45 * (1.0 - dBeta);
    }

    // This routine is solving for the mass flow rate (and hence the velocity).
    // Since it is assumed that the entire component has a constant flow rate,
    // the velocity enters through the friction loss term which is:
    //
    //    hLoss = ((dRho * v * v) / (2.0 * dGrav)) * ((f*L/D) + dCFactor)
    //
    // This must balance the head and elevation terms. This gives
    //
    //    v^2 = (2*g*D(dPipeFactor+dCFactor) / f*L)((P1 - P2 / dRho) + z1 - z2)
    dConst = (2.0 * dGrav) / (((f * dLength) / (ddiam2*dInToFt)) + dCFactor);
    dHeadTerm = dConst * (P2 - P1) * dAreaConv / dRho;
    dElevationTerm = dConst * (z2 - z1);
    vsquared = dHeadTerm + dElevationTerm;
    if (vsquared == 0.0)
    {
        v = 0.0;
    }
    else if (vsquared > 0.0)
    {
        v = sqrt(vsquared);
    }
    else
    {
        vsquared = fabs(vsquared);
        v = -sqrt(vsquared);
    }

    mdot = dRho * dArea * v;
    pCmp->adThrough_re[0] = mdot;
    return 0;
}

int ValveDeltaP(PCMP pCmp)
{
    // The value dCv is the loss coefficient for the valve from the database. The
    // Pressure drop across the valve is given by: deltaP = Cv * rho * v^2 / 2g
    PCMP pFp;
    double P1;
    double P2;
    double v;
    double mdot;
    double vSign;
    double hLoss;
    double dCv=0.0;
double dControl=0.0;
double dPSet;
double dMinCv;
double dMaxCv;
double dTarget;

// P1 is the inlet pressure and P2 is the outlet pressure. P1 is obtained
// from the feeder-path component.
pFp = pCmp->pFp;
if (pFp != NULL)
{
    // Get C Factor
    if (IsValveCmp(pCmp))
    {
        dCv = gapPtFldValv[pCmp->iPtRow]->dCFactor;
    }

    // Calculate diameter, area, pressure, mass flow, velocity, sign
    ddiam = gapPtFldValv[pCmp->iPtRow]->dSize;
    dArea = (dPiFour * ddiam * ddiam) / dAreaConv;
    P1 = pFp->adAcross_re[0];
    mdot = pCmp->adThrough_re[0];
    v = mdot / (dRho * dArea);
    vSign = (v < 0.0) ? -1.0 : 1.0;

    // Pressure reducing valve automatic controls
    if ((pCmp->iCmp == ICMP_PRV)&&(mdot != 0.0))
    {
        if (pCmp->pCmpDCtr->adVals[0] && pCmp->pCmpDCtr->adVals[0] >=
            14.7)
        {
            dPSet = pCmp->pCmpDCtr->adVals[0];
            dMinCv =    gapPtFldValvCCrv[gapPtFldValv[pCmp->iPtRow]
                ->iValvCCrv]->lPoints[1];
            dMaxCv = gapPtFldValvCCrv[gapPtFldValv[pCmp->iPtRow]
                ->iValvCCrv]->lPoints[10];
            if (P1 <= dPSet)
            {
                dControl = (fabs(mdot)/dMaxCv) * (fabs(mdot)/dMaxCv);
            }
            else
            {
                double dPres = P1 - dPSet;
                dTarget = fabs(mdot)/(sqrt(P1 - dPSet));
                if (dTarget > dMaxCv)
                {
                    dControl = (fabs(mdot)/dMaxCv) * (fabs(mdot)/dMaxCv);
                }
                else if (dTarget < dMinCv)
                {
                    dControl = (fabs(mdot)/dMinCv) * (fabs(mdot)/dMinCv);
                }
                else
                {
                    dControl = (fabs(mdot)/dTarget) * (fabs(mdot)/dTarget);
                }
                }
            }
        }

        // Head loss
        hLoss = (((dRho * v * v) / (2.0 * dGrav)) * dCv) + (dControl * dAreaConv);
        P2 = (P1 * dAreaConv) - (hLoss * vSign);
        P2 = P2/dAreaConv;
        if (P2 < dAtm)
        {
            P2 = dAtm;
        }

        pCmp->adAcross_re[0] = P2;
    }
    return 0;
}

int ValveFlowRate(PCMP pCmp)
{
    PCMP pFp;
    double P1;
    double P2;
    double v;
    double mdot;
    double vSign;
    double dCv=1.0;
    double ddiam;
    double dArea;
    double vsquared;

    pFp = pCmp->pFp;
    if (pFp != NULL)
    {
        if (IsValveCmp(pCmp))
        {
            dCv = gapPtFldValv[pCmp->iPtRow]->dCFactor;
        }

        ddiam = gapPtFldValv[pCmp->iPtRow]->dSize;
        dArea = (dPiFour * ddiam * ddiam) / dAreaConv;
        P1 = pFp->adAcross_re[0];
        P2 = pCmp->adAcross_re[0];
        vSign = (P1 >= P2) ? 1.0 : -1.0;
        vsquared = fabs((P1 - P2) * (2.0 * dGrav)/(dRho * dCv));
        v = sqrt(vsquared) * vSign;
        mdot = dRho * v * dArea;

        // Store the mass flow rate in lb/s
        pCmp->adThrough_re[0] = mdot;
    }
    return 0;
}
```

Filename:	Feinauer_LR_D_2009.doc
Directory:	C:\VirginiaTech\Dissertation\Dissertation ETD\Final
Template:	C:\Documents and Settings\lynnf\Application Data\Microsoft\Templates\Normal.dot
Title:	Generic Flow Algorithm for Analysis of Interdependent Multi-Domain Distributed Network Systems
Subject:	
Author:	Lynn R. Feinauer
Keywords:	
Comments:	
Creation Date:	10/19/2009 12:17 PM
Change Number:	11
Last Saved On:	10/23/2009 3:03 PM
Last Saved By:	Lynn R. Feinauer
Total Editing Time:	33 Minutes
Last Printed On:	10/23/2009 3:03 PM

As of Last Complete Printing

Number of Pages:	106
Number of Words:	21,347 (approx.)
Number of Characters:	121,683 (approx.)