FORM: The FORTRAN Object Recovery Model. A Methodology to Extract Object-Oriented Designs From Imperative Code.

Bonnie Lynn Achee
Louisiana State University and Agricultural & Mechanical College

FORM: THE FORTRAN OBJECT RECOVERY MODEL – A METHODOLOGY TO EXTRACT OBJECT-ORIENTED DESIGNS FROM IMPERATIVE CODE

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University an
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Bonnie Lynn Achee
B.S. Southeastern Louisiana University, 1990
May 1998

in memory of my parents,

Acknowledgements

I would like to begin by thanking God for His presence in my life. Throughout the course of my studies at LSU, I have experienced some of the most joyous and most devastating moments of my personal life. He has walked beside me to guide me and carried me through the times when I felt I could not go on. I am truly grateful for His unconditional love and its manifestation in the many friends I have made during my graduate career.

I would like to extend my sincere appreciation to Dr. Doris L. Carver who has been my professor, advisor and friend. She has influenced my graduate career from beginning to end. It is through counseling with her that I applied for a Board of Regents Fellowship at LSU and entered the program. It was her instruction in software engineering and programming languages that sparked my excitement in the area of reverse engineering. It was her guidance that helped me to develop a research topic, and her motivation and sympathetic ear that has helped me to complete it.

My appreciation is also extended to my doctoral advisory committee: Drs. Chari, Iyengar, Jones, Kraft, and Oxley, for taking the time to review my dissertation.

My husband, Eric Stegen, has been a great source of strength. He has been my source of security throughout my studies. His patience, understanding and encouragement have been a great motivation to me.

Finally, I would like to thank my parents. Although they are not here to share this milestone in my life, without them nothing would have been possible. It is their unconditional love and confidence that made me even dare to dream that I could achieve such a goal. Although my father never saw me enter graduate school, it was through his expectations and belief in me. The gratitude that I have for my mother, her role in my graduate studies, her encouragement and her sacrifice could never be adequately expressed in words. Throughout my life she has been the wind beneath my wings.

Table of Contents

List of Figures

Abstract

A majority of legacy systems in use in the scientific and engineering application domains are coded in imperative languages, specifically, COBOL or FORTRAN-77. These systems have an average age of 15 years or more and have undergone years of extensive maintenance. They suffer from either poor documentation or no documentation, and antiquated coding practices and paradigms [Chik 94] [Osbo 90]. The purpose of this research is to develop a reverse-engineering methodology to extract an object-oriented design from legacy systems written in imperative languages. This research defines a three-phase methodology that inputs source code and outputs an object-oriented design.

The three phases of the methodology include: Object Extraction, Class Abstraction, and Formation of the Inheritance Hierarchy. Additionally, there is a pre-processing phase that involves code structuring, alias resolution, and resolution of the COMMON block. Object Extraction is divided into two stages: Attribute Identification and Method Identification. The output of phase one is a set of candidate objects that will serve as input for phase two, Class Abstraction. The Class Abstraction phase uses clustering techniques to form classes and define the concept of identical objects. The output of phase two is a set of classes that will serve as input to the third phase, Formation of the Inheritance Hierarchy. The Formation of the Inheritance Hierarchy phase defines a similarity measure which determines class similarity and further refines the clustering performed in phase two, Class Abstraction. The result of the methodology is an object-oriented design including hierarchy diagrams and interaction diagrams. Additionally, the results of applying the methodology in two case studies are presented.

The research has resulted in the development of a unique methodology to extract object-oriented designs from imperative legacy systems. The benefits of using the methodology include: the ability to capture system functionality which may not be apparent due to poor system structure, and the reduction of future maintenance costs of the system as a direct effect of accurate system documentation and updated programming technologies.

Chapter 1

Introduction

In recent years, much progress has been made in the area of software development. Specifically, the introduction and acceptance of the object-oriented paradigm has resulted in software systems exhibiting such desirable properties as code reuse, modularity, deferred commitment and a model that closely resembles the real world. However, "much of the software we depend on today is, on average, 10 to 15 years old" and written primarily in imperative languages, specifically COBOL or FORTRAN-77. Thus, the above benefits are not realized in these systems. Moreover, years of "patching" has resulted in systems that are poorly structured, coded and documented. It is clear that these systems will have to be "cleaned-up", however, the cost-factor makes it unlikely that these working systems will be simply discarded. Therefore, another approach must be taken [Osbo 90].

Reverse-engineering is recognized as a way to migrate old systems to new and improved technologies [Ulri 90]. By reverse-engineering a legacy system to take advantage of new technologies, the resulting system enjoys increased flexibility, increased productivity, and accurate system documentation. Additionally, it provides a better methodology for maintenance, allows rapid adaptation to changing requirements, and utilizes the benefits of new technologies and architectures [Roch 90]. The importance of this can be fully appreciated when it is realized that software maintenance consumes over 50% of the budget in most data processing shops [Your 89]. Therefore, the objective of this research is to develop a methodology to reverse-engineer such systems into the object-oriented paradigm, thereby aiding the migration of the system to newer coding practices and paradigms while utilizing system requirement information contained in the source code itself, but not appearing in any other documentation source. This research defines a reverse-engineering methodology to extract an object-oriented design representation from an imperative legacy system.

The remainder of this chapter presents an overview of the problem of software maintenance, introduces the relevant concepts and motivations, presents the objectives of the research and describes the organization of this dissertation.

1.1 Overview

The traditional forward-engineering software lifecycle of imperative systems includes several distinguishable stages: requirements analysis and definition, system design, implementation and testing, and operation and maintenance. Legacy systems are systems that are systems that are, on an average, over 10 years old. They were developed under the forward-engineering lifecycle model just described and currently exist in the operation and maintenance phase [Somm 89].

Software maintenance not only involves error-correction but also includes such activities as modification of requirements and design, thereby requiring further implementation. Moreover, what occurs, is that the entire development process is repeated many times during system maintenance as system modifications are made to the software. Maintenance costs are known to be the greatest cost incurred in software development, averaging two to four times the development costs for large embedded systems. It is noted that "maintenance costs tend to rise with program age." Although maintenance costs tend to be less for systems that are well documented, as maintenance activities continue, the quality and accuracy of system documentation drops sharply. Therefore, systems that were well documented at release time may be poorly documented 10 years later due to excessive maintenance activities [Somm 89]. Rather than continue to nurse an antiquated system, frequently the decision is made to extract the functionality of the system to utilize new technologies while making the changes at a higher level of abstraction (i.e., system design or specification). The goal is to improve overall system quality, decrease future maintenance costs, and satisfy current user needs. Reverse-engineering is the mechanism by which the system functionality is extracted. Specifically, it is the part of software maintenance "that helps you understand the system so you can make appropriate changes." [Chik 90]

The purpose of reverse-engineering a system is to increase system understanding or comprehensibility for maintenance and development. In addition, research in the area of reverse-engineering addresses at least one of the following six key objectives: cope with complexity, generate alternate views, recover lost information, detect side-effects, synthesize higher abstractions, and facilitate reuse. Reverse-engineering involves analyzing a system to "identify the system's components and their interrelationships" and to "create a representation of the system in another form or at a higher level of abstraction [Chik 90]." As the connotations of reverse-engineering have changed from negative to necessary, there has been a concentration of research in the area. No longer is reverse-engineering clouded by the idea that it is an admission of failure because of the "get it right the first time" mentality. Today it is realized, and widely accepted that a software system is dynamic. "It is not possible to predict what you will want a system to do five, or even two years from now [Wate 94]." Therefore, by using the techniques of reverse-engineering to achieve an object-oriented design, the benefits of current software technologies can be realized without discarding a working system. [Chik 90]

Object-orientation is the amalgamation of three concepts: encapsulation, polymorphism and inheritance. Of these, only inheritance is unique to the paradigm. Encapsulation is realized as a "class" which is the implementation of an abstract data type. Classes are instantiated to give "objects" of the type, which form the basic run-time entity of the system. . The object, which is the primitive element, can be viewed as an abstract data type, encapsulating a set of data (i.e. attributes) and a corresponding set of permissible actions on the data (i.e. methods). Each object is an autonomous entity which interacts with other objects during the execution of the system. Polymorphism is a property that permits a single message to refer, at run-time, to instances of different classes. Inheritance defines a relation between classes whereby the definition of a class is based on the definition of existing classes. It encourages the reuse of classes that are similar to what the programmer wants by allowing the programmer to tailor the inherited class(es) to meet the needs of the inheriting class in a way that will not affect the inherited class(es). Thus, the combination of inheritance, polymorphism and dynamic binding localize required changes, thereby minimizing the amount of code that must be modified during software maintenance [Somm 89]. Other benefits of the object-oriented paradigm include code reuse, modularity, deferred commitment, and a model that closely resembles the real world [Pokk 89].

The research detailed in this dissertation is motivated by the following:

- The majority of scientific and engineering software systems currently in use are coded in imperative languages, specifically, FORTRAN-77.
- The object-oriented paradigm is well-suited for large-scale programming systems such as scientific and engineering systems, and provides a great opportunity for software reuse.
- The reverse-engineering of software systems allows the utilization of desirable system functionality for use in reengineering.

The goal of this research is to develop a methodology that facilitates system migration of legacy systems coded in FORTRAN-77 to the object-oriented paradigm. Additionally, the objectives of the research are as follows:

- Issues specific to FORTRAN-77, such as the COMMON block, should be addressed.
- Algorithms for each phase of the methodology should be detailed.

- A collection of design documents should be developed to represent the extracted design, such that the characteristics specific to reverse-engineering are addressed.

1.2 Outline of the Dissertation

The outline of the dissertation is as follows:

Chapter 2 presents related research in the area of reverse-engineering, specifically, object recovery. The relevance of the related work to the dissertation, as well as the distinction of the related work from the dissertation is presented.

Chapter 3 details the FORTRAN Object Recovery Methodology (FORM). It discusses each phase: pre-processing, object extraction, class abstraction and abstraction of the inheritance hierarchy. All necessary algorithms, definitions, and lemmas are presented. The representation of the object model is explained and the template for each diagram is given. The chapter concludes with a section on the validation and evaluation of FORM.

Chapter 4 presents the results of two case studies. Using two subject systems, one small-scale (less than 500 lines of code) and one medium-scale (1000 - 5000 lines of code) the FORTRAN Object Recovery Methodology (FORM) is demonstrated. Results from each phase of the methodology are presented.

Chapter 5 offers some concluding remarks and discusses possible future research directions.

Chapter 2

Related Research

In the context of software-engineering, reverse-engineering describes "the process of discovering how your own system works." It involves many activities that all relate to the understanding and modification of existing software systems including creating high-level descriptions of a system. Because software systems are dynamic, it is not possible for a system to be permanently correct. This is evidenced in such factors as changing user needs and rapidly advancing hardware technologies. The rapid decrease in computer cost is making it possible for great advancements in both hardware and software technologies. For this reason, it is impossible to predict what users will expect from systems in the distant, and not-so-distant, future. For all of these considerations, reverse-engineering has been called "one of the most important areas of software engineering, rather than being a peripheral concern." It is this realization that has sparked interest in the area and fueled the flame of research, which has resulted in conferences, such as the Working Conference on Reverse Engineering, which are devoted solely to reverse-engineering. The widespread appeal of object-oriented programming and the realization of the necessity of reverse-engineering as a software maintenance activity have motivated research in various aspects of the field [Wate 94].

Research in the area of reverse-engineering, specifically object recovery, is classified into five areas: code restructuring, program understanding, structure identification, design & specification recovery, and system migration [Wate 94]. The remainder of this chapter will present the related research in each of these five areas and discuss the relevance of the related research to this research.

2.1 Code Restructuring

Code restructuring marks the beginning of research in the area of reverse-engineering. The purpose of code restructuring and the central theme of all reverse-engineering activity is program improvement. Code restructuring seeks to improve existing code by improving its understandability. Because unstructured program logic results in increased program complexity, code restructuring improves intellectual control over programs and makes reliable modification and software evolution feasible [Haus 90; Water 94].

The beginning of research in the area is marked by [Haus 90] and [Zimm 90] which detail the restructuring of COBOL and FORTRAN code, respectfully. Hausler discusses the structuring of COBOL code to improve maintainability. By eliminating the constructs of Alter and Goto, a "top-down hierarchy of structured, single-entry, single-exit, procedures" is produced [Haus 90]. Zimmer presents a method for restructuring FORTRAN code into an object-oriented style. Zimmer's research seeks to increase program clarity by establishing global invariants, reducing data cobwebs, and designing single objects. By emphasizing program invariants, the program structure is altered towards program function to create object modules. The resulting programs are written in object-module style and contain three kinds of modules: main, traditional, and object. The main module is the main program, traditional modules contain only a single subprogram, and object modules contain one or more subprograms with an interface which may contain one or more variables. Thus, the object module style implements a particular object as a set of subprograms, and are, therefore, less general than data abstractions (which provide parameterization or object oriented programming (which provides inheritance). Although this method results in code which is not truly object-oriented, it marks the beginning of research along that path [Zimm 90].

2.2 Program Understanding

Program understanding, a key issue in reverse engineering, seeks to comprehend the underlying functional and data concepts of a system. Program understanding involves both syntactic and semantic analysis. Syntactic analysis in the most basic level of program understanding. It involves analyzing syntactic units such as variables, reserved words, strings and consonants as well as generating syntax trees. Semantic analysis provides a much deeper insight into program behavior detailing such information as control-flow and data-flow dependencies [Baum 93].

Research in the area of programming understanding has resulted in such projects as AMES [Baum 93], the Recognizer [Rich 93], COBOL/SRE [Engb 93], and DESIRE [Bigg 94]. An Extensible Maintenance Engineering System (AMES) was developed as a prototype of a semantics-based program understanding system for COBOL 74 programs. AMES uses denotational semantics and static program analysis techniques to develop tools that aid in semantic program understanding.

Semantic based tools provide a much deeper level of program understanding than syntactic based tools by supporting semantic level understanding through control-flow analyzers, data flow analyzers, and program slicers as opposed to parse trees and token lists. AMES designs and implements a prototype of a COBOL 74 software maintenance environment. It is composed of three parts: the maintenance engine, which is coded in Standard ML and consists of all the tools and methods; the user interface, which is coded in C++ as a stand-alone application running as a separate process; and the data store, which contains the parsed source code in the form of an abstract syntax tree extended with "containers" and "annotations" [Baum 93].

The Recoginzer is a prototype which finds occurrences of commonly used data structures and algorithms, defined as clichés, and builds a hierarchical description of the program. It was developed at MIT as part of the Programmer's Apprentice project. The program under evaluation is first translated into a "language-independent graphical representation", the Plan Calculus. It is then encoded as a flow graph and parsed to produce a design tree. Finally, documentation is generated [Rich 93].

The COBOL/SRE (COBOL System Renovation Environment) project is a software re-engineering environment for COBOL systems. It was developed by Andersen Consulting's Center for Strategic Technology Research. COBOL/SRE was developed as a set of tools to address the problem of "identifying and extracting components from large legacy COBOL systems" based on the concepts of reusable component recovery. Reusable component recovery represents one of several approaches in dealing with legacy systems. In reusable component recovery, functional components of the system are "recognized, recovered, adapted, and finally reused in new system development." This approach requires deep analysis and understanding of the legacy code. To this end, COBOL/SRE includes such components as system level analysis, data model recovery, concept recognition and program level analysis. The system-level analysis includes system inventory, system analysis, and system browsing capabilities. The data model recovery component identifies a virtual data model based on analysis performed on data record mappings, data assignments, and data flows. The concept recognition component uses "plans" to "describe parts and constraints among parts of concepts to be recognized." Finally, the program-level analysis component assists program analysts in program understanding activities by providing parsing and program text browsing, flow analysis, complexity analysis and anomaly detection, and program segmentation. COBOL/SRE uses such features as condition-based slicing, forward slicing/ripple-effect analysis, segment management and composition operations, and knowledge based concept recognition to facilitate system level analysis and browsing, syntactic analysis, semantic analysis, data model recovery, and distributed execution architecture in reusable software component recovery from legacy COBOL systems [Ning 94].

The DESIRE (DESign Information Recovery Environment) system is a suite of tools that uses informal information (comments, identifier names, design documents) rather than formal information (syntax trees and program semantics) to aid in program understanding, specifically addressing the concept assignment problem. The concept assignment problem is defined as "the problem of discovering these human-oriented issues and assigning them to their realizations within a specific program or its context." DESIRE is a program-understanding assistant containing facilities to assist the user in addressing the concept assignment problem. These facilities include three scenarios: suggestive data names, patterns of relationships, and intelligent agent. Scenario one, suggestive data names, assigns key concepts to specific program concepts to provide a framework whereby a human reverse engineer may perform further detailed analysis. Scenario two, patterns as relationships, identifies "clusters of related functions and data that form the framework of the program." Scenario three, intelligent agent, allows the user to browse the code looking for "evidence of key concepts based on the user's experience." DESIRE has been used for "exploration for debugging or porting," and "documentation for understanding and reporting" [Bigg 94].

2.3 Structure Identification

Structure identification is the process of determining static properties of a software system. Research in this area includes such projects as RE^2, data-flow-diagram (DFD) extraction, program graph models, and the FORTRAN Reverse-Engineering package.

Bendusi, et. al. describe the development of a methodology to extract low level design documents from Pascal code using Transformation Analysis. The methodology produces structure charts and data-flow diagrams using JSP and Warnier-Orr methodologies [Bend92]. The RE^2 project explores "reverse-engineering and re-engineering techniques to facilitate reuse re-engineering." The project uses functional and data abstraction to extract reusable components from existing systems. Additionally, it seeks to introduce abstract data types into languages that do not make any provisions for the implementation of abstract data types [Canf 93]. Cimitile introduces an algebraic representation of program modules to generate program graph models. Using these program graph models, a flow-graph, a nesting tree of program control structures, and a tree of program paths are produced from software coded in FORTRAN, COBOL or Pascal [Cimi 91]. Finally, the Fortran Reverse Engineering package analyzes FORTRAN code and produces structure charts and module specifications [Gili 90].

Each of these research efforts extracts static components from source code. They do not modify or interpret the extracted data. This extraction process does not provide the deeper comprehension that program understanding offers, but rather serves as a preliminary stage of program understanding. Additionally, program understanding and structure identification serve as preliminary stages to object recovery.

2.4 Design & Specification Recovery

A software design is a description of the software system [Pfle 91]. It is the process of "producing a description of implementation from which source code can be developed [Ruga 90]" A software specification, on the other hand, is a description of the systems capabilities [Pfle 91]. Both software specification and design are vital components of the forward engineering lifecycle. They are vital documentation that represents the system. However, as previously discussed, because legacy systems have undergone years of extensive maintenance, what was originally accurate specification and design documentation typically is no longer accurate. Therefore, a key objective of reverse engineering is recovery of lost information, specifically, system specification and design [Chik 90]. There have been numerous research efforts in design and specification recovery including such projects as REDO, RECAST, NuMIL and RIGI [Lano 93; Edwa 93; Choi 90; Till 93].

Choi & Sacchi explore the extraction of the functional and dynamic properties of large systems and develop a process to reverse-engineer system level design description. They developed a module interconnection language, NuMIL, which is used to represent the extracted design [Choi 90]. Liu & Wilde propose a methodology to recover object-like features from a non-object oriented system. The methodology uses features such as abstract data types to identify object-like features using persistent data and formal parameters [Liu 90]. Lividas & Roy extend this research by introducing the concept of the receiver of a procedure. Thus, the research is extended to explore the object-like features in receivers [Livi 92].

Sward & Steigerwald have developed a two-phase methodology to reverse-engineer procedural code into natural language. Phase one describes the data structures in three steps: list the data structures, list where the data structures are defined, and write a natural language description of the data structures. Phase two uses a five step procedure to describe the procedures of the system. Phase two involves the following steps: list all procedures in the system, list the parameters for each of these procedures, consider each data structure individually, list the parts of the data structure that each procedure uses and write natural language descriptions for each procedure. They claim that by extracting a natural language description of the source code, that the forward-engineering process will allow the system to be implemented in the object-oriented paradigm [Swar 94].

The RECAST project was carried out at the Centre for Software Maintenance, University of Durham, as part of the DTI/ED project under the Information and Engineering Advanced Technology program supported by a SERC grant. The RECAST (Reverse Engineering into CaSe Technology) methodology extracts a SSADM (Structured Systems Analysis and Design Method) from COBOL code. Thus, the project extracts a procedural design from legacy COBOL code. The RECAST framework has four phases: population of the repository, preliminary transformations, logical restructuring, and translation into SSADM notation. Phase one parses the source code and generates PSL statements to populate the PSL/PSA repisotory of SSADM. Phase two resolves any naming difficulties (synonyms and homonyms) and carries out preliminary transformations on the source system. Phase three is the heart of RECAST. During this phase, a set of transformations is used to restructure the repository descriptions while maintaining system functionality. Finally, phase four translates the meta language description of the system into SSADM. The final output is the set of documents required for SSAMD's Physical Design phase [Edwa 93].

The RIGI project involves such concepts of reverse-engineering as software analysis and program understanding to identify software artifacts and form an abstract representation of the system. RIGI discovers and analyzes the structure of large software systems in two phases. Phase one involves the automatic and language-dependent extraction of software artifacts. This is performed by parsing the source code and storing the artifacts in a repository. Phase two involves semi-automatic and language independent "subsystem composition methods that generate hierarchies of subsystems." That is, using the variables, procedures, modules, and subsystems to construct software components [Till 93; Mull 94].

Subramaniam and Byrne strive to derive an object model from legacy FORTRAN code. Using application domain knowledge and system artifacts including documentation and source code they define a nine-step process involving both top-down and bottom-up approaches. The process includes identifying potential objects and classes, mapping objects to classes, identifying attributes, identifying methods, and determining relationships [Subr 96].

In addition to design recovery, there have been several projects dedicated to the recovery of a system specification. Such projects include REDO, REFORM and a methodology introduced by Gannod & Cheng. The aim of the REFORM (Reverse Engineering using FORmal Methods) project is to develop a formal specification in

Z from legacy IBM Assembler code. This project was conducted at the Centre for Software Maintenance at the University of Durham as part of research funded by IBM(UK), DTI, and SERC. The REFORM system uses a three phase methodology to transform the source code into a specification. Phase one uses a source to wide-spectrum language (WSL) translator to translate the assembler into an intermediate form. It produces code modules and the relations among the modules and stores this information in a database. Phase two involves using the code stored in the database and working interactively with the program transformer to produce a specification. Again, the resulting code is stored in the database. The third and final phase uses a program integrator to assemble the code in a WSL and translate this specification in WSL to a Z specification [Yang 91].

The REDO (Reengineering, Documentation and validation of systems) project was conducted at Oxford University by the Programming Research Group (PRG) as part of funded research known as the ESPRIT project. The goal of the REDO project was to reverse-engineer COBOL to the formal specification language, Z. This was done via an intermediate language, Uniform, and a functional description language. Further research in the project included the development of the object-oriented specification language Z++ and the development of a methodology to extract the object-oriented components of COBOL and represent those components in Z++ [Lano 93].

2.5 System Migration

System migration involves the transformation of a software system from one language to another. Bryne discusses the development of a methodology to develop an Ada implementation from a FORTRAN code and update the documentation. The methodology involves extracting detailed design information from the FORTRAN source code. From the detailed design, a high level design is extracted and represented by data-flow and contrl-flow diagrams. The following eight-step procedure to extract the design is presented: collect information, examine information, extract the structure, record functionality, record data-flow, record control-flow, review recovered design, and generate documentation. The recovered design is then implemented in Ada and new documentation is produced [Bryn 91].

Ong & Tsai have developed a methodology to translate FORTRAN code to class-based C++ code. They examine aggregated data structures and subprogram parameters to help facilitate the translation. Data flow analysis is used to gather information on program variables. Using this information, subprograms are analyzed for objects in global variables, local variables and formal parameters. Methods are extracted based on a heuristic that determines data usage by defining three categories: use-only, use-and-set, and define-only. Based on the classification of the variables, methods are defined which read the object's state, create the object (constructor), or modify the object's state [Ong 93].

2.6 Relevance to the Dissertation

The research presented in the dissertation serves to extend and enhance the body of research discussed in this chapter. It spans several areas of reverse-engineering including program understanding and structure identification with major emphasis in the area of design and specification recovery. The methodology presented in the dissertation has similar theoretical foundations to the research presented, however there exists some fundamental differences. These similarities and differences are detailed in Figure 2.1. In the related research, only [Liu 90; Liva 92; Lano 93; Ong 93; Subr 96] seek to obtain object-oriented characteristics from non-object-oriented code. [Lano 93] focus strictly on COBOL source code, and therefore, does not deal with many of the issues encountered with FORTRAN. [Liu 90; Liva 92; Ong 93] each seek to extract object characteristics from non-object-oriented code, but use abstract data types as the basis of forming these object groupings. These methodologies fall short for FORTRAN because of its lack of abstract data types and user defined types. Therefore, although other methodologies may also take a data-driven approach to extraction of objects, the data elements that are considered are vastly different. Thus, the methodology presented in the dissertation expands and enhances the current body of research by increasing the domain of applicability of object recovery methodologies. This is exemplified in [Subr 96], which references and is based upon the research contained herein, and specifies C++ as the language of implementation.

Methodology	Source Language	Data Structures	Class Extraction
REDO	COBOL	required	yes
RECAST	COBOL	required	no
Ong & Tsai	FORTRAN	not required	methods only; class-based C++
Liu & Wilde	COBOL	reqired	methods only
FORM	imperative	not required	yes

Figure 2.1
Comparison of Related Research

Chapter 3

FORM: The FORTRAN Object Recovery Methodology

The methodology detailed in the dissertation is a three phase methodology for extracting an object-oriented design from imperative code, specifically FORTRAN-77. The FORTRAN Object Recovery Methodology (FORM) consists of a pre-processing phase to restructure the source code, resolve the aliases in the global variables and actual parameters, and resolve the COMMON block followed by Object Extraction, Class Abstraction and Formation of the Inheritance Hierarchy. An overview of the methodology is given in Figure 3.1. The result of FORM is an object-oriented design including hierarchy diagrams and interaction diagrams.

3.1 Pre-processing

The pre-processing phase is necessary to "clean up" the source code so that it may be evaluated accurately for object extraction. There are two types of preprocessing that must occur: actual parameters and global variables. The actual parameters must undergo a pre-processing phase to resolve any aliasing that occurs with the formal parameters. The global variables must undergo a pre-processing phase to resolve aliasing and to determine which variables serve only as placeholders. This pre-processing phase results in code that is ready to be analyzed for object extraction.

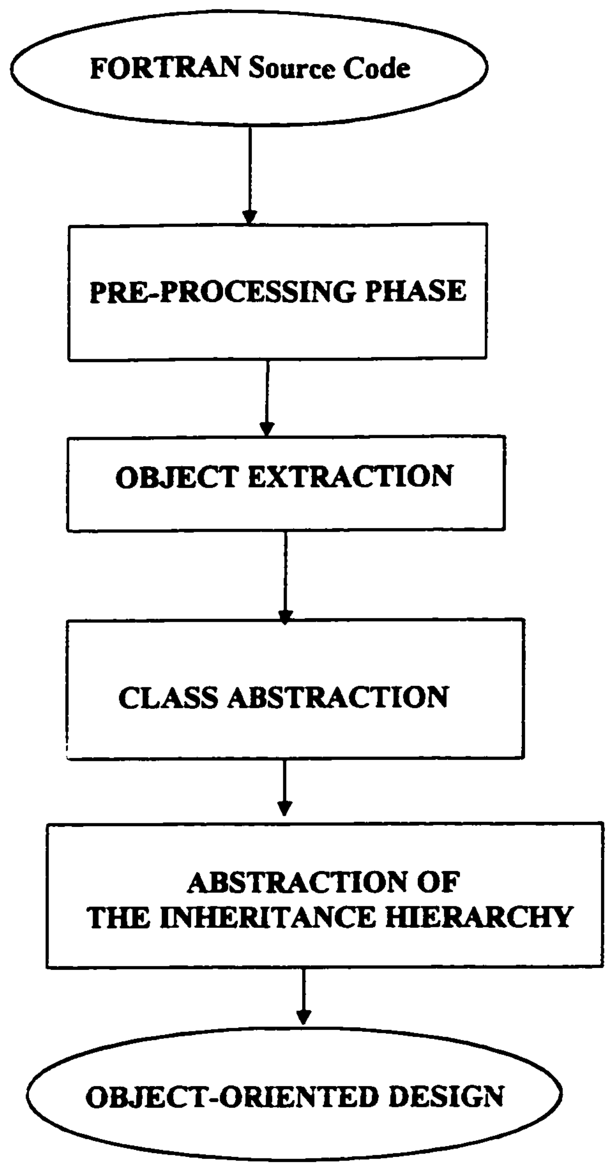

Figure 3.1
Overview of FORM

Input: Call graph of the source code represented as a tree with n levels where the formal parameters, and actual parameters for each node are maintained in two lists: formal and actual.

Output: Call graph of the source code with aliasing resolved.

1. Begin at the root node and traverse the tree using a depth first traversal.
2. For each node, maintain the following lists:

formal[i,k]:	the list of formal parameters
actual[i,j]:	the list of actual parameters

   where: i = 1..n is the level in the tree

   j is the number of actual parameters

   k is the number of formal parameters

```
3. for i = 1..n do
       j = 1
       while actual[i,j] <> eol do
           k=0
           while formal[i,k] <> eol do
               if actual[i,j] == formal[i,k]
               then actual[i,j] = actual[i-1,j]
           od
       od
   od
```

Figure 3.2
Algorithm to Resolve Actual Parameter Aliasing

3.1.1 The Aliasing Problem -- Actual Parameters

Because the first phase of the extraction process involves the analysis of the actual parameters, it is important that a single name of an actual parameter corresponds to a unique data element. That is, when the subroutine calls are considered independently of the source code and the call graph, no aliasing should occur. This is not the case, however. Because of the scoping rules of FORTRAN, as well as most imperative languages, the data element that is referenced in a subroutine call is based not only on the subroutine issuing the call, but also the order of nesting of the subroutines in the call graph.

There are two instances when the aliasing of parameter names in the subroutines occurs: (1) local variables of the same name in different subroutines are used as actual parameters, and (2) actual parameters and formal parameters have the same names and nested calls occur. Before any type of analysis is performed, the calling sequence must be considered. Thus, the call graph is used to give unique names to variables referencing a single location.

To solve the problem of local variables of the same name in different subroutines used as actual parameters, the name of the subroutine of declaration is attached in a dot notation to the beginning of each local variable. The source code is updated with the new variable names and uniqueness is guaranteed. To solve the problem of aliasing that occurs when actual parameters have the same names as formal parameters, the call graph is necessary. Using the call graph, mappings are determined as follows: when a call is issued by SUB1 to a subroutine SUB2, and

SUB2 issues a call to SUB3 where the name of a formal parameter appears in as an actual parameter, the actual parameter in SUB2 is replaced by the corresponding actual parameter of the calling subroutine, SUB1. Considering the call graph as a tree with the main program as the root, the resolution proceeds in a top-down manner. Beginning at the root node, list the mapping of actual parameters to formal parameters for each call. Traverse the tree in a depth-first manner. Maintain the mapping for all actual parameters to formal parameters. For each node, replace all occurrences of formal parameters in the call statements with the corresponding actual parameters and derive the actual parameter to formal parameter mapping for the next node in the sequence. The algorithm to resolve the actual parameter aliasing is given in Figure 3.2.

Thus, the aliases have been resolved, and attribute extraction can proceed. Using the resolved call statements the actual parameter lists are evaluated and attributes extracted.

3.1.2 Determining Placeholder Variables

To facilitate the identification of objects in the COMMON block, the first step is to determine, for each COMMON statement, which variables are actually used and which variables are serving as placeholders, i.e. variables not referenced or defined in the subroutine. This determination is done by analyzing the program unit over which the COMMON statement is valid (for each COMMON statement) using data reference analysis techniques. The algorithm to detect placeholders is given in Figure 3.3.

Let P be a FORTRAN program and let C be the COMMON area where

C = union of all C_j , $j = 0 .. k$ where

C_0 is unnamed

C_i is a uniquely named COMMON, $i = 1 ... k$.

Consider all references to the COMMON areas, and assume that there are m>=0 such references.

Let A_n , $n = 1..m$, be a reference to the common area, and

Let B_n , $n = 1..m$, be the program block over which A_n is valid

Let D_n , $n = 1..m$, be the set of placeholders of A_n .

Define ref[n] = the set of variables of A_n that are referenced in B_n , $n = 1..m$.

Define def[n] = the set of variables of A_n that are defined in B_n , $n = 1..m$

Then, Union(ref[n] , def[n]) = the set of all variables of A_n that are referenced directly in B_n, and

D_n = A_n \ Union(ref[n] , def[n]) = the set of all placeholders of A_n .

Figure 3.3
Algorithm to Determine Placeholders

The placeholders are determined using the use and defines lists. For each reference to a COMMON area, the union of both the use and defines lists is determined. This represents all COMMON variables that are not placeholders in the subroutine for that reference. Using set subtraction, the non-placeholder variables are subtracted from the list of all COMMON variables for the reference leaving the set of placeholder variables for the reference. Hence, by computing variable usage for each COMMON statement, and excluding the set of used variables from the set of all variables, the set of variables acting as placeholders is determined.

3.1.3 Resolving the Aliases

When a COMMON statement is used there is a risk of aliasing. Variables specified to be in the COMMON area are available to all program modules in which the corresponding COMMON statement appears. The problem of aliasing occurs because the variable names associated with the COMMON area may be different in the main program and each subroutine. Because the position of a variable in the COMMON statement determines the memory location to which it maps, and because unique variable names can be used in each module, it is possible that in each module, different names are given to a single memory location, i.e. aliasing. To accurately identify the objects, these aliases must be resolved. Therefore, the second step is to resolve the aliases to the COMMON block. There are essentially two approaches that can be taken to perform the resolution: (1) brute force and (2) computational.

3.1.3.1 The Brute Force Method

One method for resolving the aliases of the COMMON block is simply "brute force." This method enumerates each element of the COMMON statements and proceeds to match variables position by position. This method works, but is tedious, slow and not very elegant. Arrays must be enumerated and matched element by element.

Example:

Given the following primary COMMON statement:

```
COMMON   r1, r2, A[2, 3], r 3
```

and the following candidate COMMON statement:

```
COMMON   s1, s2, s3, s4, s5, B[2, 2]
```

enumerate as follows:

```
COMMON   r1, r2, A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3],
         r3
COMMON   s1, s2, s3, s4, s5, B[1,1], B[2,1], B[1,2], B[2,2]
```

and match the variables position by position:

(s1, r1), (s2, r2), (s3, A[1,1]), (s4, A[2,1]), (B[1,1], A[1,2]), ... etc.

3.1.3.2 The Computational Method

The alternative method provides a more elegant solution. This method begins matching positions and upon reaching an array, computes the offset required for its resolution. This method avoids the enumeration of each array.

Since the resolution is performed on a syntactically correct program, we can assume that there are no type conflicts in the COMMON statements since this would result in a syntax error. Let the COMMON statement in the main program serve as the COMMON statement to which all others will be resolved, and refer to it as the "primary COMMON statement." As resolution is being performed on a COMMON statement, it will be referred to as the "candidate COMMON statement." Once resolution is performed on a COMMON statement it is referred to as a "resolved COMMON statement."

Let the primary COMMON statement be of the form: COMMON v_1, v_2, ..., v_p.

Let the candidate COMMON statement be of the form: COMMON w_1, w_2, ..., w_c.

A function, f, that will map a variable from the candidate COMMON statement to a variable in the primary COMMON statement is defined using separation of case. There are four cases that must be considered in alias resolution of the COMMON block:

1. No arrays are present in the primary or candidate COMMON statements.
2. Arrays are present in the primary but not the candidate COMMON statement.
3. Arrays are present in the candidate but not the primary COMMON statement.
4. Arrays are present in both the primary and candidate COMMON statements.

Consider case 1, the case when there are no arrays present in either the primary COMMON statement or the candidate COMMON statement. This is the most trivial case in that each variable listed in the candidate COMMON statement will correspond directly to a variable listed in the primary COMMON statement, and, as discussed before, no type consideration is necessary. For all i, 1..p, v_i is not and array and for all j, 1..c, w_j is not an array. Then for k = 1..c, $f(w_k) = v_k$. Thus, the mapping is direct, i.e. the variables of the candidate COMMON statement are mapped to the variables in the primary COMMON statement that occupy corresponding positions in the statements.

Next, consider cases 2 and 3, when arrays appear in either the primary COMMON statement or the candidate COMMON statement but not both. When an nxm array is reached in the primary (resp. candidate) COMMON statement, mn consecutive locations are assigned in column major manner, i.e. (1,1) (2,1), (3,1) ... (1,2), (2, 2) etc. Map the array to nm locations in the candidate (resp. primary) COMMON statement. That is to say, if the array is in the primary COMMON statement, mn variables in the candidate COMMON statement will be mapped to the array name. If the array appears in the candidate COMMON statement, then the array will be renamed to the nm corresponding variables in the primary COMMON statement.

More specifically, consider case 2, when arrays are present in the primary COMMON statement. For some i, 1..p, v_i is an array, and for all j, 1..c, w_j is not an array. Then define a function g such that

$g(v_i)$ = 1 if and only if v_i is not an array and

= mn if and only if v_i is an nxm array,

Then $g(v_i)$ is the number of locations allocated in the primary COMMON statement. If v_i is an array, further note the elements of v_i as $v_i\{row, col\}$. Now consider w_k, the kth element in the candidate COMMON statement. Then find a z, z > 0, such that $g(v_i) = k$, if such a z exists. If there exists such a z, then $f(w_k) = v_z$ if v_z is not an array, and $f(w_k) = v_z\{n,m\}$ if v_z is an array. If no such z exists, then find y such that $g(v_i)$ = glb(k), where glb(k) is the greatest integer less than k.. Let l = glb(k) and define t = k-l. Since v_{y+1} is an array, we must define f such that w_k is mapped to a particular element of v_{y+1}. Let c = TRUNC((t-1),n) and let d = MOD((t-1),n). Then $f(w_k) = v_{y+1}\{d+1, c+1\}$.

Now, consider case 3, where arrays are present in the candidate COMMON statement, but not the primary COMMON statement. Let w_k be an nxm array and v_h is not an array. Then we must define f such that w_k is mapped to v_h, v_{h+1}, ..., v_{j+nm-1}. Therefore, $f(w_k\{i,j\}) = v_{(h-1) + (j-1)n + i}$.

Finally, consider case 4, when arrays appear in both the primary and candidate COMMON statements, two subcases are possible: (i) the arrays have equal dimension, (ii) the dimension of the array in the primary COMMON statement is greater than the dimension of the array in the candidate COMMON statement. We consider only the case where the array in the candidate COMMON statement does not exceed the boundaries of the array in the primary COMMON statement. If the array dimensions are equal, then this reduces to the trivial case. The mapping between the two arrays is direct. If the dimensions of the arrays do not match, specifically, if the dimension of the primary array is greater than the dimension of the candidate array, then the resolution becomes more complex.

Let A be an array in the primary COMMON statement with dimension mn, and let B be an array in the candidate COMMON statement with dimension ij such that $mn > ij$. In the general case, B is an alias to some ij locations of A, not necessarily beginning at A[1,1] or ending with A[n,m]. Thus, B may reference the ij "middle" locations of A, such that $r+ij+s = mn$ for some $0 < r <= mn$, $0 < s <= mn$. Pictorially,

```
|------------------ nm ---------------------|   array A[n,m]
|---r ----|------- ij -----------|--------- s -----|   array B[i,j] aliases a subset of A.
```

It is necessary to determine exactly which segment of A is aliased by array B because of the possibility of dummy variables representing a portion of A. The algorithm to resolve the COMMON block using the computational method is given in Figure 3.4. It should be noted that the resolution method involves two-dimensional arrays, but is easily adapted to one-dimensional arrays by letting $m = 1$, i.e. setting the number of columns equal to 1.

```
1. Determine the size of the array in the primary COMMON statement, i.e. mn, and call this value
       "size_primary"
2. Determine the size of the array in the candidate COMMON statement and call this value
       "size_candidate"
3. Compute x = TRUNC(r/n),    y = MOD(r/n)
4. The variables in the r locations alias A[n,x] A{1..y, x+1} (note: the notation { } is used to
       denote only a single column of an array, while [ ] is standard array notation).
5. Compute u = n - y,
       (u is the number of locations necessary to complete the column started by the r variables.)
6. If u < ij  then  A{y+1 .. n, x+1}  denotes the remainder of the column,
       (i.e. y+1...n is u rows of column x+1),
   Compute the following:
       c = TRUNC(ij-u /n), (c = the number of complete columns of A included in B)
       d = MOD(ij-u / n), (d = the remaining number of slots of A included in B)
   If c > 0  then  the remainder of B aliases A{n , (x+2) ... (x+2+c-1) } A{ d, (x+2+c) }
       else (c = 0) B aliases A{d, (x+2) }
7. If u = ij  then A{y+1..n, x+1} is the aliased positions of A, i.e. the remainder of the column
8. If u > ij      then  A{ (y+1) ... (y+1+ij), (x+1) } denotes the alias of array B,
       i.e. a portion of the x+1 column of A.
9. The next s variables in the candidate COMMON statement alias the next s locations of A.
       f(w_j) = A{(y+1)...n, (x+1)} A{n, (x+2)...(x+2+c-1)} A{d,(x+2+c)}
                       if and only if u < ij and c > 0
       f(w_j) = A{(y+1)...n, (x+1)} A{d, (x+2)}   if and only if  u < ij and c = 0
       f(w_j) = A{(y+1)...n, (x+1)}   if and only if  u = ij
       f(w_j) = A{(y+1)...(y+1+ij), (x+1)}  if and only if  u > ij
          and f(w_{j+1}) = A{d+1, x+2+c} if and only if  u < ij and c > 0
       f(w_{j+1}) = A{d+1, x+2} if and only if  u < ij and c = 0
       f(w_{j+1}) = A{1, x+2} if and only if  u = ij
       f(w_{j+1}) = A{(y+1+ij)+1, x+1} if and only if  u > ij
              and    f(w_{j+k}) = A{((d+k-1) mod n) +1, (x+2+c) + ((k-1)trunc n)+1} for k = 1...s
```

Figure 3.4
Algorithm for COMMON Block Resolution

Notation:

A{1..2, 3} represents the elements A[1,3], A[2,3]

A{2,3} represents the single element A[2,3]

A[2,2] represents the elements A[1,1], A[2,1] A[1,2] A[2,2]

A_n includes all COMMON statements, both named and unnamed. Since alias resolution must be performed on COMMON statements that access the same area, i.e. those with the same name or no name at all, it is necessary to partition A into k+1 disjoint sets where each set contains all COMMON statements that access the same area of the COMMON block. Thus, there will be (k+1) primary COMMON statements, one for each of the (k+1) sets. When performing the resolution, it is necessary to use the primary COMMON statement for the set to which the candidate COMMON statement belongs.

At this point, the preprocessing phase is completed. The placeholders have been determined and the aliases have been resolved. The source code is prepared for the object extraction phase.

3.2 Object Extraction

The Object Extraction phase is divided into two stages: Attribute Identification and Method Identification. Attribute Identification uses a data-driven bottom up approach to analyze two aspects of the source code: actual parameters and global variables. Method Identification uses a new variation of traditional program slicing techniques to extract methods based on an object's attributes. The output of the Object Extraction phase is a set of candidate objects that will serve as input to phase two, Class Abstraction.

An object, O, is identified as a two-tuple, (D,M) where D is the set of data items (or attributes) and M is the set of methods that act on those data items. Two possible approaches to object extraction are top-down or bottom-up. A top-down approach begins with all data elements contained in a single object and proceeds to divide the object into smaller objects. Although this approach has some merit, the margin of error is on the side of objects whose attribute sets are too small, i.e. they do not adequately represent a functionally cohesive unit. The bottom-up approach constructs objects by first determining the cohesive strength between each pair of data items and proceeds to form groupings based on levels of cohesion. Because a bottom-up approach results in objects that are highly cohesive, this is the approach taken by FORM.

The functionality of a program is viewed at three levels. At the top-level, the functionality is that of the entire program. This view is too coarse grained to directly aid in object extraction, however, it may provide some insight into the type of objects that may be formed. The second level concentrates on the functionality of the individual subroutines. The third, and most fine-grained view, considers the functionality of each line of code. The view of the subroutine as the unit of functionality is the approach taken in this work. The consideration of the subroutine as the unit of functionality of the analysis facilitates the evaluation of two aspects of program variables: actual parameters and global variables.

3.2.1 Attribute Extraction -- Actual Parameter Analysis

By considering each subroutine as a unit of functionality, the actual parameters are then necessary to perform the function of the given subroutine. Based on these guidelines, FORM seeks to obtain the largest set of parameters representing the strongest cohesive unit. The cohesive strength of a pair of parameters is measured by determining the frequency in which they are both necessary for the execution of a function (where the subroutine is the unit of functionality). Measuring the cohesion of a pair parameters for a given subroutine results in the consideration of three cases:

(i) both parameters are necessary,

(ii) only one parameter of the pair is necessary, and

(iii) neither parameter is necessary.

The cohesion value of a given pair of parameters is affected by each case as follows: case (i) increases the cohesive value, case (ii) decreases the cohesive value, and case (iii) does not affect the cohesive value. A cost function, i.e. a function that maps each pair of parameters to a real number, is defined for a pair of parameters i and j with respect to subroutine f. Using a greedy approach results in a cost function, c, that, for a given pair of parameters, weights the necessity of both parameters of the pair as a stronger condition than the necessity of only one parameter of the pair. Therefore, the cost function is increased by .2 when both parameters are necessary for the execution of f, and only reduced by .1 when only one parameter is necessary for the execution of f. By adjusting the amount c is increased or decreased, the "greediness" of the approach is determined. Thus, the cost function, c, for a pair of parameters i and j with respect to subroutine *f* is as follows:

$$c(i,j) = \begin{cases} c(i,j) & \text{if and only if neither i nor j is necessary for the execution of } f \\ c(i,j) + .2 & \text{if and only if both i and j are necessary for the execution of } f \\ c(i,j) - .1 & \text{if and only if either i or j (but not both) is necessary for the execution of } f \end{cases}$$

A bottom-up approach is used to construct a graph that maintains the value of the cost function, c. This graph is represented as a weighted adjacency matrix, M, i.e. an adjacency matrix whose entries are real numbers. M[i,j] is assigned a value based on the result of the cost function, c(i,j). Thus, the value assigned by the cost function c(i,j) is stored in M[i,j] and is proportional to the degree of functionality by which i and j are related. Once the matrix, M, is instantiated, a *threshold table* is determined. A *threshold table* consists of two columns: threshold value and data sets. The *threshold value* is a non-negative real number that is determined from M as follows: for each non-negative real number in M, add a row to the threshold table with that value as the threshold value. For each threshold value, corresponding data sets are computed using the transitive closure algorithm. From this table, a "desirable" threshold level is determined by the human reverse engineer. In selecting a desirable threshold level, the goal is to find the largest data sets with the strongest cohesion, thereby minimizing the number of singletons. By setting a threshold on the value necessary to be considered relevant, the set of data contained in an object is determined. One benefit of this representation is that it facilitates the consideration of various sets of objects based upon varying the threshold.

Step 1.	Resolve aliasing of local variables
Step 2.	Resolve aliasing of actual parameters
Step 3.	Perform parameter analysis on the call statements generated in Step 2 and generate a weighted adjacency matrix using the cost function, c.
Step 4.	Generate the threshold table.

Figure 3.5
General Algorithm for Attribute Extraction from Actual Parameters

The general algorithm for attribute extraction from actual parameters is shown in Figure 3.5. This four-step procedure inputs the raw source code and performs two resolution steps: resolve all local variables, and resolve the actual parameters. Once the raw source code has been resolved, the resolved source code is analyzed and a weighted adjacency matrix, M is generated using the algorithm in Figure 3.6. The threshold table is then generated from M using the transitive closure algorithm. It is at this point that the human reverse engineer determines an appropriate threshold level, thereby determining the data sets for the candidate

```
Let P be a structured FORTRAN program
With n subroutine calls
And m (distinct) actual parameters

(i) For i = 1 to n do
        CALLi = set of actual parameters of subroutine call I
        If subroutine i is a function
        Then CALLi = CALLi union with the function resultant of subroutine I
    od
(ii) Let G = (V,E) be a graph
        V = the arbitrary ordered set of actual parameters and function resultants,
            and denote the elements of V as v1, v2, ...vm /* note |V| = m */
        E = {}

(iii) M[1..m, 1..m] ARRAY of REAL              /* a weighted adjacency matrix */
    AP[1..n, 1..m] ARRAY of BOOLEAN /* n sets of actual parameters */
    /* Construct the graph, represented as a weighted adjacency matrix */
    /* Initially G consists of only a set of vertices with no edges */
    For i = 1 to m do
        For j =1 to m do
            M[i,j] = 0
        od
    od
    /* Initialize the sets of actual parameters; AP[i,j] = 1 if and only if vj is an element of CALLI

    For i = 1 to n do
        For j = 1 to m do
            If vj is an element of CALL I
            Then AP[i,j] = 1
            Else AP[i,j] = 0
        od
    od
    /* Perform the analysis on the sets AP */
    For i = 1 to n do
        For j = 1 to m do
            For k = j+1 to m do
                If AP[i,j] = 1 and AP[i,k] = 1          /* both parameters are necessary]
                Then M[j,k] = M[j,k] + 0.2
                Else if AP[i,j] = 0 and AP[j,k] = 0  /* inconclusive */
                    Then skip
                    Else M[j,k] = M[j,k] - 0.1           /* only one is 0 *
            od
        od
    od
(iii) The output is r<n connected graphs. The vertices of each connected graph represents
    the number of attributes for a distinct candidate object.
```

Figure 3.6
Attribute Extraction – Parameter Analysis

objects. Therefore, the result of the threshold analysis on actual parameters is a grouping of the actual parameters into data sets which will represent the attribute sets of candidate objects.

3.2.2 Attribute Extraction -- Global Variable Analysis

The approach to global variable analysis concentrates on the COMMON block in FORTRAN because special considerations must be made for FORTRAN that are not necessary in other imperative languages. Specifically, COMMON block resolution and the determination of placeholders is not necessary in imperative languages such as C, Pascal and COBOL. The approach described in the dissertation for attribute extraction based on global variable analysis generalizes to these languages readily by using scoping rules and location of global variable declaration. Therefore, the primary focus will be on the COMMON block in FORTRAN as it is the most involved.

By considering the subroutine as the unit of functionality, the COMMON variables referenced in the COMMON statement of a subroutine are necessary to perform the function of the given subroutine. Realizing this, FORM seeks to obtain the largest set of global variables that represent the strongest cohesive unit. The cohesive strength of a pair of COMMON variables is measured by determining the frequency in which they are both necessary for the execution of the subroutine. To measure the cohesion of a pair of COMMON variables, three cases must be considered:

(i) both COMMON variables are necessary,

(ii) only one COMMON variable is necessary, and

(iii) neither COMMON variable is necessary.

The value of the cohesive measure of a given pair of COMMON variables is affected by each case as follows: case (i) increases the value, case (ii) decreases the value, and case (iii) does not modify the value. The greedy approach taken by the algorithm results in a cost function that, for a given pair of variables, weights the necessity of both variables of a pair as a stronger condition than the necessity of only one variable of the pair. Therefore, the cost function, c, is increased by 0.2 when both parameters are necessary for the execution of the subroutine, and only decreased by 0.1 when only one variable is necessary for the execution of the subroutine. By adjusting the amount that c is increased or decreased, the "greediness" of the approach is determined. The values chosen are 0.1 and 0.2, and define the necessity of both variables to be twice as important as the necessity of only one. Thus, the cost function, c, for a pair of variables i and j with respect to COMMON statement *f* is as follows:

c(i,j) = c(i,j)	if and only if neither i nor j is necessary for the execution of *f*
c(i,j) + .2	if and only if both i and j are necessary for the execution of *f*
c(i,j) - .1	if and only if either i or j (but not both) is necessary for the execution of *f*.

The cohesive strength for the pair of variables is stored in a weighted adjacency matrix, M, where M[i,j] is assigned a real number value based on the result of the cost function c(i,j). Once the matrix M is instantiated, a threshold table is constructed. The threshold table consists of two columns: threshold value and data sets. The threshold value is a non-negative real number and is determined from M by creating a row in the threshold table for each non-negative value in M. Corresponding data sets are computed for each threshold value using the transitive closure algorithm. Based upon this threshold table, the human reverse engineer selects the appropriate threshold level. The corresponding data sets translate to attribute sets of candidate objects.

The general algorithm for attribute extraction based on global variable analysis is given in Figure 3.7. This four-step procedure inputs the raw source code and performs two resolution steps: resolution of placeholders in the COMMON statements and resolution of aliases in the COMMON statements. Once the resolution has been performed, the source code is analyzed and a weighted adjacency matrix, M, is generated using the algorithm in Figure 3.8. The threshold table is then generated from M using the transitive closure algorithm. At this point, the human reverse engineer determines an appropriate threshold level, thereby determining the attribute sets for the candidate objects. The result of the threshold analysis on global variables is a grouping of the global variables into data sets which will represent the attribute sets of the candidate objects.

```
Step 1.   Determine placeholders COMMON statements
Step 2.   Resolve aliasing of COMMON variables
Step 3.   Perform parameter analysis on the COMMON
          statements generated in Step 2 and
          generate a weighted adjacency matrix
          using the cost function, c.
Step 4.   Generate the threshold table.
```

Figure 3.7
General Algorithm for Attribute Extraction from COMMON variables

At this point, the attribute extraction phase is completed. Actual parameters and global variables have been evaluated independently and attribute sets determined. Actual parameter analysis and global variable analysis result in threshold tables that are evaluated by a human reverse engineer to determine attribute sets for candidate objects. Note that the threshold analysis for actual parameters and global variables is performed independently because what may be a desirable threshold value for actual parameters is not necessarily a desirable threshold value for global variables.

The object extraction phase is not yet complete. Recall the definition of an object, O = (D,M). To this point, the data sets, D, of the candidate objects have been defined. Thus, the final step in object extraction is the extraction of the methods, M, that correspond with the data sets, D, thereby producing the candidate objects.

```
Let     P   be a structured FORTRAN program
With    n   COMMON statements
And         m   distinct variables in the COMMON area
(i) Preprocessing Phase
    a. Resolve the aliases of the COMMON block
    b. Determine R_I for each B_i
(ii) Let G = (V,E) be a graph
        V = the arbitrary ordered set of all variables in the COMMON area,
            and denote the elements of V as v1, v2, ...vm  /* note |V| = m */
        E = {}

(iii) M[1..m, 1..m] ARRAY of REAL                /* a weighted adjacency matrix */
    AP[1..n, 1..m]  ARRAY of BOOLEAN /* n sets of COMMON variables */
    /* Construct the graph, represented as a weighted adjacency matrix */
    /* Initially G consists of only a set of vertices with no edges */
    For i = 1 to m do
        For j =1 to m do
            M[i,j] = 0
        od
    od
    /* Initialize the sets of COMMON variables; C[i,j] = 1 if and only if vj is an element of  R_I

    For  i = 1 to n do
        For j = 1 to m do
            If vj is an element of  R_I
            Then C[i,j] = 1
            Else C[i,j] = 0
        od
    od
    /* Perform the analysis on the sets C */
    For i = 1 to n do
        For j = 1 to m do
            For k = j+1 to m do
                If  C[i,j] = 1 and C[i,k] = 1              /* both parameters are necessary]
                Then M[j,k] = M[j,k] + 0.2
                Else if  C[i,j] = 0 and C[j,k] = 0     /* inconclusive */
                    Then skip
                    Else M[j,k] = M[j,k] - 0.1             /* only one is 0 *
            od
        od
    od
(iii) The output is r<n connected graphs. The vertices of each connected graph represents
    the number of attributes for a distinct candidate object.
```

Figure 3.8
Attribute Extraction – COMMON Variable Analysis

3.2.3 Method Extraction

The procedure for extracting methods from the source code identifies statements that modify at least one of the attributes of the object to which it belongs. One approach is to first look to extract methods with no concern of the grouping of data sets to form object attributes. In this approach, the focus is on obtaining groups of statements that constitute some unit of functionality and the entire object extraction procedure is then driven by the method extraction. While this approach has some merit, to determine reasonable functionality would require extensive domain knowledge. Thus, the degree of generality for such an approach is very low in that the methodology itself would require major adaptations to adequately extract objects from various application domains. FORM takes the alternative approach.

The approach to method extraction is a data-driven approach and has a much higher degree of generality. Analysis is performed on the source code and results in data groupings that correspond to attributes of candidate objects. The nature of the analysis is such that it is domain independent but language dependent, however, in most cases this language dependence generalizes further to paradigm dependence. That is, the analysis is performed on a given language (e.g. FORTRAN, COBOL, C, Pascal, etc.) or a class of languages (i.e. imperative) without prior knowledge of the application domain. Thus, methods are extracted following the extraction of data sets. Moreover, the data sets drive the method extraction algorithm. By performing the object extraction in this manner, a large part of the extraction can be automated and the knowledge of a domain expert utilized for refinement.

The theoretical foundation of the method extraction procedure is program slicing [Weis 84]. Program slicing is defined as a "decomposition based on data flow and control analysis [Weis 84]." A slicing criterion is defined as the tuple C = <i, V> where "i" is a statement number and V is a set of variables. R(0,C,n) is defined as the set of relevant variables at statement "n" and is defined as all variables "v" such that:

1. n = i and v is in V, or
2. n is an immediate predecessor of a node m such that either
    a. v is in REF(n) and there is a w in both DEF(n) and R(0,C,m), i.e. w is relevant at statement m and v is used to define w at the previous statement. Thus, if w is a relevant variable at the node following n, and w is given a new value at n, then w is no longer relevant and all the variables used to define w's value are relevant, or
    b. v is not in DEF(n) and v is in R(0,C,m), i.e. v is relevant at statement m and is not defined in the previous statement. Thus, if a relevant variable at the next node is not given a value at node n, then it is still relevant at node n.

Then S(0,C) is a set of statements, i.e. the slice where S(0,C) is all nodes n such that R(0, C, n+1) intersect DEF(n) is not empty. Thus, if a relevant variable is defined in a previous statement, than that statement is included in the slice. Additionally, any branch statement which can choose to execute or not execute some statement in S(0,C) should also be included in the slice [Weis 84].

To use program slicing for method extraction, we defined extensions to the program slicing in [Weis 84]. First, the slicing criterion, C, is defined as C = <M,V> where M is a module, or block of statements, i.e. a subroutine or the main program, and V is the set of attributes of a given object. Recall that once the appropriate threshold value is chosen, the corresponding data sets represent attributes of candidate objects. These data sets are then used to drive the method extraction. For each data set, program slicing is performed on a subset of the modules and each resultant slice becomes a method in the corresponding object. Because the data sets are derived from actual parameters or global variables, the method extraction process must be performed with variation in each case. Specifically, when slicing for methods corresponding to actual parameters, consideration must be given to the formal parameter mapping based upon the CALL statements in which the actual parameters appear. Such consideration is not necessary when slicing for methods corresponding to global variables.

Method extraction for data sets comprised of actual parameters is driven by the system's CALL statements. Let K be the set of all CALL statements of the system. Then for a given data set, D, form the set P, such that P contains all CALL statements that contain one or more elements of D as an actual parameter. For each element of P, define the slicing criterion C = <M,V> such that M is the called subroutine and V is the set of formal parameters that correspond to the elements of D appearing in the CALL statement as actual parameters. Note that because P is a set and not a bag, redundancy is eliminated. Thus, each data set D has at most |P| methods, i.e. one method for each element of P.

```
Let n =    the number of data sets comprised
                of COMMON variables, and
    m =    the number of subroutines in the
                system.

Then,
    For i = 1 to n do
           For j = 1 to m do
                  Compute the slice on <$M_j$, $D_i$> and let
                    Method j of object i equal this slice.
           od
    od
```

Figure 3.9
Method Extraction Algorithm for Data Sets Comprised of COMMON Variables

Method extraction for data sets comprised of global variables is more straightforward than for actual parameters. Each subroutine in the system is sliced on the criterion C = <M,V> where M is the module and V is the set of COMMON variables in the data set. Thus for data sets comprised of COMMON variables, the extraction algorithm is given in Figure 3.9. For each object there are at most m methods which are defined by using program slicing on each subroutine. Using each data set that is comprised of COMMON variables as a variable set for a slicing criterion and each subroutine as the statement set for a slicing criterion, the methods are extracted.

It is important to realize that the method extraction procedure in both the actual parameters and COMMON variables does not attempt to assign semantic names to the methods. Although this may appear a shortcoming at first, a deeper investigation into the area reveals that by not attempting to semantically qualify the methods, the scope of the methodology is much broader. That is, an objective of the methodology is generality across many application domains. The assignment of semantic naming would require a great amount of domain specific knowledge and, therefore, restrict the generality of the methodology as a whole.

At this point, the first phase, Object Extraction, of the methodology is completed. The result of this phase is a set of candidate objects. These candidate objects serve as input to the second phase, Class Abstraction.

3.3 Class Abstraction

In phase one, Object Extraction, candidate objects were extracted from the source code. Phase two, Class Abstraction evaluates the candidate objects extracted in phase one, and clusters the candidate objects to form classes. To facilitate the clustering process, the candidate objects are analyzed to obtain the following information: number of attributes, number of methods, type signature of the attributes, and use and defines lists of each method. Clusters are formed from the set of candidate objects based on an identity measure. The classes are determined by mapping each object cluster to a unique class.

Definition 3.1: Identical Objects

Two objects O_1 and O_2 are defined to be *identical objects* when the following criteria is satisfied:

1. The number of attributes of O_1 equals the number of attributes of O_2.
2. The number of methods of O_1 equals the number of methods of O_2.
3. The attribute type signature of O_1 is identical to the attribute type signature of O_2.
4. The subroutines of derivation of O_1 are identical to the subroutines of derivation of O_2.
5. Corresponding methods of O_1 and O_2 differ only where attribute names are concerned.

If all five criteria are met, the two objects are said to be identical and are clustered together. This determination is made for each pair of objects. Thus, for n objects, at most C(n,2) tests must be performed because the definition of identical objects is transitive and partitions the set of objects into equivalence classes. Additionally, at least (n-1) tests must be performed. The upperbound C(n,2) occurs when at most two objects are grouped together and those two are the last two tested. The lowerbound of (n-1) occurs when a single equivalence class is formed that contains all objects, i.e., all objects are identical and a single cluster is formed.

Once the set of objects has been partitioned into equivalence classes, the mapping to classes is straightforward. Each equivalence class maps to a unique class in the object model. Thus, the mapping between object equivalence classes and classes in the object model is one-to-one and onto, i.e. isomorphic.

Lemma 3.1 Let C be the set of classes in the object model, E be the set of object equivalence classes, and O be the set of objects. Then $|C| = |E| <= |O|$.

Proof:

By definition, each element of E (an object equivalence class) maps to one and only one element of C (a class in the object model). Assume $|C| <> |E|$. Then either some element of E maps to more than one element of C ($|C| > |E|$) or some element of E maps to no element of C ($|C| < |E|$). But this contradicts the definition, therefore, $|C| = |E|$.

By definition, each element of O maps to one and only one element of E. Assume $|E| > |O|$. Then some element of O maps to more than one element of E. but this contradicts the definition, therefore $|E| <= |O|$.

Therefore, $|C| < |E| <= |O|$. (QED)

3.3.1 Attribute Definition

Once the equivalence classes have been defined and mapped to classes in the object model, it is necessary to determine the set of attributes for each class. Each object in the equivalence class has a unique set of attributes. Because the objects of an equivalence class are identical, the attributes of the objects in that class are identical by Definition 3.1. Using this definition, it is possible to abstract attributes for the class.

Consider some object equivalence class, E, and the corresponding class in the object model, C. Let O be some object in E and represent the number of attributes of O as na(O). By Definition 3.1, every object in E has an equal number of attributes, so denote the number of attributes of any object in E as na(E). Then, the corresponding class, C, will have na(E) attributes. Because no consideration is given to the semantic nature of these attributes, they are to be designated as att_1, att_2, att_3, .., $att_{na(E)}$.

Once the attributes are determined, it is necessary to determine the type of each attribute. Let O be some object in E, and denote the type signature of the attributes of O as ts(O). Since every object in E has an identical attribute signature by Definition 3.1, denote the attribute type signature of any object in as ts(E). Thus, the corresponding class, C, will have type signature ts(E). Thus, careful definition of identical objects and properly clustering the objects in such a way that the clusters represent equivalence classes, the procedure to define attributes of the classes is quite elegant.

3.3.2 Method Definition

The final step in the class abstraction phase is to abstract a set of class methods from the cluster of objects that corresponds to the class. The definition of methods of a class requires the consideration of two factors: number of methods and method content. That is, for each class the first step in the method definition procedure is to determine the number of methods the class will contain and the second procedure is to instantiate the methods. Again, careful definition of identical objects and properly forming the object clusters has served to reduce the effort required to define the methods of a class.

Lemma 3.2 Let E be some object equivalence class, C be the class in the object model that corresponds to E and O be some object in E. Denote the number of methods of O as nm(O). Further, denote the number of methods of E as nm(E) an the number of methods of C as nm(C). Then nm(O) = nm(E) = nm(C).

Proof:

nm(O) = nm(E): By Definition 3.1, identical objects have an equivalent number of methods. Consider some object equivalence class, E. Then, E is formed through a mapping of clusters of identical objects. Moreover, the mapping of object clusters to object equivalence classes has been shown to be isomorphic. Therefore, E maps to a single object cluster containing objects which all have an equivalent number of methods. Therefore, nm(O) = nm(E).

nm(E) = nm(C): Each object equivalence class, E, maps directly to one and only one object class C. Specifically, each method in E maps to a method of C. Therefore, nm(E) = nm(C).

Therefore, nm(O) = nm(E) = nm(C). (Q.E.D.)

By Lemma 3.2, each class, C, contains exactly the number of methods of its corresponding object equivalence class. Each method is then denoted as Method1, Method2, ...Method*nm(C)*. Hence, the first step in method definition, determination of the number of methods of an object class, is straightforward. The next step in the method definition procedure is to instantiate the methods.

Consider an object equivalence class, E. Then, the objects contained in the cluster of objects, O, that maps to E are identical and by Definition 3.1, their methods differ only where attribute names are concerned. So, consider some object *o* in O. Then, each method of *o* maps to a method in C such that the attribute variable names in the methods of *o* are replaced with the corresponding attribute variable names of C.

Upon completion of the method definition procedure, the Class Abstraction phase of FORM is completed. At this point, objects have been identified and classes in the object model have been abstracted. At this point we have an object oriented design that consists of a single-level hierarchy. That is, no parent classes are defined for the class structure. Therefore, the next phase of the methodology is to define parent classes by Abstraction of the Inheritance Hierarchy.

3.4 Abstraction of the Inheritance Hierarchy

Abstraction of the Inheritance Hierarchy involves clustering the classes of the object model based on similarity measures. Because the classes have two components (attributes and methods) two similarity measures are discussed: attribute-based similarity and method-based similarity. The flexability and generality of the method then allows the human reverse engineer to determine how these similarity measures may be used to best serve the needs of the project. Thus, the human reverse engineer may choose to utilize the information generated by only one similarity measure, or perform the procedure to synthesize the information of both measures. Again, in an effort to preserve the generality of the methodology, the decision of how best to utilize the information generated is left to the human reverse engineer.

The similarity analysis is performed using the five criteria considered for class formation: number of attributes, number of methods, type signature of attributes, and subroutine of derivation of the methods. Consider a class, C, with na(C) attributes and nm(C) methods. Then the type signature of the attributes of C is represented as a bag, B_1, whose elements are the types of the attributes of C. Moreover, $|B_1|$ = na(C). Further, the subroutines of derivation of the methods of C are represented as a bag, B_2, whose elements are the names of the subroutines that were sliced to generate the methods of the class. Additionally, $|B_2|$ = nm(C). This criteria is used to perform the two types of analysis on the classes: attribute-based similarity and method-based similarity. The result of this analysis is a weighted adjacency matrix that is further analyzed to produce a threshold table. Based upon this threshold table, the set of classes is partitioned into disjoint sets of clusters which map to the parent classes of the inheritance hierarchy.

3.4.1 Attribute-Based Similarity

The attribute-based similarity measure determines the similarity of a pair of classes based on the similarity of their attributes. For a pair of classes, a similarity measure is assigned based on two characteristics of the classes' attributes: number and type signature. This value is stored in a weighted adjacency matrix from which a threshold table is constructed using the transitive closure algorithm. The algorithm for computing the attribute-based similarity measure for each pair of classes in the object model is given in Figure 3.10.

Let n be the number of classes in the object model.

Let M be a nXn weighted adjacency matrix whose rows and columns represent the classes of the object model.
Define R to be the row vector such that R_i is the ith class of the object model.
Define L to be the column vector such that L_j is the jth class of the object model.
Let na(C) represent the number of attributes of a class C.
Let as(C) represent the attribute type signature of a class C.

```
1.  Initialize M to the zero matrix.
2.  For i = n do
3.      For j = i+1 to n do
4.          If na(Ri) = na(L_j)
5.          then M[i,j] = M[i,j] + 0.2
6.          else M[i,j]  = M[i,j] - 0.1.
7.          fi
8.      id = |as(R_i) intersect as(L_j)|
9.      sim = id / max(na(R_i), na(L_j))
10.     M[i,j] = M[i,j] + sim
11.     od
12. od
```

Figure 3.10
Algorithm to Compute Attribute-Based Similarity

The algorithm for computing the attribute-based similarity evaluates pairs of classes for similarity in number of attributes by determining if the classes have the same number of attributes, or they do not, no consideration is given to the difference in the number of attributes. This approach is taken because counting the number of attributes does not give information as to how they are used. However, because objects can not be considered identical without having the same number of attributes, it is a necessary consideration. Evaluation of the attribute type is used to provide information on how the attributes may be used. By considering the type signature of the attributes of a class, it is possible to ascertain a similarity relationship between classes. Additionally, it is important to realize that the similarity measure is not binary, but serves to provide a measurement of the degree of similarity. Through the use of the thresholding concept, the human reverse engineer may determine a desirable degree of similarity.

3.4.2 Method-Based Similarity

The method-based similarity measure determines the similarity of a pair of classes based on the similarity of their methods. For a pair of classes, a similarity measure is assigned based on two characteristics of the classes' methods: number and subroutine of derivation. This value is stored in a weighted adjacency matrix from which a threshold table is constructed using the transitive closure algorithm. The algorithm for computing the method-based similarity measure for each pair of classes in the object model is given in Figure 3.11.

The algorithm for computing the method-based similarity evaluates pairs of classes for similarity in number of methods by determining if the classes have the same number of methods. This is a binary consideration, that is either the classes have the same number of methods, or they do not, no consideration is given to the difference in the number of methods. This approach is taken because counting the

```
Let n be the number of classes in the object model.
    Let M be a nXn weighted adjacency matrix whose rows and columns
    represent the classes of the object model.
    Define R to be the row vector such that R_i is the ith class of the object
    model.
    Define L to be the column vector such that L_j is the jth class of the
    object model.
    Let nn(C) represent the number of methods of a class C.
    Let sd(C) represent the subroutine of derivation of the methods
    of a class C.

    1.  Initialize M to the zero matrix.
    2.  For i = n do
    3.      For j = i+1 to n do
    4.          If nm(Ri) = nm(L_j)
    5.          then  M[i,j] = M[i,j] + 0.2
    6.          else M[i,j]  = M[i,j] - 0.1.
    7.          fi
    8.      id = |sd(R_i) intersect sd(L_j)|
    9.      sim = id / max(sd(R_i), sd(L_j))
    10.     M[i,j] = M[i,j] + sim
    11.     od
    12. od
```

Figure 3.11
Algorithm to Compute Method-Based Similarity

number of methods does not give information as to their functionality. However, because objects can not be considered identical without having the same number of methods, it is a necessary consideration. Evaluation of the subroutine of derivation is used to provide information the functionality of the methods. By considering the subroutine of derivation of the methods of a class, it is possible to ascertain a similarity relationship between classes. As with attribute-based similarity, the method-based similarity measure is not binary, but serves to provide a measurement of the degree of similarity. Additionally, the thresholding concept allows the human reverse engineer to determine a desirable degree of similarity

3.4.3 Combining Threshold Tables

Upon completion of the attribute and method based similarity measures, the human reverse engineer then determines the most appropriate use of these methods for their particular case. The methodology affords them several options: emphasize the attribute-based similarity measure, or emphasize the method-based similarity measure. This results in the necessity to synthesize the information contained in the separate threshold tables.

To combine the information of both threshold tables, each table is first considered independently and a threshold value is chosen for each table. The result is two sets of class clusters, one for attribute-based similarity, AC, and a second for method-based similarity, MC. These two clusters are combined to form a single cluster, CC. The clustering algorithm is given in Figure 3.12.

This analysis results in the set CC of parent classes. The inheritance hierarchy is formed by creating a parent class as follows: for every powerset PS in CC, create a parent class whose child classes are the elements of the powerset. The abstraction of the inheritance hierarchy is the last step in the methodology to extract the object model. The final phase of the process involves the representation of the object model.

```
1.  Let U be the universe of all classes in the object model.
2.  Let FINISHED be a set that is initially empty
3.  For each element e in U
4.      if e is a singleton in AC, and e is a singleton in MC then
5.          e is a singleton in CC
6.          add e to FINISHED
7.      else
8.          for each element f that is clustered with e in both AC and MC do
9.              cluster e and f in OC
10.             add e and f to the set FINISHED
11.         od
12.     endif
13.  for each element e in U/FINISHED do
14.     if e is a clustered with f in (AC or MC)
                          // the reverse engineer choose AC or MC
15.     then  cluster e and f in OC
16.     else cluster e as a singleton in OC
17.     endif
18.  od
19.  od
```

Figure 3.12
Clustering Algorithm

3.5 Representation of the Object Model

The representation of the object model marks the final phase in the methodology. The extracted object-oriented design is represented in two types of diagrams and several supporting documents. The two categories of diagrams are the hierarchy diagrams and the interaction diagrams. The hierarchy diagrams represent the static elements of the design and include such documents as the object templates, class templates, and the object/class mapping diagram. The interaction diagrams describe the dynamic elements of the design and include such documents as the method invocation diagram and the state interrogation diagram.

3.5.1 The Hierarchy Diagrams

The first type of diagrams is the hierarchy diagrams. The hierarchy diagrams represent the inheritance hierarchy. These diagrams depict a data view of the system. The hierarchy diagrams consist of a set of object templates, class templates and an object/class mapping diagram.

The object template contains all pertinent information about each object in the system. For each object extracted from the system, there is a corresponding object template. The object template contains the object name, attribute names and types, method interface and subroutines of derivation of the methods. The object template is given in Figure 3.13.

ObjectName **is**

Attributes attribute1:type1, ... attributen:typei

Methods method1, method2, ..., methodj

SubDerivation subroutine1, subroutine2,..., subroutinek

Figure 3.13
Object Template

The class templates are similar to the object templates, but represent the result of the abstraction of objects into classes. The class template contains the class name, parent class(es), attribute names and types, and method interface. The object/class mapping diagram is incorporated into the class template and describes which objects are clustered to form classes. The class template is given in Figure 3.14.

ClassName **is** {object_name1, object_name2, etc.}

ParentClass is {ParentClass}

Attributes attribute1: type1, .., attributei: typei

Methods method1, method2, ..., methodj

Figure 3.14
Class Template

3.5.2 The Interaction Diagrams

The interaction diagrams describe class interactions and introduce the element of control flow into the object model. There are two considerations in forming the interaction diagrams: (a) Call statements in subroutines possibly perform a computation and pass information back to the calling subroutine. This information is possibly used elsewhere in the calling subroutine. This is preserved in the object model as a message passed to an object with the corresponding method. (b) Messages may need to be passed to an object to interrogate state information. These two considerations define under what circumstances messages are passed between objects, specifically, method invocation and state interrogation.

The method invocation diagram is related to subroutine invocation in FORTRAN. In FORTRAN, a call statement invokes a subroutine, while, in the object model, methodA passes a method to methodB. MethodA will be determined by which subroutine the call statement appears. This is done by the following procedure: (1) determine the subroutine of invocation of the call statement, (2) determine which objects have a method that was derived from the given subroutine, (3) determine the information returned by the call statement, (4) determine which of the objects in #2 use the information returned by the call statement. Each of these objects is an object that initiates the message, i.e. methodA. MethodB is determined as follows: (1) determine which subroutine is invoked by the call statement, (2) determine which methods of each object have that subroutine as the subroutine of derivation, and (3) each of the corresponding objects will be the object to which the message is passed. The method invocation template is given in Figure 3.15.

```
ClassName
          MethodName invokes ClassName.methodname1
                ...
          MethodName invokes ClassName.methodnamej
```

Figure 3.15
Method Invocation Template

The state interrogation diagram represents which messages are passed among the objects to interrogate state information. If objectA references a variable which is not an attribute or local variable of objectA, then it is referencing an attribute of another object. Therefore, a message must be sent to the object containing the variable as an attribute. Each method of every object must be evaluated. Because there are no global variables in the object model as were in FORTRAN, each reference to a COMMON variable will translate to a state interrogation method. The state interrogation template is given in Figure 3.16.

```
ClassName
          MethodNamen  interrogates ClassNamei.attributej
                  . . .
          MethodNamem  interrogates ClassNamek.attributez
```

Figure 3.16
State Interrogation Template

This collection of diagrams comprises the representation of the object model as extracted from the source code. These diagrams represent both static and dynamic properties of the system. The diagrams are object-oriented and represent the system at a higher level of abstraction than the original source code. This representation facilitates the modification of the system's properties at a higher level of abstraction in an effort to reduce future maintenance costs. Moreover, because this is a system design, there are no restrictions placed on the choice of an implementation language.

3.6 Validation and Evaluation of FORM

Software quality is a multifaceted concept that can be described from different perspectives. Five perspectives, as in [Kitc 96] are the transcendental view, user view, manufacturing view, product view, and value-based view. The definition of software quality is based upon the perspective taken. However, these views do not address the design phase of the software lifecycle. Moreover, the issues pertinent to forward engineering do not necessarily parallel those pertinent to reverse engineering. In the forward engineering process, a design is developed based upon a requirements specification which details user expectations and needs. Thus, we expect and require that the design meets the requirements specification, thereby meeting user needs. The quality of a forward engineered design can be discussed in several contexts, such as how well the design represents the requirements specification, or how the design evaluates using a given set of metrics, as in [Li 95]. In reverse engineering, however, this is not the case.

The reverse engineering process begins with source code that has undergone extensive maintenance and no longer satisfies the user needs. Rather than continue to nurse an antiquated system, the decision is made to extract the functionality of the system and utilize new technologies while making the required changes at a higher level of abstraction, thereby improving overall system quality, decreasing future maintenance costs, and satisfying current user needs. Hence, to measure the quality of a reverse engineered design in terms of current user needs is not reasonable. The reverse engineered design necessarily does not satisfy user needs. That was the primary motivation for reverse engineering the system. Therefore, when evaluating the quality of a reverse engineered design, one cannot simply rely on traditional methods.

There are numerous research efforts which present methodologies to extract an object-oriented design from imperative code [Liu 90; Liva 92; Lano 93; Ong 93; Subr 96]. Little discussion, however, is given to the issues involved in evaluating the quality of an object-oriented design extracted from an imperative language because the issues surrounding reverse-engineered designs differ from those involved in evaluating a forward-engineered design.

3.6.1 Traditional View of Software Quality

To discuss the quality of a software design, whether forward or reverse engineered, the concept of software quality must be defined. As stated in [Kitc 96], the definition of software quality is based on the perspective taken -- transcendental, user, manufacturing, product, and value-based. The transcendental view sees quality as an unattainable goal because it can be recognized, but never completely defined. The user view is a very personal and concrete view of how the product meets the user's needs. The manufacturing view focuses on whether or not the right product was produced, i.e. it is based on "document what you do and do what you say." The product view considers the inherent characteristics of the product. The value-based view measures quality by what the user is willing to pay for. Based on the perspective taken, a single product's quality measure may evaluate very differently. This is the underlying issue of the anomalous results caused by using forward engineering metrics to measure quality in a reverse engineered design.

The traditional discussion of a "good" design takes the product view. A "good" design is classified as having the following characteristics: modifiability, modularity, levels of abstraction, loose coupling, high cohesion. Moreover, a good design should support understanding and automation. [Pfle 91, Jone 90] Because, "there is no definitive way of establishing what is meant by a "good" design," (e.g., allowing efficient code to be produced, a minimal design whose implementation is maximally compact, or a design that has maximal maintainability) no single metric will suffice, hence, a suite of metrics for measuring the quality of object-oriented designs must be considered. [Somm 89, Chid 94]

The following metrics, discussed fully in [Li 95] and [Chid 94], were developed to measure the quality of object-oriented designs because metrics related to procedural systems are unable to characterize the concepts of inheritance, classes and message passing (the core concepts of the object-oriented paradigm): depth of the inheritance tree, number of children, response for a class, and lack of cohesion in methods [Li 95, Chid 94]. Each of these metrics is discussed individually, explaining any anomalous values that result in the context of reverse engineering.

Two metrics are related to the inheritance hierarchy, and, therefore, the scope of properties: depth of inheritance and number of children. The ***depth of the inheritance tree (DIT)*** for a class is the length of the maximum path from the root node to the class node. It defines the depth of the class in the inheritance tree. Therefore, a derived (sub)class has a DIT value equal to the DIT of its parent class + 1. The DIT represents the tradeoff between reusability through inheritance and simplicity of system understanding. The *number of children (NOC)* of a class is the number of immediate descendants of a class. These metrics are collected from a reverse-engineered object-oriented design in the same manner as they are collected from a forward-engineered design. During the reverse engineering process, the criteria used for the formation of the inheritance hierarchy may predetermine the depth of the inheritance tree. Unless some threshold value is determined such that the abstraction of the inheritance hierarchy continues until the threshold is reached, the maximum DIT will be determined before the process ever begins. Therefore, the maximum scope of properties of a class is dependent on the methodology. [Li 95, Chid 94]

The *response for a class (RFC)* is defined as the number of methods that can be invoked in response to a message that is sent to an object of the class. This includes not only methods within the class, but also external methods. Because objects communicate using message passing, the RFC uses the methods as a measure of communication and, therefore, an indication of the effort required to test and debug an object. In the context of reverse engineering, one factor that will greatly inflate the value of RFC is the use of global variables. Excessive use of global variables will result in increased message passing and method invocation, thereby inflating the RFC. [Li 95, Chid 94]

The *lack of cohesion in methods (LCOM)* is the number of disjoint sets of local methods. The local methods are grouped into sets such that in each set, methods in the set share an instance variable with at least one other member of the set. Hence, a highly cohesive class will have many methods that share instance variables and therefore, a low LCOM value. This promotes encapsulation, and reduces complexity. [Li 95, Chid 94] In the context of reverse engineering, the classes will necessarily have a high degree of cohesion based on the methodology used for object extraction. The methodology defined in [Ache 94] and [Ache 95] results in classes that are highly cohesive by extracting the objects based on the cohesive strength of the parameters.

The four metrics described above were designed to evaluate a forward-engineered object-oriented design. They quantitatively measure the quality of the design based on coupling (DIT, NOC), cohesion (LCOM), abstraction (DIT, NOC), modifiability (RFC, LCOM), and modularity (DIT, NOC). Each of these metrics measures the quality of the design based on the product view. Because the design is forward engineered, these criteria can be used to drive the development and, therefore provide a reasonable yardstick to measure the resulting design. In the context of reverse engineering, however, the constraints set forth by the source code may make it unreasonable to expect to achieve the same design quality when measured by these forward-engineering metrics. For this particular reason, an alternative view is recommended for the evaluation of the quality of reverse engineered designs.

3.6.2 Software Quality – Reverse Engineering Perspective

Because reverse engineering does not involve making modifications to the extracted design, the designs that are reverse engineered from legacy systems will necessarily evaluate less than optimally using current metrics. However, when reengineering activities begin, (i.e., the reverse-engineered design is forward engineered to meet the new system requirements), the extracted design will be modified to improve the overall quality as per the product view. Thus, evaluating the reverse-engineered design requires that a different perspective be adopted.

Based on the definition of reverse engineering, the goal of the reverse engineering process is to extract the design of the system under evaluation. This means that the view most appropriate to evaluating the quality of the extracted design is the manufacturing view. Therefore, the extracted design is to be evaluated using the manufacturing view and the reengineered design is to be evaluated using the product view. By evaluating the extracted design using the manufacturing view, the reverse-engineer can be certain that qualities such as functional equivalence are maintained while giving less attention to such qualities as cohesion and coupling. Likewise, once the reverse-engineered design is reengineered to meet the new requirements of the system, the design is then evaluated using the product view, thereby focusing attention on measuring such qualities as coupling and cohesion which become vital considerations.

To evaluate the system under the manufacturing view, two criteria are presented: statement coverage and functional equivalence. Although neither of these are new metrics, the consideration in the context of reverse engineering is novel.

Let S1 be the source code of the legacy system to be reverse-engineered, and let D1 be the reverse-engineered design. Then, the *statement coverage* of the D1 is equal to the number of statements of S1 mapped into methods of D1, divided by the total number of statements of S1.

The statement coverage value is a direct computation. It provides insight into functional equivalence and may also aid in the detection of dead code. If the statement coverage is not 100%, it implies one of two cases: (i) some of the system's functionality may not be preserved or (ii) the statements not extracted represent dead code. In either case, it provides the reverse engineer with valuable information that will assist the reengineering process. Case (i) implies that the reverse engineering process itself must be reconsidered whereas case (ii) represents the detection of dead code. It is not the case, however, that 100% statement coverage implies functional equivalence, and vice versa. Therefore, both measurements are necessary.

Consider two software systems S1 and S2, where S1 is the legacy system to be reengineered and S2 is the implementation of the reverse-engineered design of S1. The design of S2 is said to be *functionally equivalent* to S1 if and only if when S1 and S2 are run on the same input, the resultant outputs are identical. This concept of functional equivalence is most important. It answers the question "did we extract the right design." There is a disadvantage, however, associated with the determination of functional equivalence. To make the determination of functional equivalence requires that the reverse engineered design be implemented, thereby providing the opportunity for errors in the translation from design to code. Hence, a functionally equivalent design that is incorrectly translated may appear functionally unequivalent, and vice versa. Until developments are made towards guaranteeing 100% correctness in design implementation, this measurement will be dependent on the skill of the programmer that implements the design.

Two case studies were performed on FORTRAN systems. System A evaluated to 100% statement coverage with functional equivalence. System B evaluated to 94% statement coverage, as a result of dead code, with functional equivalence.

3.7 Summary and Concluding Remarks

The FORTRAN Object Recovery Methodology (FORM) is a three phase methodology that extracts an object-oriented design from imperative code. Because FORTRAN is chosen as the source language, special considerations were required, thereby resulting in the development of an extensive pre-processing phase. The pre-processing phase includes alias resolution and determining placeholder values resulting from the use of the COMMON block.

The phases of the methodology include object extraction, class abstraction and abstraction of the inheritance hierarchy (Figure 3.4). Algorithms are given for each phase. The representation of the resulting object-oriented design required the development of a series of diagrams to adequately express the features unique to reverse-engineered designs.

The evaluation of reverse-engineered designs require considerations unique from those of forward-engineered designs., Because the original system was reverse-engineered for a variety of reasons, including no longer satisfying user needs it is not reasonable to measure the quality of such designs using metrics formulated for forward-engineered designs which, necessarily, satisfy user needs. Therefore, the evaluation and validation of FORM required the development of a new set of metrics. The metrics used to evaluate the resulting design include statement coverage and functional equivalence. These metrics evaluate the system under the manufacturing view. They are not new metrics, however, their use in this context is novel. The evaluation resulting from the use of these metrics ensure the validity of the extracted design.

Chapter 4

Case Studies

The research detailed in the dissertation develops a methodology that utilizes the concepts of reverse engineering to extract an object-oriented design from imperative code. This chapter will offer a step-by step demonstration of the methodology by performing two case studies. The case studies were chosen for several reasons. Firstly, because the majority of scientific and engineering software systems currently in use are coded in imperative languages, specifically, FORTRAN-77, this language was chosen for the source code. Secondly, because the COMMON block in FORTRAN requires such great attention, examples were chosen to demonstrate both the complexity involved by extensive use of the COMMON block as well as the methodology's performance.

The first case study is a statistics program to compute the standard deviation of two sets of numbers. The second is a program based on graph theory. Both are written in FORTRAN.

4.1 Statistics Case Study

The program used in this case study is a statistics program written in FORTRAN. It is small in size (less than 500 lines of code), and reads two data sets of integers from a file, stores them in arrays and computes the standard deviation of each data set. The program makes extensive use of subprogramming but does not include any COMMON variables.

Using the algorithm for paramater analysis in Figure 3.6, we have:

P = STATISTICS /* the source program */

n = 5 /* the number of subroutine calls */

m = 6 /* the number of distince actual parameters */

The next step is to form the sets CALL1 - CALL5 and the sets V and E. CALL1 - CALL5 correspond to the sets of actual parameters of each subroutine call. The sets V and E represent the distinct actual parameters and funciton resultants. The corresponding sets are as follows:

CALL1 = {m, expa, n, expb,}
CALL2 = {expa, m, stda}
CALL3 = {expa, m, stda, expb, n, stdb}
CALL5 = { }
V = {m, expa, n,expb, stda, stdb} E = {}2

Continuing with the algorithm, construct the weighted adjacency matrix, M. The resulting matrix is shown in Figure 4.1.

	k	1	2	3	4	5	6
j		m	expa	n	expb	stda	stdb
1	m	0	0.6	0.2	0.2	0.3	-0.1
2	expa	0	0	0.2	0.2	0.3	-0.1
3	n	0	0	0	0.6	-0.1	0.3
4	expb	0	0	0	0	-0.1	0.3
5	stda	0	0	0	0	0	0
6	stdb	0	0	0	0	0	0

Figure 4.1
Weighted Adjacency Matrix, M

Once the weighted adjacency matrix M has been formed, the next step in attribute extraction is the generation of the threshold table. A threshold table is generated for M using the transitive closure algorithm on each non-negative value of M. The resulting threshold table is shown in Figure 4.2.

Threshold Values	**Data Sets**
>0	{m, n, expa, expb, stda, stdb}
> 0.2	{m, expa, stda} {n, expb, stdb}
> 0.3	{m, expa} {n, expb} {stda} {stdb}
> 0.6	{m} {n} {stda} {stdb} {expa} {expb}

Figure 4.2
Threshold Table for Matrix M

Given the threshold table for program P, we must select a desirable threshold to complete the attribute analysis. This is done by the human reverse engineer. In selecting a threshold level, the goal is to find the largest data sets with the strongest cohesion, thereby minimizing the number of singleton data sets. Using this heuristic, a threshold value of 0.2 is selected from the threshold table. Selecting a threshold of 0.2 results in two objects. Object 1 has the attribute set {m, expa, stda}. Object 2 has the attribute set {n, expb, stdb}. Using the method extraction algorithms, object 1 has 3 methods and object 2 has 3 methods. The resulting objects templates are given in Figure 4.2 and Figure 4.3.

```
Object1 is
        Attributes
                expa:  array[14] of integer
                m    : integer
                stda : real
        Methods
        Method1
                  begin
                          Read (5,10, end = 15) m, (expa(i),i=1, 14)
                  end
        Method2
          begin
                tot = 0.0
                sum = 0.0
                DO 60 i = 1, m
                  tot = tot + expa(i)
           60   continue
                mean = tot / m
                DO 70 j = 1, m
                  ind(j) = mean - exp(j)
                  sum = sum + ind(j) **2
           70   continue
                stda = sqrt(sum/(m-1))
          end

        Method3
          begin
                write(6, 80) 'Experiment A ', 'Measurements ', ((expa(i),i=1, m)
          80    format /* excluded */
                write (3,90) 'Standard Deviation ', stda
          90    format /* excluded */
                return
          end

        SubDerivation   Input, Std, Print
```

Figure 4.3
Extracted Object Templates: Object 1

```
Object2 is
        Attributes
                expb  : array[14] of integer
                n     : integer
                stdb  : real
        Methods
                Method1
                 begin
                        Read (5,10, end = 15) n, (expb(i),i=i, 14)
                 end

        Method2
         begin
                tot = 0.0
                sum = 0.0
                DO 60 i = 1, n
                 tot = tot + expb(i)
          60    continue
                mean = tot / n
                DO 70 j = 1, n
                 ind(j) = mean - exp(j)
                 sum = sum + ind(j) **2
          70    continue
                stdb = sqrt(sum/(m-1))
         end

        Method3
         begin
                write(6, 80) 'Experiment A ', 'Measurements ', ((expb(i),i=1, n)
         80     format /* excluded */
                write (3,90) 'Standard Deviation ', stdb
         90     format /* excluded */
                return
         end
```

Figure 4.4
Extracted Object Templates: Object 2

Hence, two objects were extracted, each with three attributes and three methods. Next, we abstract the inheritance hierarchy using the two techniques of attribute-based similarity and method-based similarity. Using attribute-based similarity, the following statistics are generated: maximum number of attributes is 3, number of identical types is 3, and percentage of similarity of attributes is 100%. Using method-based similarity, the following statistics are generated: maximum number of methods is 3, number of identical subroutines of derivation is 3, and percentage similarity of methods is 100%. Based on the similarity measures, we can see that object 1 and object 2 are identical objects and are, therefore, combined to form a class. The resulting class template follows in Figure 4.5.

A deeper analysis of the result reveals a statement coverage of 100% and functional equivalence. That is to say, all of the statements of the source code were extracted into the object model. Moreover, functional equivalence was determined by implementing the extracted design in C++ and comparing the outputs. Because the outputs were identical, functional equivalence was extablished. No method invocation templates or state interrogation templates were necessary as the example doesn't make use of the COMMOM block.

Thus, the case study demonstrates the methodology's capability to extract an object-oriented design that maintains the functionality of the source code. The simplicity of the extracted desgin reflects that of the source code. The two extracted objects were identified as identical and mapped into a single class. This case study was well defined and well documented. Therefore, the resulting design was easily evaluated for functional equivalance and statement coverage. The next case study involves a much more complex system with poor documentation.

```
Class1 is {object1, object2}
   ParentClass is { }
   Attributes
      a1: array[14] of integer
      a2: integer
      a3: real
   Methods
      Method1
       begin
            read (a2)
       end

      Method2:
       begin
            tot = 0.0
            sum = 0.0
            DO 60 i = 1, a2
              tot = tot + a1(i)
       60   continue
            mean = tot / a2
            DO 70 j = 1, a2
              ind(j) = mean - exp(j)
              sum = sum + ind(j) ** 2
       70   continue
            a3 = sqrt(sum / (a2 - 1))
       end

      Method3
       begin
            write 'Experiment ', 'Measurements ', (a1(j), j=1,a2)
            write ' Standard Deviation ', a3
       end
```

Figure 4.5
Extracted Class Templates

4.2 Graph Case Study

The program used in this case study is a medium-sized (500 - 3000 lines of code) FORTRAN program. It is a graph reduction program that makes extensive use of subprograms and the COMMON block. Extensive use of subprogramming required the need for resolving aliases using local variable resolution. Local variable resolution using the algorithm given in Figure 3.2 resulted in the 31 distinct actual parameters given in Figure 4.6.

main.pnum	initial.num	initial.i	reduce.face
reduce.arc	reduce.node	loop.face1	loop.num
vertex.node1	vertex.num	series.num	parallel.num
wye.num	wye.arc(i)	wye.k	wye.da
wye.db	wye.dc	wye.wa	wye.wb
wye.wc	delta.num	delta.arc(i)	delta.k
delta.da	delta.db	delta.dc	delta.wa
delta.wb	delta.wc	loop.num1	

Figure 4.6
Distinct Actual Parameters

Using the algorithm in Figure 3.6, a weighted adjacency matrix was formed and the threshold table in Figure 4.7 was formed.

Threshold Value	Data Sets
>= 0	{main.pnum, initial.num, initial.i, reduce.face, reduce.arc, reduce.node, loop.face1, loop.num, loop.num1, vertex.node1, vertex.num, series.num, parallel.num, wye.num, wye.arc(i), wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc, delta.num, delta.arc(i), delta.k, delta.da, delta.db, delta.dc, delta.wa, delta.wb, delta.wc}
>= 0.1	{main.pnum}{vertex.node1,vertex.num} {loop.face1, loop.num1, loop.num} {reduce.arc, reduce.node, reduce.face, series.num, parallel.num}{wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc}{delta.k, delta.da, delta.db, delta.dc, delta.wa, delta.wb, delta.wc} {initial.i, initial.num}{wye.num}{delta.num} {wye.arc(i)} (delta.arc(i)}
>= 0.3	{main.pnum}{vertex.node1,vertex.num} {loop.face1}{loop.num1}{ loop.num} {reduce.arc, reduce.node}{reduce.face} {series.num} {parallel.num}{wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc}{delta.k, delta.da, delta.db, delta.dc, delta.wa, delta.wb, delta.wc} {initial.i, initial.num}{wye.num}{delta.num} {wye.arc(i)} {delta.arc(i)}
>= 0.6	{main.pnum}{vertex.node1}{vertex.num} {loop.face1}{loop.num1}{ loop.num} {reduce.arc}{reduce.node}{reduce.face} {series.num} {parallel.num}{wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc}{delta.k, delta.da, delta.db, delta.dc, delta.wa, delta.wb, delta.wc} {initial.i, initial.num}{wye.num}{delta.num} {wye.arc(i)} {delta.arc(i)}
>= 1.2	{main.pnum}{vertex.node1}{vertex.num} {loop.face1}{loop.num1}{ loop.num} {reduce.arc}{reduce.node}{reduce.face} {series.num} {parallel.num}{wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc}{delta.k, delta.da, delta.db, delta.dc, delta.wa, delta.wb, delta.wc} {initial.i} {initial.num}{wye.num}{delta.num} {wye.arc(i)} {delta.arc(i)}
>= 2.4	{main.pnum}{vertex.node1}{vertex.num} {loop.face1}{loop.num1}{ loop.num} {reduce.arc}{reduce.node}{reduce.face} {series.num} {parallel.num}{wye.k, wye.da, wye.db, wye.dc}{wye.wa}{wye.wb} {wye.wc}{delta.k, delta.da, delta.db, delta.dc} {delta.wa}{delta.wb}{delta.wc} {initial.i} {initial.num}{wye.num}{delta.num} {wye.arc(i)} {delta.arc(i)}

Figure 4.7
Threshold Table For Actual Parameters

Using the heuristics described for selecting a desirable threshold level, a threshold value of 0.3 is selected. Selecting a threshold of 0.3 we get 16 data sets, corresponding to 16 objects. Several of the corresponding object templates are given in the following Figure 4.8 with the complete set given in Appendix A1. The objects shown in Figure 4.8 represent varying complexity: from Object1, with one simple method to Object10 with multiple, more complicated methods.

```
Object1 is
        Attributes
                main.pnum  :  real

        Methods

                Method1
                  begin
                          READ(*,*) PNUM
                  end

        SubDerivation   {Main}

Object8 is
        Attributes
                series.num  :  integer

        Methods

                Method1
                  begin
                        IF (FNODE(EDGE1) .EQ. NODE) THEN
                               EDGE2 = FLEDGE(EDGE1)
                               IF (FNODE(EDGE2) .EQ. NODE) THEN
                                  FLEDGE(EDGE1) = BREDGE(EDGE2)
                                ELSE
```

Figure 4.8
Extracted Object Templates

```
                FLEDGE(EDGE1) = FLEDGE(EDGE2)
              ENDIF
              NUM = FLEDGE(EDGE1)
              IF (FREDGE(NUM) .EQ. EDGE2) THEN
                FREDGE(NUM) = EDGE1
              ENDIF
              NUM = FREDGE(EDGE1)
          ELSE
              EDGE2 = BLEDGE(EDGE1)
              IF (FNODE(EDGE2) .EQ. NODE) THEN
                BREDGE(EDGE1) = BREDGE(EDGE2)
                BLEDGE(EDGE1) = BLEDGE(EDGE2)
              ELSE
                BREDGE(EDGE1) = FLEDGE(EDGE2)
                BLEDGE(EDGE1) = FREDGE(EDGE2)
              ENDIF
          .   NUM = BREDGE(EDGE1)
              IF (BLEDGE(NUM) .EQ. EDGE2) THEN
                BLEDGE(NUM) = EDGE1
             ENDIF
              NUM = BLEDGE(EDGE1)
           NUM = RFACE(EDGE1)
           IF (NUM .EQ. LFACE(EDGE1)) THEN
           ELSE
              NUM = LFACE(EDGE1)
           ENDIF

    SubDerivation {Series}

Object10 is
    Attributes
        wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc  :  real

    Methods

        Method1
          begin
              ARC(1) = EDGE
              IF  (BNODE(EDGE) .EQ. NODE) THEN
                   ARC(2) = BREDGE(EDGE)
                   ARC(3) = BLEDGE(EDGE)
```

(figure continued)

```
            ELSE
                ARC(2) = FLEDGE(EDGE)
                ARC(3) = FREDGE(EDGE)
            ENDIF
            DO  110 J = 1,10
                DA = 1- POS(ARC(1),J)
                DB = 1- POS(ARC(2),J)
                DC = 1- POS(ARC(3),J)
                RHO = DA*DB*DC - (DA+DB+DC) + 1
                IF (RHO .GT. 0 ) THEN
                   K = 1.0
                ELSE
                   K= 0.0
                ENDIF
                POS(ARC(1),J) = 1-WC
                POS(ARC(2),J) = 1-WA
                POS(ARC(3),J) = 1-WB
                DA = 1- NEG(ARC(1),J)
                DB = 1- NEG(ARC(2),J)
                DC = 1- NEG(ARC(3),J)
                RHO = DA*DB*DC - (DA+DB+DC) + 1
                IF (RHO .GT. 0 ) THEN
                   K = 0.0
                ELSE
                   K= 1.0
                       ENDIF
                 DA = 1- POINT(ARC(1),J)
                DB = 1- POINT(ARC(2),J)
                DC = 1- POINT(ARC(3),J)
                K = 0.5
                POINT(ARC(1),J) = 1-WC
                POINT(ARC(2),J) □= 1-WA
                POINT(ARC(3),J) = 1-WB
            110   CONTINUE
        end


      Method2
         begin
            IF ( K .EQ. 0 ) THEN
                   A1 = DA + DB*DC - DA*DB*DC
```

(figure continued)

```
                    B1 = DB + DA*DC - DA*DB*DC
                    C1 = DC + DB*DA - DA*DB*DC
                    WA = DSQRT(B1*C1/A1)
                    WB = DSQRT(A1*C1/B1)
                    WC = DSQRT(B1*A1/C1)
                ENDIF
                IF (DABS(WA-WB) .LE. 0.00001) THEN
                ELSE
                     WC = (1-DC)*(DB-DA)/(WA-WB)
                ENDIF
            end

        SubDerivation {Wye, Transform}


Object16 is
        Attributes
                delta.arc(i) : integer

        Methods
                Method1
                   begin
                       ARC(1) = EDGE
                          IF (RFACE(EDGE) .EQ. FACE) THEN
                             ARC(2) = FREDGE(EDGE)
                             ARC(3) = BREDGE(EDGE)
                          ELSE
                             ARC(2) = BLEDGE(EDGE)
                             ARC(3) = FLEDGE(EDGE)
                          ENDIF

                  end

        SubDerivation {Delta}
```

The next step involves performing analysis on the COMMON variables. Using the algorithm given in Figure 3.8, a weighted adjacency matrix is generated, and a thresold table is formed. The partial threshold table is shown in Figure 4.9. Upon observation of the table, it becomes apparent that as the threshold value

Threshold	Data Sets
> 0	{fnode, bnode, lface, rface, dnode, dface, label, fledge, fredge, bredge, bledge, nn, na, nf, sore, sink1, flag, facnum, nodnum, arcnum, pos, neg, point, stprob, upprob, loprob, errpos, errneg, nterm, term1, term2, term3, term4, above, below, incdnt, topdel, topwye, topser, toppar, topvert, toploop} {sink2} {oface}
> 0.1	{fnode, bnode, lface, rface, dnode, dface, label, fledge, fredge, bredge, bledge, nn, na, nf, sore, sink1, flag, facnum, nodnum, arcnum, pos, neg, point, stprob, upprob, loprob, errpos, errneg, nterm, term1, term2, term3, term4, above, below, incdnt, topdel, topwye, topser, toppar, topvert, toploop} {sink2} {oface}
> 0.2	{fnode, bnode, lface, rface, dnode, dface, label, fledge, fredge, bredge, bledge, nn, na, nf, sore, sink1, flag, arcnum, pos, neg, point, errpos, errneg, term1, term2, term3, term4, above, below, incdnt, topdel, topwye, topser, toppar, topvert, toploop} {stprob, upprob, loprob} {sink2} {oface} {facnum} {nodnum} {nterm}
> 0.3	{fnode, bnode, lface, rface, dnode, dface, label, fledge, fredge, bredge, bledge, nn, na, nf, flag, arcnum, pos, neg, point, errpos, errneg, incdnt} {topdel, topwye, topser, toppar, topvert, toploop} {stprob, upprob, loprob} {term1, term2, term3, term4} {above, below} {sink2} {oface} {facnum} {nodnum} {nterm} {sore} {sink1}
> 0.4	{fnode, bnode, lface, rface, dnode, dface, label, fledge, fredge, bredge, bledge, nn, na, nf, flag, pos, neg, point, incdnt} {errpos, errneg} {topdel, topwye, topser, toppar, topvert, toploop} {stprob, upprob, loprob} {term1} {term2} {term3} {term4} {above} {below} {sink2} {oface} {facnum} {nodnum} {nterm} {sore} {sink1} {arcnum}
> 0.5	{fnode, bnode, lface, rface, dnode, dface, label, fledge, fredge, bredge, bledge, nn, na, nf, flag, pos, neg, point, incdnt} {errpos, errneg} {topdel, topser, toppar, topvert, toploop} {stprob, upprob, loprob} {term1} {term2} {term3} {term4} {above} {below} {sink2} {oface} {facnum} {nodnum} {nterm} {sore} {sink1} {arcnum} {topwye}
> 1.7	{bnode} {rface, dnode} {dface} {incdnt} {label} {fledge} {fredge} {bredge} {bledge} {pos} {neg} {point}{errpos} {errneg} {topdel} {topser} {toppar} {topvert} {toploop} {stprob} {upprob} {loprob} {term1} {term2} {term3} {term4} {above} {below} {sink2} {oface} {facnum} {nodnum} {nterm} {sore} {sink1} {arcnum} {topwye} {nf} {nn} {na} {lface} {fnode} {flag}
> 1.8	{bnode} {rface} {dnode} {dface} {incdnt} {label} {fledge} {fredge} {bredge} {bledge} {pos} {neg} {point}{errpos} {errneg} {topdel} {topser} {toppar} {topvert} {toploop} {stprob} {upprob} {loprob} {term1} {term2} {term3} {term4} {above} {below} {sink2} {oface} {facnum} {nodnum} {nterm} {sore} {sink1} {arcnum} {topwye} {nf} {nn} {na} {lface} {fnode} {flag}

Figure 4.9
Threshold Table For COMMON Variables

increases, so does the number of singleton sets. Thus, after the 0.5 threshold level, the number of singleton sets becomes undesirable, and only continues to increase. Therefore, using the heuristics outlined for selecting a threshold value, 0.3 is selected. Thus, global variable analysis results in the extraction of 12 objects at the 0.3 threshold level. The complete set of corresponding object templates are given in Appendix A2, with a representative subset following in Figure 4.10. The objects in Figure 4.10 were chosen to represent varying degrees of complexity. Object17, the more complex object, has nine methods while Object21 has a single method and Object 23 has no methods, and serves to detect dead code.

```
Object 17 is
        Attributes      fnode, bnode, lface, rface, dnode, dface, label  :  integer
                        fledge, fredge, bredge, bledge, nn, na, nf, flag, incdnt:  integer
                        pos, neg, point, errpos, errneg  :  real

        Methods

                Method1
                  begin
                        IF (BNODE(EDGE) .EQ. NODE) THEN
                              NODE1 = FNODE(EDGE)
                              NUM = FLEDGE(EDGE)
                              IF(FREDGE(NUM).EQ. EDGE) THEN
                                 FREDGE(NUM) = FREDGE(EDGE)
                              ELSE
                                 BLEDGE(NUM) = FREDGE(EDGE)
                              ENDIF
                              NUM = FREDGE(EDGE)
                              IF(FLEDGE(NUM).EQ. EDGE) THEN
                                 FLEDGE(NUM) = FLEDGE(EDGE)
```

Figure 4.10
Extracted Object Templates

```
                 ELSE
                   BREDGE(NUM) = FLEDGE(EDGE)
                 ENDIF
               ELSE
                 NODE1 = BNODE(EDGE)
                 NUM =BLEDGE(EDGE)
                 IF(BREDGE(NUM).EQ. EDGE) THEN
                   BREDGE(NUM) = BREDGE(EDGE)
                 ELSE
                   FLEDGE(NUM) = BREDGE(EDGE)
                 ENDIF
                       NUM = BREDGE(EDGE)
                 IF(BLEDGE(NUM).EQ. EDGE) THEN
                   BLEDGE(NUM) = BLEDGE(EDGE)
                 ELSE
                   FREDGE(NUM) = BLEDGE(EDGE)
                 ENDIF
               ENDIF
           DNODE(NODE1)= DNODE(NODE1)-1
           DFACE(NUM1) = DFACE(NUM1)-2
           DNODE(NODE) = 0
           LABEL(EDGE) = 0
           NN=NN-1
           NA=NA-1
       end


     Method2
       begin
           IF (RFACE(EDGE) .EQ. FACE) THEN
                 IF(BLEDGE(NUM).EQ. EDGE) THEN
                   BLEDGE(NUM) = BLEDGE(EDGE)
                 ELSE
                   FREDGE(NUM) = BLEDGE(EDGE)
                 ENDIF
                 NUM = BLEDGE(EDGE)
                 IF(FLEDGE(NUM).EQ. EDGE) THEN
                   FLEDGE(NUM) = FLEDGE(EDGE)
                 ELSE
                   BREDGE(NUM) = FLEDGE(EDGE)
                 ENDIF
```

(figure continued)

```
                ELSE
                   NUM = BREDGE(EDGE)
                   IF(FREDGE(NUM).EQ. EDGE) THEN
                      FREDGE(NUM) = FREDGE(EDGE)
                   ELSE
                      BLEDGE(NUM) = FREDGE(EDGE)
                   ENDIF
                   NUM = FREDGE(EDGE)
                   IF(BREDGE(NUM).EQ. EDGE) THEN
                      BREDGE(NUM) = BREDGE(EDGE)
                   ELSE
                      FLEDGE(NUM) = BREDGE(EDGE)
                   ENDIF
                ENDIF
                DFACE(FACE1)= DFACE(FACE1)-1
                DNODE(NUM1) = DNODE(NUM1)-2
                DFACE(FACE) = 0
               LABEL(EDGE) = 0
               NF=NF-1
               NA=NA-1
   end


Method3
   begin
         ARC(1) = EDGE
         IF  (BNODE(EDGE) .EQ. NODE) THEN
                 ARC(2) = BREDGE(EDGE)
                 ARC(3) = BLEDGE(EDGE)
        ELSE
                 ARC(2) = FLEDGE(EDGE)
                 ARC(3) = FREDGE(EDGE)
            ENDIF
        DO  10 I = 1,3
                ROT(I) = MOD(I,3) + 1
                IF (BNODE(ARC(I)) .EQ. NODE) THEN
                    TEMP(1,I) = FNODE(ARC(I))
                    TEMP(2,I)  = RFACE(ARC(I))
                    TEMP(3,I) = FREDGE(ARC(I))
                    TEMP(4,I) = FLEDGE(ARC(I))
                ELSE
```

(figure continued)

```
                TEMP(1,I) = BNODE(ARC(I))
                TEMP(2,I) = LFACE(ARC(I))
                TEMP(3,I) = BLEDGE(ARC(I))
                TEMP(4,I) = BREDGE(ARC(I))
          ENDIF
                TEMP(5,I) = LABEL(ARC(I))
10     CONTINUE
       DO  20 I = 1,3
          RFACE(ARC(I)) = NEWFAC
          BNODE(ARC(I)) = TEMP(1,I)
          LFACE(ARC(I)) = TEMP(2,I)
          FNODE(ARC(I)) = TEMP(1,ROT(I))
          BLEDGE(ARC(I)) = TEMP(3,I)
          FLEDGE(ARC(I)) = TEMP(4,ROT(I))
          FREDGE(ARC(I)) = ARC(ROT(I))
          BREDGE(ARC(I)) = ARC(ROT(ROT(I)))
        LABEL(ARC(I)) = MIN(TEMP(5,I),TEMP(5,ROT(I)))
20      CONTINUE
            DFACE(NEWFAC) = 3
       DO 25  I = 1,3
          NUM = FLEDGE(ARC(I))
          IF(LFACE(NUM) .EQ. TEMP(2,I)) THEN
               BLEDGE(NUM) = ARC(I)
          ELSE
               FREDGE(NUM) = ARC(I)
          ENDIF
25      CONTINUE
          DO  30 I = 1,3
               NUM = LFACE(ARC(I))
               DFACE(NUM) = DFACE(NUM) - 1
30      CONTINUE
          DO  40 I=1,3
               NUM = FNODE(ARC(I))
               DNODE(NUM) = DNODE(NUM)+ 1
40      CONTINUE
          DO  110 J = 1,10
                DA = 1- POS(ARC(1),J)
                DB = 1- POS(ARC(2),J)
                DC = 1- POS(ARC(3),J)
                POS(ARC(1),J) = 1-WC
                POS(ARC(2),J) = 1-WA
```

(figure continued)

```
                    POS(ARC(3),J) = 1-WB
                    ERRPOS(J) =
ERRPOS(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*DA*DB*DC)
                    DA = 1- NEG(ARC(1),J)
                    DB = 1- NEG(ARC(2),J)
                    DC = 1- NEG(ARC(3),J)
                    NEG(ARC(1),J) = 1-WC
                    NEG(ARC(2),J) = 1-WA
                    NEG(ARC(3),J) = 1-WB

ERRNEG(J)=ERRNEG(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*D
A*DB*DC)
                    DA = 1- POINT(ARC(1),J)
                    DB = 1- POINT(ARC(2),J)
                    DC = 1- POINT(ARC(3),J)
                    POINT(ARC(1),J) = 1-WC
                    POINT(ARC(2),J) = 1-WA
                    POINT(ARC(3),J) = 1-WB
            110   CONTINUE
                   DNODE(NODE) = 0
                    NN=NN-1
                    NF=NF+1
        end

          Method4
            begin
                ARC(1) = EDGE
                  IF  (RFACE(EDGE) .EQ. FACE) THEN
                     ARC(2) = FREDGE(EDGE)
                     ARC(3) = BREDGE(EDGE)
                  ELSE
                     ARC(2) = BLEDGE(EDGE)
                     ARC(3) = FLEDGE(EDGE)
                  ENDIF
                  DO  10 I = 1,3
                     ROT(I) = MOD(I,3) + 1
                     IF (RFACE(ARC(I)) .EQ. FACE) THEN
                        TEMP(1,I) = FNODE(ARC(I))
                        TEMP(2,I) = LFACE(ARC(I))
                        TEMP(3,I)  = FLEDGE(ARC(I))
                        TEMP(4,I) = BLEDGE(ARC(I))
```

(figure continued)

```
          ELSE
              TEMP(1,I) = BNODE(ARC(I))
              TEMP(2,I) = RFACE(ARC(I))
              TEMP(3,I) = BREDGE(ARC(I))
              TEMP(4,I) = FREDGE(ARC(I))
          ENDIF
              TEMP(5,I) = LABEL(ARC(I))
       DO  20 I = 1,3
          BNODE(ARC(I)) = NEWNOD
          FNODE(ARC(I)) = TEMP(1,I)
          LFACE(ARC(I)) = TEMP(2,I)
          RFACE(ARC(I)) = TEMP(2,ROT(I))
          FLEDGE(ARC(I)) = TEMP(3,I)
          FREDGE(ARC(I)) = TEMP(4,ROT(I))
          BREDGE(ARC(I)) = ARC(ROT(I))
          BLEDGE(ARC(I)) = ARC(ROT(ROT(I)))
LABEL(ARC(I)) = MIN(TEMP(5,I),TEMP(5,ROT(I)))
 20    CONTINUE
       DNODE(NEWNOD) = 3
       DO 25  I = 1,3
          NUM = FREDGE(ARC(I))
          IF(FNODE(NUM) .EQ. TEMP(1,I)) THEN
             FLEDGE(NUM) = ARC(I)
          ELSE
             BREDGE(NUM) = ARC(I)
          ENDIF
 25    CONTINUE
       DO  30 I = 1,3
         NUM = RFACE(ARC(I))
         DFACE(NUM) = DFACE(NUM)+ 1
 30    CONTINUE
       DO  40 I = 1,3
         NUM = FNODE(ARC(I))
         DNODE(NUM) = DNODE(NUM) - 1
 40    CONTINUE
       DO  110 J =1,10
       DA =  POS(ARC(1),J)
       DB =  POS(ARC(2),J)
       DC =  POS(ARC(3),J)
       POS(ARC(1),J) = WC
       POS(ARC(2),J) = WA
```

(figure continued)

```
                    POS(ARC(3),J) = WB

ERRPOS(J)=ERRPOS(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*DA
*DB*DC)
                    DA = NEG(ARC(1),J)
                    DB = NEG(ARC(2),J)
                    DC = NEG(ARC(3),J)
                    NEG(ARC(1),J) = WC
                    NEG(ARC(2),J) = WA
                    NEG(ARC(3),J) = WB
ERRNEG(J)=ERRNEG(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*D
A*DB*DC)
                   DA = POINT(ARC(1),J)
                    DB = POINT(ARC(2),J)
                    DC = POINT(ARC(3),J)
                  . POINT(ARC(1),J) = WC
                    POINT(ARC(2),J) = WA
                    POINT(ARC(3),J) = WB
                110   CONTINUE
                    DFACE(FACE) = 0
                    NF=NF-1
                    NN=NN+1
           end

           Method5
             begin
                  IF (RFACE(EDGE1) .EQ. FACE) THEN
                       EDGE2 = FREDGE(EDGE1)
                       IF (RFACE(EDGE2) .EQ. FACE) THEN
                          FREDGE(EDGE1) = BLEDGE(EDGE2)
                          BREDGE(EDGE1) = FLEDGE(EDGE2)
                          RFACE(EDGE1) = LFACE(EDGE2)
                       ELSE
                          FREDGE(EDGE1) = FREDGE(EDGE2)
                          BREDGE(EDGE1) = BREDGE(EDGE2)
                          RFACE(EDGE1) = RFACE(EDGE2)
                       ENDIF
                       NUM = FREDGE(EDGE1)
                       IF(BREDGE(NUM).EQ. EDGE2) THEN
                          BREDGE(NUM) = EDGE1
```

(figure continued)

```
               ELSE
                  FLEDGE(NUM) = EDGE1
               ENDIF
               NUM = BREDGE(EDGE1)
               IF(FREDGE(NUM).EQ. EDGE2) THEN
                  FREDGE(NUM) = EDGE1
               ELSE
                  BLEDGE(NUM) = EDGE1
               ENDIF
            ELSE
               EDGE2 = FLEDGE(EDGE1)
               IF (RFACE(EDGE2) .EQ. FACE) THEN
                  FLEDGE(EDGE1) = FLEDGE(EDGE2)
                  BLEDGE(EDGE1) = BLEDGE(EDGE2)
                  LFACE(EDGE1) = LFACE(EDGE2)
               ELSE
                  FLEDGE(EDGE1) = BREDGE(EDGE2)
                  BLEDGE(EDGE1) = FREDGE(EDGE2)
                  LFACE(EDGE1) = RFACE(EDGE2)
               ENDIF
               NUM = FLEDGE(EDGE1)
               IF(BLEDGE(NUM).EQ. EDGE2) THEN
                  BLEDGE(NUM) = EDGE1
               ELSE
                  FREDGE(NUM) = EDGE1
               ENDIF
               NUM = BLEDGE(EDGE1)
               IF(FLEDGE(NUM).EQ. EDGE2) THEN
                  FLEDGE(NUM) = EDGE1
               ELSE
                  BREDGE(NUM) = EDGE1
               ENDIF
            ENDIF
    LABEL(EDGE1) = MIN(LABEL(EDGE1),LABEL(EDGE2))
            LABEL(EDGE2) = 0
            NUM = FNODE(EDGE1)
            DNODE(NUM) = DNODE(NUM)-1
       1000 IF (RFACE(EDGE2) .EQ.FACE) THEN
              IF(INCDNT(LFACE(EDGE2)) .GT. 0)
                        INCDNT(LFACE(EDGE2)) = EDGE1
           ELSE
```

(figure continued)

```
                    IF(INCDNT(RFACE(EDGE2)) .GT. 0)
                       INCDNT(RFACE(EDGE2)) = EDGE1
                    ENDIF
                    DFACE(FACE)=0
                    DO 110 J = 1,10
POS(EDGE1,J)=
           POS(EDGE1,J)+POS(EDGE2,J)-POS(EDGE1,J)*POS(EDGE2,J)
              POINT(EDGE1,J)=POINT(EDGE1,J)+POINT(EDGE2,J)-
                                  POINT(EDGE1,J)*POINT(EDGE2,J)
  NEG(EDGE1,J) =
      NEG(EDGE1,J)+NEG(EDGE2,J)-NEG(EDGE1,J)*NEG(EDGE2,J)
                 110  CONTINUE
                    NF=NF-1
                    NA=NA-1
           end

           Method6
             begin
                 IF (FNODE(EDGE1) .EQ. NODE) THEN
                      EDGE2 = FLEDGE(EDGE1)
                      IF (FNODE(EDGE2) .EQ. NODE) THEN
                         FLEDGE(EDGE1) = BREDGE(EDGE2)
                         FREDGE(EDGE1) = BLEDGE(EDGE2)
                         FNODE(EDGE1) = BNODE(EDGE2)
                      ELSE
                         FLEDGE(EDGE1) = FLEDGE(EDGE2)
                         FREDGE(EDGE1) = FREDGE(EDGE2)
                         FNODE(EDGE1) = FNODE(EDGE2)
                      ENDIF
                      NUM = FLEDGE(EDGE1)
                      IF (FREDGE(NUM) .EQ. EDGE2) THEN
                         FREDGE(NUM) = EDGE1
                      ELSE
                         BLEDGE(NUM) = EDGE1
                      ENDIF
                      NUM = FREDGE(EDGE1)
                      IF (FLEDGE(NUM) .EQ. EDGE2) THEN
                         FLEDGE(NUM) = EDGE1
                      ELSE
                         BREDGE(NUM) = EDGE1
                      ENDIF
```

(figure continued)

```
          ELSE
             EDGE2 = BLEDGE(EDGE1)
             IF (FNODE(EDGE2) .EQ. NODE) THEN
                BREDGE(EDGE1) = BREDGE(EDGE2)
                BLEDGE(EDGE1) = BLEDGE(EDGE2)
                BNODE(EDGE1) = BNODE(EDGE2)
             ELSE
                BREDGE(EDGE1) = FLEDGE(EDGE2)
                BLEDGE(EDGE1) = FREDGE(EDGE2)
                BNODE(EDGE1) = FNODE(EDGE2)
             ENDIF
             NUM = BREDGE(EDGE1)
             IF (BLEDGE(NUM) .EQ. EDGE2) THEN
                BLEDGE(NUM) = EDGE1
             ELSE
                FREDGE(NUM) = EDGE1
             ENDIF
             NUM = BLEDGE(EDGE1)
             IF (BREDGE(NUM) .EQ. EDGE2) THEN
                BREDGE(NUM) = EDGE1
             ELSE
                FLEDGE(NUM) = EDGE1
             ENDIF
          ENDIF
 LABEL(EDGE1) = MIN(LABEL(EDGE1),LABEL(EDGE2))
          LABEL(EDGE2) = 0
          NUM = RFACE(EDGE1)
          DFACE(NUM) = DFACE(NUM)-1
    1000 IF (BNODE(EDGE2) .EQ.NODE) THEN
            IF(INCDNT(FNODE(EDGE2)) .GT. 0)
              INCDNT(FNODE(EDGE2)) = EDGE1
         ELSE
            IF(INCDNT(BNODE(EDGE2)) .GT. 0)
              INCDNT(BNODE(EDGE2)) = EDGE1
         ENDIF
         DNODE(NODE) = 0
         DO 110 J= 1,10
              POS(EDGE1,J) = POS(EDGE1,J)*POS(EDGE2,J)
             NEG(EDGE1,J) = NEG(EDGE1,J)*NEG(EDGE2,J)
POINT(EDGE1,J) = POINT(EDGE1,J)*POINT(EDGE2,J)
    110   CONTINUE
```

(figure continued)

```
           NN=NN-1
           NA=NA-1
   end

 Method7
   begin
        READ(4,*)  NN, NA, NF
         DO 15 J=1, NA
        READ(4,*) I,BNODE(I),FNODE(I),LFACE(I),RFACE(I)
READ(4,*) BREDGE(I),BLEDGE(I),FLEDGE(I),FREDGE(I)
             ARCNUM(J) = I
        15   CONTINUE
           DO 16 J=1,NN
              READ(4,*) I, DNODE(I)
              NODNUM(J) = I
        16   CONTINUE
           DO 17 K= 1,NF
              READ(4,*) I, DFACE(I)
              FACNUM(K) = I
        17   CONTINUE
           READ(*,*) PNUM
           DO 110 J = 1,10
            DO 47 I = 1,NA
             POS(I,J) = PNUM + J*0.02
             NEG(I,J) = PNUM + J*0.02
             POINT(I,J) =PNUM + J*0.02
        47     CONTINUE
        110    CONTINUE
           LABEL(START) = 1
          EBOT = START
           OCURR=START
        1111 IF (BNODE(OCURR) .EQ. SORE) THEN
             SFACE(OCURR) = RFACE(OCURR)
             OCURR = BREDGE(OCURR)
           ELSE
             SFACE(OCURR) = LFACE(OCURR)
             OCURR = FLEDGE(OCURR)
           ENDIF
           IF (OCURR .NE. START) THEN
            LABEL(OCURR) = 1
            NCOUNT = NCOUNT + 1
```

(figure continued)

```
              NEXT(EBOT) =OCURR
              EBOT = OCURR
              NEXT(EBOT) = 0
              GOTO 1111
            ENDIF
        137  ESCAN = ETOP
            CFACE = SFACE(ESCAN)
            WRITE(*,*) 'SCANNING EDGE #', ESCAN
            ECURR = ESCAN
        122  IF (CFACE .EQ. RFACE(ECURR)) THEN
               NUM = FNODE(ECURR)
               ECURR = FREDGE(ECURR)
            ELSE
               NUM = BNODE(ECURR)
               ECURR = BLEDGE(ECURR)
            ENDIF
          . IF (LABEL(ECURR) .EQ. 0) THEN
                LABEL(ECURR) = LABEL(ESCAN) + 1
   .            NCOUNT = NCOUNT + 1
WRITE(*,*) 'labelling edge #', ECURR,'by label', LABEL(ECURR)
                IF (NCOUNT .EQ. NA) GOTO 1333
                SNODE(ECURR) = NUM
                IF (OTOP .EQ. 0 ) THEN
                   OTOP = ECURR
                ELSE
                   NEXT(OBOT) = ECURR
                ENDIF
                OBOT = ECURR
                NEXT(OBOT) = 0
            ENDIF
            IF (LABEL(ECURR) .EQ. LABEL(ESCAN)) THEN
                ETOP = NEXT(ETOP)
                IF (ETOP .EQ. 0) THEN
                    IF(OTOP .EQ. 0) THEN
                      GOTO 1333
                    ELSE
                      GOTO 237
                    ENDIF
                  ENDIF
        GOTO 137
            ENDIF
```

(figure continued)

```
          GOTO 122
      237  OSCAN = OTOP
          CNODE = SNODE(OSCAN)
          OCURR = OSCAN
      222  IF (CNODE .EQ. BNODE(OCURR)) THEN
             NUM = LFACE(OCURR)
             OCURR = BLEDGE(OCURR)
          ELSE
             NUM-= RFACE(OCURR)
             OCURR = FREDGE(OCURR)
          ENDIF
          IF (LABEL(OCURR) .EQ. 0) THEN
             NCOUNT = NCOUNT + 1
             LABEL(OCURR) = LABEL(OSCAN) + 1
WRITE(*,*) 'labelling edge #', OCURR,'by label', LABEL(OCURR)
             IF (NCOUNT .EQ. NA) GOTO 1333
             SFACE(OCURR) = NUM
             IF (ETOP .EQ. 0 ) THEN
                ETOP = OCURR
             ELSE
                NEXT(EBOT) = OCURR
             ENDIF
             EBOT = OCURR
             NEXT(EBOT) = 0
          ENDIF
          IF (LABEL(OCURR) .EQ. LABEL(OSCAN)) THEN
              OTOP = NEXT(OTOP)
             NEXT(OSCAN) = 0
              IF (OTOP .EQ. 0) THEN
                 IF(ETOP .EQ. 0) THEN
                  GOTO 1333
                 ELSE
                    GOTO 137
                 ENDIF
              ENDIF
               GOTO 237
           ENDIF
           GOTO 222
     end
```

(figure continued)

```
Method8
   begin
         INCDNT(NUM) = 0
              ABOVE(BELOW(NUM)) = ABOVE(NUM)
              BELOW(ABOVE(NUM)) = BELOW(NUM)
              ABOVE(NUM) = 0
              BELOW(NUM) = 0
   end


Method9
  begin
       IF (FLAG .EQ. 1) THEN
              FACE = NUM
              IF(DFACE(FACE) .EQ. 1) THEN
                IF(TOPLOOP .GT. 0) THEN
                   ABOVE(TOPLOOP)=FACE
                   BELOW(FACE) =TOPLOOP
                ENDIF
                TOPLOOP = FACE
                INCDNT(FACE) = ARC
                GOTO 1000
              ENDIF
              IF(DFACE(FACE) .EQ. 2) THEN
                IF(TOPPAR .GT. 0) THEN
                   ABOVE(TOPPAR)=FACE
                   BELOW(FACE) =TOPPAR
                ENDIF
                TOPPAR = FACE
                INCDNT(FACE) = ARC
                GOTO 1000
              ENDIF
              EDGE1= ARC
              IF (RFACE(EDGE1) .EQ. FACE) THEN
                 EDGE2 = FREDGE(EDGE1)
                 EDGE3 = BREDGE(EDGE1)
              ELSE
                 EDGE2 = BLEDGE(EDGE1)
                 EDGE3 = FLEDGE(EDGE1)
              ENDIF
NCHECK = MIN(LABEL(EDGE1),LABEL(EDGE2),LABEL(EDGE3))
```

(figure continued)

```
NSUM =LABEL(EDGE1)+LABEL(EDGE2)+LABEL(EDGE3)
                IF (NSUM .EQ. (3*NCHECK+2)) THEN
                   INCDNT(FACE) = EDGE1
                   IF (TOPDEL .GT. 0) THEN
                     ABOVE(TOPDEL)=FACE
                     BELOW(FACE) =TOPDEL
                   ENDIF
                   TOPDEL = FACE
                ENDIF
                GOTO 1000
             ENDIF
                NODE = NUM
                IF(NODE .EQ. SORE)  GOTO 1000
                IF(NODE .EQ. TERM1)  GOTO 1000
                IF(NODE .EQ. TERM2)  GOTO 1000
                IF(NODE .EQ. TERM3)  GOTO 1000
                IF(NODE .EQ. TERM4 )  GOTO 1000
                IF (DNODE(NODE) .EQ. 0) GOTO 1000
                IF(DNODE(NODE) .EQ. 1) THEN
                  IF(TOPVERT .GT. 0) THEN
                     ABOVE(TOPVERT)=NODE
                     BELOW(NODE) =TOPVERT
                  ENDIF
                  TOPVERT = NODE
                  INCDNT(NODE) = ARC
                  GOTO 1000
                ENDIF
                IF(DNODE(NODE) .EQ. 2) THEN
                  IF(TOPSER .GT. 0) THEN
                     ABOVE(TOPSER)=NODE
                     BELOW(NODE) =TOPSER
                  ENDIF
                  TOPSER = NODE
                  INCDNT(NODE) = ARC
                  GOTO 1000
                ENDIF
                EDGE1= ARC
                IF (BNODE(EDGE1) .EQ. NODE) THEN
                   EDGE2 = BREDGE(EDGE1)
                   EDGE3 = BLEDGE(EDGE1)
                ELSE
```

(figure continued)

```
                         EDGE2 = FLEDGE(EDGE1)
                         EDGE3 = FREDGE(EDGE1)
                      ENDIF
        NCHECK = MIN(LABEL(EDGE1),LABEL(EDGE2),LABEL(EDGE3))
        NSUM =LABEL(EDGE1)+LABEL(EDGE2)+LABEL(EDGE3)
                      IF (NSUM .EQ. (3*NCHECK+2)) THEN
                         INCDNT(NODE) = EDGE1
                          IF (TOPWYE .GT. 0) THEN
                            ABOVE(TOPWYE)=NODE
                            BELOW(NODE) =TOPWYE
                          ENDIF
                          TOPWYE = NODE
                      ENDIF
           end

SubDerivation {vertex, loop, wye, delta, parallel, series,remove, positive, main}

Object21 is
     Attributes
           above, below  : integer

     Methods

           Method1
             begin
                 IF(DFACE(FACE) .EQ. 1) THEN
                    IF(TOPLOOP .GT. 0) THEN
                       ABOVE(TOPLOOP)=FACE
                       BELOW(FACE) =TOPLOOP
                    ENDIF
                 ENDIF
                 IF(DFACE(FACE) .EQ. 2) THEN
                    IF(TOPPAR .GT. 0) THEN
                       ABOVE(TOPPAR)=FACE
                       BELOW(FACE) =TOPPAR
                    ENDIF
                 ENDIF
                 IF (NSUM .EQ. (3*NCHECK+2)) THEN
                     IF (TOPDEL .GT. 0) THEN
                       ABOVE(TOPDEL)=FACE
                       BELOW(FACE) =TOPDEL
```

(figure continued)

```
                   ENDIF
                ENDIF
                 NODE = NUM
                IF(DNODE(NODE) .EQ. 1) THEN
                  IF(TOPVERT .GT. 0) THEN
                     ABOVE(TOPVERT)=NODE
                     BELOW(NODE) =TOPVERT
                  ENDIF
                ENDIF
                IF(DNODE(NODE) .EQ. 2) THEN
                  IF(TOPSER .GT. 0) THEN
                     ABOVE(TOPSER)=NODE
                     BELOW(NODE) =TOPSER
                  ENDIF
                ENDIF
                IF (NSUM .EQ. (3*NCHECK+2)) THEN
                  . IF (TOPWYE .GT. 0) THEN
                     ABOVE(TOPWYE)=NODE
                     BELOW(NODE) =TOPWYE
                   ENDIF
                ENDIF
           end

     SubDerivation {positive}

Object23 is
     Attributes
          oface : integer
     Methods

     SubDerivation {}
```

Because object 22 and object 23 have no methods, a more in depth analysis is performed. These objects were determined to contain attributes that correspond to variables in the source code which are never used. That is, the attributes of object 22 and object 23 are never defined or referenced in the source code. Thus, the declaration of these variables in the source code is unnecessary. It is in this manner that "dead code" can be detected during object extraction.

At this point, the object extraction process is completed. A total of 28 objects were extracted from the source code. Upon reviewing the extracted objects, two cases of dead code were detected. These two objects are then dropped from further consideration. Thus, class abstraction and instantiation of the inheritance hierarchy are performed on the remaining 26 objects. At this point, we begin class abstraction and inheritance hierarchy instantiation using the similarity technique. Similarity is measured in two distinct areas: attributes and methods. Then, the results are compiled and a class hierarchy is reported.

Attribute-based similarity is computed for the actual parameters using the algorithm given in Figure 3.10. The results are given in Figure 4.11.

	Threshold Value	Clusters
	>= 0.0	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)
	>= 0.2	(1, 10, 11) (2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16)
	> = 0.4	(1, 10, 11) (2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16)
	>= 0.6	(1) (10, 11) (2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16)
	>= 0.7	(1) (10, 11) (2, 3, 4, 5, 6, 7, 8, 9, 12, 13, 14, 15, 16)
**	>= 0.9	(1) (2, 6, 12) (10, 11) (3, 4, 5, 7, 8, 9, 13, 14, 15, 16)

Figure 4.11
Attribute-Based Similarity of Actual Parameters

Next, method based similarity is computed using the algorithm in Figure 3.11. The results are shown in Figure 4.12.

	Threshold value	Clusters
	>= 0.0	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16)
	>= 0.2	(1) (2) (3,4,5) (6, 7) (8) (9) (12)
		(10, 11, 13, 14, 15, 16)
**	>= 0.4	(1) (2) (3,4,5) (6, 7) (8) (9) (10, 11) (12)
		(13, 15) (14, 16)
	>= 0.7	(1) (2) (3,4,5) (6, 7) (8) (9) (10) (11) (12)
		(13, 15) (14, 16)

Figure 4.12
Method-Based Similarity of Actual Parameters

The results of the attribute-based similarity analysis and the method-based similarity analysis are used in the algorithm given in Figure 3.12 to generate clusters. Using either attribute clustering or method clustering techniques, two distinct sets of clusters are obtained from which the human reverse engineer selects to give the inheritance model. The clusters formed using attribute clustering are {(1) (2, 6, 12) (3, 4, 5) (7, 8, 9) (10, 11) (13, 15) (14, 16)}. The clusters formed using method clustering are { (1) (2) (3, 4, 5) (6, 7) (8) (9) (10, 11) (12) (13, 15) (14, 16)}. Next attribute based similarity analysis is performed for the COMMON variables. The results are given in Figure 4.13.

	Threshold value	Clusters
	> 0	(19) (21) (17, 18, 20, 24, 25, 26, 27, 28)
	> 0.1	(17) (19) (21) (18, 20, 24, 25, 26, 27, 28)
**	> 0.2	(17) (19) (21) (18, 20) (24, 25, 26, 27, 28)
	> 0.6	(17)(18) (19)(20) (21) (24, 25) (26, 27, 28)

Figure 4.13
Attribute-Based Similarity of COMMON variables

Next, method-based similarity is computed. The results are given in Figure 4.14.

	Threshold value	Clusters
	> 0	(17, 18, 19, 20, 21, 20, 24, 25, 26, 27, 28)
	> 0.1	(17, 18, 19, 20, 21, 20, 24, 25, 26, 27, 28)
	> 0.2	(17) (18, 21) (19, 28) (20, 24, 25, 26, 27)
**	> 0.4	(17) (18) (21) (19, 28) (20, 24, 25, 26, 27)
	> 0.9	(17) (18) (19) (21) (28) (20, 24, 25, 26, 27)

Figure 4.14
Method-Based Similarity of COMMON variables

Using the attribute clustering techniques the following clusters are formed: (17) (19) (21) (18, 20) (24, 25) (26, 27,28). Using the method clustering techniques, the following clusters are formed: (17) (18) (21) (19, 28) (24, 25) (20, 26, 27).

Because there are no identical objects, each of the objects extracted is abstracted as a class. The clusters obtained from combining threshold tables represent the clustering of the classes into the inheritance hierarchy. Either attribute-based clustering or method-based clustering is chosen and the corresponding clusters are used in the formation of the inheritance hierarchy. So, based on the clusters formed using method-based clustering, a parent class, say P1, is formed from classes 19 and 28. Likewise, P2 is formed from classes 24 and 25, and P3 is formed from classes 20, 26 and 27. Similarly for the clusters formed using attribute-based clustering.

After class abstraction is performed, the human reverse-engineer is given the option of attribute-based clustering or method-based clustering for the instantiation of the inheritance hierarchy. Choosing the attribute-based clustering for both parameter and global variable analysis results in the following inheritance hierarchy: (1) (2,6,12) (3,4,5) (7,8,9) (10,11) (13,15) (14,16) (17) (19) (21) (18, 20) (24, 25) (26, 27, 28). This means that classes (1) (17) (19) have no parent class. However, a parent class is abstracted for classes (2) (6) (12), namely, (2, 6, 12). Likewise, seven other parent classes are abstracted: (7, 8, 9) (10, 11) (13, 15) (14, 16) (18, 20) (24, 25) (26, 27, 28). A representative selection of the corresponding class templates follow in Figure 4.15. Classes were chosen to represent those that are formed of single objects (Class1 and Class2), as well as multiple objects

(Class2-6-12). Similarly, classes were chosen that have no parent class (Class1, Class17, and Class2-6-12) as well as that have a parent class (Class2). Additionally, the classes were chosen which represent varying numbers of attributes and methods.

```
Class1  is {object1}

        ParentClass  {}

        Attributes
                main.pnum  :  real

        Methods

                Method1
                 begin
                          READ(*,*) PNUM
                 end

Class2 is {object2}
        ParentClass is {Class2-6-12}

        Attributes
                vertex.node1,  vertex.num  :  integer
        Methods

                Method1
                 begin
                         IF (BNODE(EDGE) .EQ. NODE) THEN
                          NODE1 = FNODE(EDGE)
                        NUM = FLEDGE(EDGE)
                        IF(FREDGE(NUM).EQ. EDGE) THEN
                           FREDGE(NUM) = FREDGE(EDGE)
                        ENDIF
                        NUM = FREDGE(EDGE)
                        IF(FLEDGE(NUM).EQ. EDGE) THEN
                           FLEDGE(NUM) = FLEDGE(EDGE)
```

Figure 4.15
Class Templates

```
          ENDIF
      ELSE
          NODE1 = BNODE(EDGE)
          NUM =BLEDGE(EDGE)
          IF(BREDGE(NUM).EQ. EDGE) THEN
             BREDGE(NUM) = BREDGE(EDGE)
          ENDIF
          NUM = BREDGE(EDGE)
     ENDIF
    end
```

Class2-6-12 is {object2, object6, object12}
    ParentClass is {}

    Attributes
        vertex.node1, vertex.num : integer
        reduce.arc, reduce.node : integer
        initial.i, initial.num : integer

    Methods

        *** see object templates for object2, object6 and object 12 ***

Class17 is {object17}

    ParentClass {}

    Attributes
        fnode, bnode, lface, rface, dnode, dface, label : integer
        fledge, fredge, bredge, bledge, nn, na, nf, flag, incdnt: integer
        pos, neg, point, errpos, errneg : real

    Methods

        *** see object17 object template ***

The state interrogation templates represent which messages are passed among objects to interrogate state information. If an class references an attribute which is not an attribute or local variable of the class, then it is interrogating another class. Each method of every class is evaluated. The state interrogation templates are given in Figure 4.16.

A deeper analysis of the design indicates a statement coverage of 97% and displays the methodology's ability to detect dead code. Hence, the extraction process is complete and the object-oriented design and related design documents have been extracted from the FORTRAN source code.

Thus, the methodology is demonstrated successful for a system of moderate complexity. As expected, the objects and classes in this case study were more complex than those of the statistics case study. This is due to the increased

---

```
Class17
        Method9 interrogates Class18.toppar
        Method9 interrogates Class18.topser
        Method9 interrogates Class18.topvert
        Method9 interrogates Class18.toploop

Class18
        Method1 interrogates Class21.below
        Method2 interrogates Class17.dface
        Method2 interrogates Class17.dnode

Class19
        Method1 interrogates Class17.pos
        Method1 interrogates Class17.neg
        Method1 interrogates Class17.point
        Method3 interrogates Class17.pos
```

Figure 4.16
State Interrogation Templates

```
        Method3 interrogates Class17.neg
        Method3 interrogates Class17.point

Class20
        Method2 interrogates Class27.sore

Class21
        Method1 interrogates Class17.dface
        Method1 interrogates Class17.dnode
        Method1 interrogates Class18.topdel
        Method1 interrogates Class18.toppar
        Method1 interrogates Class18.topser
        Method1 interrogates Class18.topwye
        Method1 interrogates Class18.toploop

Class24
        Method1 interrogates Class17.nf

Class25
        Method1 interrogates Class17.nn

Class28
        Method1 interrogates Class17.bnode
        Method1 interrogates Class17.fnode
```

complexity in the original system itself, which the diagrams resulting from the methodology mirror accurately. One major factor in this complexity is the extensive use of COMMON variables. Their influence can be clearly seen in the state interrogation templates which tell when a class must interrogate the state of another class. This type of message passing is greatly increased by the use of COMMON variables in the original system. Additionally, the methodology was able to detect dead code in the system. Due to the magnitude of the original system, and the lack of documentation, there is a high probability that the dead code would have continued to go undetected under routine maintenance. Finally, the class hierarchy is also demonstrative of the complexity of the system. In a system of little or no complexity, there would have been fewer classes with a small or no hierarchy.

4.3 Summary

The two case studies in this chapter demonstrate varying degrees of difficulty. The statistics case study demonstrates the accuracy of the methodology for small systems. In addition to the validation of the extracted design using the metrics of statement coverage and functional equivalence, evaluation of the extracted design indicates that it exhibits the qualities expected. The graph case study demonstrates the scaleability of the methodology. This system is more complex because of increased lines of code and extensive use of global variables in the form of the COMMON block. Again, the results of the methodology are as expected. The extracted design includes a more sophisticated inheritance hierarchy. The extensive use of COMMON variables results in the necessity for numerous state interrogations, which is portrayed in the state interrogation diagrams. Thus, through the case studies, we have demonstrated that the methodology successfully extracts an object-oriented design from systems of varying complexity.

Chapter 5

Conclusions and Future Research

The goal of this research was to study the issues related to migrating legacy systems coded in imperative languages to the object-oriented paradigm and to develop reverse-engineering techniques to facilitate the migration. Section 5.1 summarizes the contributions resulting from this work. Section 5.2 describes future work.

5.1 Contributions

FORM facilitates the extraction of an object-oriented design from imperative FORTRAN code using graph theory combined with a data-driven approach. The contributions of the work include:

- The simultaneous paradigm shift and design extraction facilitates the migration of legacy systems to updated technologies, specifically the object-oriented paradigm.
- The extraction of a design from legacy code allows necessary maintenance to be performed at a higher level of abstraction, specifically the design level, thereby reducing current and future maintenance costs.
- System functionalities which are otherwise lost due to incomplete or inaccurate documentation are recovered.

- Issues related to the necessity of the development of software metrics specifically for reverse engineering are identified.

Thus, the research results in a comprehensive methodology, the development of which identified several open problems in the area of reverse engineering. Throughout the development of FORM are various contributions to the area of reverse engineering including the following:

- The construction of new algorithms, including algorithms to resolve actual parameter aliasing, determine placeholders in the COMMON block of FORTRAN, resolve the COMMON block of FORTRAN, extract attributes from formal parameters and global variables, extract methods for candidate objects, and form class clusters to abstract an inheritance hierarchy.
- The formulation and proof of lemmas which determine the class cardinality in the object model and the method cardinality for each class.

These contributions will now be related to the various phases of the methodology. The preprocessing phase, which prepares the code for extraction, required the development of algorithms to resolve the aliases in the actual parameters and the COMMON block.

The first phase of the methodology, Object Extraction, uses a data-driven approach to define candidate objects in the imperative code. This required the development and definition of such concepts as weighted adjacency matrix and thresholding. Additionally, modifications to the traditional program slicing were required to facilitate the definition of methods of the candidate objects.

Phase two, Class Abstraction, uses the concept of identical objects to facilitate the forming of classes from the candidate objects. The classes are partitioned into equivalence classes which are mapped into unique classes in the object model. This results in Lemma 3.1 which describes the relationship among the cardinality of the set of classes in the object model, the set of object equivalence classes, and the set of objects. To complete phase two, the methods are extracted. This is done by first determining the number of methods contained in a class and then instantiating the methods. This results in the development of Lemma 3.2 which describes the relationship between the number of methods of a class in the object model and a class in the object equivalence class. The corresponding lemmas are stated and proved.

Phase three, Abstraction of the Inheritance Hierarchy, defines the concept of similarity analysis of object classes and uses this analysis to develop a class clustering technique. The definition of these techniques involved the development of algorithms to calculate both attribute-based similarity and method-based similarity.

Thus, the research has resulted in the development of a methodology to extract object-oriented designs from imperative legacy systems, with specific attention given to FORTRAN. The benefits of using the methodology include: the ability to capture system functionality which may not be apparent due to poor system structure, and the reduction of future maintenance costs of the system as a direct effect of accurate system documentation and updated programming techniques.

5.2 Future Research

The research described in this dissertation has numerous potential future directions. At this point several research opportunities exist:

- The next phase in the reverse-engineering process involves the abstraction of the object-oriented design into a formal specification. This phase requires the investigation of the current formal specification languages for the object-oriented paradigm and possible extension of such languages.
- The development of techniques to translate the design into an object-oriented implementation would be a valuable extension.
- The research could be further extended by formulating an adaptation of the methodology to other imperative languages (such as COBOL) or to other paradigms.
- The development of metrics to measure features specifically related to reverse-engineered systems is a possible area of research.
- Related to the research of reverse-engineered system metrics is that of refinement. With a suite of metrics defined for a reverse-engineered design, the area of refinement of a design with respect to the metrics is envisioned as future research.

References

[Ache94] Achee, B.L. & Carver, Doris L. "A Greedy Approach to Object Identification in Imperative Code," *In Proceedings 3rd Workshop on Program Comprehension,* IEEE Computer Science Press, 1994, pp 4 – 11.

[Ache95] Achee, B. L. & Carver, Doris L. "Identification and Extraction of Objects in Legacy Code," *In Proceedings 1995 IEEE Aerospace Applications Conference*, 1995, pp181 – 190.

[Beck93] Beck, J. & Eichmann, D. "Program and Interface Slicing for Reverse Engineering" *Proceedings IEEE Workshop on Program Comprehension*, 1993, pp 54-63.

[Bend92] Bendusi, P. Cimitile, A. & DeCarlini, U. "Reverse Engineering Processes, Design Document Production, and Structure Charts" *In Journal of Systems and Software,* 1992, pp 225 – 245.

[Bigg89] Biggerstaff, T.j. "Design Recovery for Maintenance and Reuse" *Computer*, July 1989, pp 36 – 49.

[Bigg90] Biggerstaff, T.J. "Human-Oriented Conceptual Abstractions in the Reengineering of Software" *In IEEE Conference on Software Engineering*, 1990, pp 120.

[Bowe91] Bowen, J. "From Programs to Object Code and Back Again Using Logic Programming" *2487-TN-PRG-1044*, March 1991.

[Breu91] Breuer, P.T. & Lano, K. "Creating Specifications from Code: Reverse Engineering Techniques" *In Software Maintenance: Research and Practice*, Vol 3, 1991, pp 145 – 162.

[Bush90] Bush, E. "Software Re-engineering Position Statement" *In Proceedings IEEE Conference on Software Engineering*, 1990, pp 121.

[Byrn91] Byrne, E.J. "Software Reverse Engineering: A Case Study" *In Software Practice and Experience*, Vol 21(12), Dec 1991, pp 1349 – 1364.

[Canf92a] Canfora, G., Cimitile, A., & deCarlini, U., "A Logic-Based Approach to Reverse Engineering Tools Production" *IEEE Trans on Software Engineering*, Vol 18, No 12, Dec 1992, pp 1053-1063.

[Canf92b] Canfora G., & Cimitile, A. "Reverse Engineering and Intermodular Data Flow: A Theoretical Approach" *Software Maintenance: Research and Practice*, Vol 4, 1992, pp 37-59.

[Canf92c] Canfora G. Sansone, L. & Visaggio, G. "Data Flow Diagrams: Reverse Engineering Production and Animation" *In Proceedings IEEE Conference on Software Maintenance*, 1992, pp 366 – 375.

[Canf92d] Canfora, G., Cimitile, A. Munro, M. & Tortorella, M. "Experiments in Identifying Reusable Abstract Data Types in Program Code" *In Proceedings 2nd Annual Workshop on Program Comprehension*, 1992, pp 36 – 45.

[Canf93] Canfora, G. Cimitile, A., & Munro, M. "A Reverse Engineering Method for Identifying Reusable Abstract Data Types" *In Proceedings IEEE Working Conference on Reverse Engineering*, 1993, pp 73 – 82.

[Chid94] Chidamber, S.R., & Kemerer, C.F. "A Metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, Vol 20, No 6, June 1994, pp. 476-493.

[Chik90] Chikofsky, E.J., & Cross, J.H. "Reverse Engineering and Design Recovery: A Taxonomy" *In IEEE Software*, Jan 1990, pp 13 – 17.

[Choi90] Choi, S.C., & Sacchi, W. "Extracting and Restructuring the Design of Large Systems" *In IEEE Software*, Jan 1990, pp 66 – 71.

[Cimi91] Cimitile, a. & DeCarlini, U. "Reverse Engineering: Alorithms for Program Graph Production" *In Software Practice and Experience*, Vol 21 (5), May 1991, pp 519-537.

[Colb90] Colbrook, A., Smythe, C. & Darlinson,, A. "Data Abstraction in a Software Re-engineering Reference Model" *In Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, pp 2 – 11.

[Drom96] Dromey, R.G. "Cornering the Chimera," *IEEE Software*, Jan 1996, pp 33-43.

[Edwa93] Edwards, H.M. & Munro, M. "RECAST: Reverse Engineering from COBOL to SSADM Specifications" *In Proceedings of the 1993 IEEE International Conference on Software Maintenance*, 1993, pp 499-508.

[Engb93] Engberts, A., Kozacyznski, W., Liongosari, E., & Ning, J., "COBOL/SRE: A COBOL System Renovation Environment" *Proceedings IEEE International Conference on Software Maintenance*, 1993, pp 199-239.

[Gann93] Gannod, G.C., & Cheng, B.H.C. "A Two-Phase Approach to Reverse Engineering Using Formal Methods" *In Proceedings of Formal Methods in Programming and their Applications Conference*, June 1993, pp 335 – 348.

[Gill90] Gillis, K.D., & Wright, D.G. "Improving Software Maintenance Using System-Level Reverse Engineering" *In Proceedings IEEE Conference on Software Maintenance*, 1990, pp 84-90.

[Jaco91] Jacobson, I. 7 Lindstrom, F. " Re-engineering of Old Systems to an Object-Oriented Architecture" *In Proceedings OOPSLA 1991*, 1991, pp 340-350.

[Jone90] Jones, G.W., *Software Engineering*, New York: John Wiley & Sons, 1990.

[Kitc96] Kitchenham, B. & Pfleeger, S.L. "Software Quality: The Elusive Target," *IEEE Software*, Jan 1996, pp 12-21.

[Lano91a] Lano, K., Breuer,P. Haughton, H. "Reverse Engineering of Library Case Study," Technical Report, 2487-TN-PRG-1064, May 1994.

[Lano91b] Lano, K., Haughton, H. & Breuer, P. "Using Object Oriented Extensions of Z for Maintenance and Reverse Engineering," Technical Report PRG-TR-22-91, Nov 1991.

[Lano93] Lano, K., Breuer, P., Haugton, H. "Reverse Engineering COBOL via Formal Methods" *Software Maintenance: Research and Practice*, Vol 5, 1993, pp 13-35.

[Li95] Li, W., Henry, S., et.al. JOOP, July-Aug 1995, pp. 48-55.

[Liu90] Liu, Sying-Syang, Wilde,N. "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery" *In Proceedings Conference on Software Maintenance*, IEEE Computer Society Press, 1990, p,266-271.

[Liva92] Livadas, P.E. & Roy, P.K., "Program Dependence Analysis," *In Proceedings IEEE 2nd Workshop on Program Comprehension*, 1992, pp 356-365.

[Liva93] Livadas, P.E., & Alden, S.D., "A Toolset for Program Understanding," *In Proceedings of 3rd Workshop on Program Comprehension*, 1993, pp 110-118.

[Mull92] Muller, H.A., Wong, K., Tilley, S.R. "Understanding Software Systems Using Reverse Engineering Technology" Technical Report, University of Victoria, 1992.

[Ogan91] Ogando, R.M. "An Approach to Software System Modularization Based on Data and Type Bindings," PhD Dissertation, University of Florida, 1991.

[Ong93] Ong , C.L., Tsai, W.T., "Class and Object Extraction from Imperative Code" *Journal of Object Oriented Programming*, March/April 1993, pp 58-68.

[Osbo90] Osborne, W.M. & Chikofsky E.J., "Fitting Pieces to the Maintenance Puzzle," *IEEE Software*, Jan 1990, pp 11-12.

[Pfle91] Pfleeger, S.L. *Software Engineering: The Production of Quality Software*, New York: Macmillan, 1991.

[Rich90] Rich,C. & Willis, L.M. "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan 1990, pp 46-54.

[Rick89] Ricketts, J.A., DelMonaco, J.C., & Weeks, M.W., "Data Reengineering for Application Systems," *In Proceedings Conference on Software Maintenance*, 1989, pp 174-179.

[Ruga90] Tugaber, S., Ornburn, S.B. & LeBlanc, R.J., "Recognizing Design Decisions in Programs," *IEEE Software*, Jan 1990, pp.46 – 54.

[Ruga93] Rugaber, S., & Clayton, R. "The Representation Problem in Reverse Engineering," *Proceedings IEEE Workshop on Program Comprehension*, 1993, pp 8-16.

[Sitj91] Sijtsma, B. & Mager, J. "A Theory for Software Design Extraction" *In Proceedings 3rd European Software Engineering Conference*, 1991, pp.251-265.

[Snee92] Sneed, H.M., "Reverse Engineering versus Reenginering" *In Proceedings 2nd Workshop on Program Comprehension*, 1992, pp. 85-86.

[Somm89] Sommerville, I. *Software Engineering*, New York: Addison-Wesley, 1989.

[Swar94] Sward, R.E. & Steigerwald, R.A., "Issues in Re-Engineering from Procedural to Object-Oriented Code" *In Proceedings Conference on Software Maintenance*, 1994, pp 327-333.

[Subr96] Subramaniam, G.V. & Bryne, E.J., "Deriving an Object Model from Legacy Fortran Code" *In Proceedings International Conference on Software Maintenance*, 1996, pp 3-12.

[Till94] Tilley, S.R., Muller, H.A., et.al. "Domain-Retargetable Reverse Engineering" Technical Report, University of Victoria.

[Wate94] Waters, R.C. & Chikofsky, E.J., "Reverse Engineering: Progress Along Many Dimensions" *Communications of the ACM*, May 1994, Vol 37, No 5, pp.22-24.

[Weis84] Weiser, M. "Program Slicing" *IEEE Transactions on Software Engineering*, Vol SE-10, No 4, July 1984, pp. 352-357.

[Yang91] Yang, H. "The Supporting Environment for A Reverse Engineering System - The Maintainer's Assistant" *In Proceedings International Conference on Software Maintenance*, 1991, pp 13-22.

[Yu91] Yu, D. "A View on Three R's: Reuse, Re-engineering and Reverse Engineering" *ACMSIGSOFT Software Engineering Notes*, Vol 16, No 3, July 1991, pg. 69.

[Zimm90] Zimmer, J.A., "Restructuring for Style" *Software Practice and Experience*, Vol 20, No.4, April 1990, pp.365-389.

Appendix A1
Object Templates for Actual Parameters

```
Object1 is
        Attributes
                main.pnum  :  real

        Methods

                Method1
                 begin
                         READ(*,*) PNUM
                 end

        SubDerivation  {Main}

Object2 is
        Attributes
                vertex.node1,  vertex.num  :  integer


        Methods

                Method1
                 begin
                         IF (BNODE(EDGE) .EQ. NODE) THEN
                         NODE1 = FNODE(EDGE)
                        NUM = FLEDGE(EDGE)
                        IF(FREDGE(NUM).EQ. EDGE) THEN
                           FREDGE(NUM) = FREDGE(EDGE)
                        ENDIF
                        NUM = FREDGE(EDGE)
                        IF(FLEDGE(NUM).EQ. EDGE) THEN
                           FLEDGE(NUM) = FLEDGE(EDGE)
                         ENDIF
                 ELSE
                        NODE1 = BNODE(EDGE)
                        NUM =BLEDGE(EDGE)
                        IF(BREDGE(NUM).EQ. EDGE) THEN
                           BREDGE(NUM) = BREDGE(EDGE)
                         ENDIF
```

```
                    NUM = BREDGE(EDGE)
                ENDIF

        SubDerivation   {Vertex}


Object3 is
        Attributes
                loop.face1  :  integer

        Methods

                Method1
                 begin
                        IF (RFACE(EDGE) .EQ. FACE) THEN
                                FACE1 = LFACE(EDGE)
                         ELSE
                                FACE1 = RFACE(EDGE)
                         ENDIF
                 end

        SubDerivation  {Loop}


Object4 is
        Attributes
                loop.num1  :integer

        Methods

                Method1
                 begin
                       NUM1= BNODE(EDGE)
                 end

        SubDerivation  {Loop}

Object5 is
        Attributes
                loop.num  :  integer

        Methods
```

```
                Method1
                 begin
                      IF (RFACE(EDGE) .EQ. FACE) THEN
                            NUM = FLEDGE(EDGE)
                            IF(BLEDGE(NUM).EQ. EDGE) THEN
                              BLEDGE(NUM) = BLEDGE(EDGE)
                             ENDIF
                            NUM = BLEDGE(EDGE)
                       ELSE
                            NUM = BREDGE(EDGE)
                            IF(FREDGE(NUM).EQ. EDGE) THEN
                              FREDGE(NUM) = FREDGE(EDGE)
                            ENDIF
                            NUM = FREDGE(EDGE)
                       ENDIF
                 end

          SubDerivation {Loop}


     Object6 is
          Attributes
                reduce.arc, reduce.node  :  integer

          Methods

                Method1
                 begin
                10   IF (TOPLOOP .GT. 0) THEN
                       ARC = INCDNT(TOPLOOP)
                       GOTO 10
                     ENDIF
                     IF (TOPVERT .GT. 0) THEN
                       NODE = TOPVERT
                       ARC = INCDNT(TOPVERT)
                       GOTO 10
                     ENDIF
                     IF (TOPSER .GT. 0) THEN
                       NODE = TOPSER
                       ARC = INCDNT(TOPSER)
                      GOTO 10
                     ENDIF
                20   IF (TOPPAR .GT. 0) THEN
```

```
                 ARC = INCDNT(TOPPAR)
                GOTO 10
               ENDIF
            30  IF (TOPWYE .GT. 0) THEN
                  NODE = TOPWYE
                  ARC = INCDNT(TOPWYE)
                  GOTO 10
               ENDIF
            40  IF (TOPDEL .GT. 0) THEN
                  ARC = INCDNT(TOPDEL)
                  GOTO 10
               ENDIF
    end

   SubDerivation  {Reduce}


Object7 is
   Attributes
        reduce.face :  integer

   Methods

        Method1
         begin
        10  IF (TOPLOOP .GT. 0) THEN
              FACE = TOPLOOP
              GOTO 10
            ENDIF
        20  IF (TOPPAR .GT. 0) THEN
             FACE = TOPPAR
             GOTO 10
            ENDIF
        40  IF (TOPDEL .GT. 0) THEN
              FACE = TOPDEL
              GOTO 10
            ENDIF
    end

   SubDerivation  {Reduce}


Object8  is
```

```
Attributes
        series.num  :  integer

Methods

        Method1
          begin
               IF (FNODE(EDGE1) .EQ. NODE) THEN
                     EDGE2 = FLEDGE(EDGE1)
                     IF (FNODE(EDGE2) .EQ. NODE) THEN
                        FLEDGE(EDGE1) = BREDGE(EDGE2)
                      ELSE
                        FLEDGE(EDGE1) = FLEDGE(EDGE2)
                      ENDIF
                     NUM = FLEDGE(EDGE1)
                     IF (FREDGE(NUM) .EQ. EDGE2) THEN
                        FREDGE(NUM) = EDGE1
                      ENDIF
                     NUM = FREDGE(EDGE1)
                  ELSE
                     EDGE2 = BLEDGE(EDGE1)
                     IF (FNODE(EDGE2) .EQ. NODE) THEN
                        BREDGE(EDGE1) = BREDGE(EDGE2)
                        BLEDGE(EDGE1) = BLEDGE(EDGE2)
                     ELSE
                        BREDGE(EDGE1) = FLEDGE(EDGE2)
                        BLEDGE(EDGE1) = FREDGE(EDGE2)
                     ENDIF
                     NUM = BREDGE(EDGE1)
                     IF (BLEDGE(NUM) .EQ. EDGE2) THEN
                        BLEDGE(NUM) = EDGE1
                    ENDIF
                     NUM = BLEDGE(EDGE1)
                 NUM = RFACE(EDGE1)
                  IF (NUM .EQ. LFACE(EDGE1)) THEN
                  ELSE
                     NUM = LFACE(EDGE1)
                  ENDIF

        SubDerivation {Series}


Object9  is
```

Attributes

```
    parallel.num  :  integer
```

Methods

```
    Method1
      begin
          IF (RFACE(EDGE1) .EQ. FACE) THEN
                EDGE2 = FREDGE(EDGE1)
                IF (RFACE(EDGE2) .EQ. FACE) THEN
                   FREDGE(EDGE1) = BLEDGE(EDGE2)
                   BREDGE(EDGE1) = FLEDGE(EDGE2)
                ELSE
                   FREDGE(EDGE1) = FREDGE(EDGE2)
                   BREDGE(EDGE1) = BREDGE(EDGE2)
                ENDIF
                NUM = FREDGE(EDGE1)
                IF(BREDGE(NUM).EQ. EDGE2) THEN
                   BREDGE(NUM) = EDGE1
               ENDIF
                NUM = BREDGE(EDGE1)
            ELSE
                EDGE2 = FLEDGE(EDGE1)
                IF (RFACE(EDGE2) .EQ. FACE) THEN
                   FLEDGE(EDGE1) = FLEDGE(EDGE2)
                   BLEDGE(EDGE1) = BLEDGE(EDGE2)
               ELSE
                   FLEDGE(EDGE1) = BREDGE(EDGE2)
                   BLEDGE(EDGE1) = FREDGE(EDGE2)
                ENDIF
                NUM = FLEDGE(EDGE1)
                IF(BLEDGE(NUM).EQ. EDGE2) THEN
                   BLEDGE(NUM) = EDGE1
                ENDIF
                NUM = BLEDGE(EDGE1)
            ENDIF
            NUM = FNODE(EDGE1)
            IF (NUM .EQ. BNODE(EDGE1)) THEN
            ELSE
                NUM = BNODE(EDGE1)
             ENDIF
      end
```

```
          SubDerivation {Parallell}


Object10 is
          Attributes
                    wye.k, wye.da, wye.db, wye.dc, wye.wa, wye.wb, wye.wc : real

          Methods

                    Method1
                      begin
                           ARC(1) = EDGE
                           IF (BNODE(EDGE) .EQ. NODE) THEN
                                 ARC(2) = BREDGE(EDGE)
                                  ARC(3) = BLEDGE(EDGE)
                           ELSE
                                  ARC(2) = FLEDGE(EDGE)
                                  ARC(3) = FREDGE(EDGE)
                           ENDIF
                           DO 110 J = 1,10
                              DA = 1- POS(ARC(1),J)
                              DB = 1- POS(ARC(2),J)
                              DC = 1- POS(ARC(3),J)
                              RHO = DA*DB*DC - (DA+DB+DC) + 1
                              IF (RHO .GT. 0 ) THEN
                                 K = 1.0
                              ELSE
                                 K= 0.0
                              ENDIF
                              POS(ARC(1),J) = 1-WC
                              POS(ARC(2),J) = 1-WA
                              POS(ARC(3),J) = 1-WB
                              DA = 1- NEG(ARC(1),J)
                              DB = 1- NEG(ARC(2),J)
                              DC = 1- NEG(ARC(3),J)
                              RHO = DA*DB*DC - (DA+DB+DC) + 1
                              IF (RHO .GT. 0 ) THEN
                                 K = 0.0
                              ELSE
                                 K= 1.0
                                      ENDIF
                               DA = 1- POINT(ARC(1),J)
                              DB = 1- POINT(ARC(2),J)
```

```
                DC = 1- POINT(ARC(3),J)
                K = 0.5
                POINT(ARC(1),J) = 1-WC
                POINT(ARC(2),J) = 1-WA
                POINT(ARC(3),J) = 1-WB
          110   CONTINUE
      end


    Method2
      begin
         IF ( K .EQ. 0 ) THEN
                A1 = DA + DB*DC - DA*DB*DC
                B1 = DB + DA*DC - DA*DB*DC
                C1 = DC + DB*DA - DA*DB*DC
                WA = DSQRT(B1*C1/A1)
                WB = DSQRT(A1*C1/B1)
                WC = DSQRT(B1*A1/C1)
         ENDIF
         IF (DABS(WA-WB) .LE. 0.00001) THEN
         ELSE
                 WC = (1-DC)*(DB-DA)/(WA-WB)
         ENDIF
      end

  SubDerivation {Wye, Transform}


Object 11 is
  Attributes
    delta.k, delta.da, delta.db, delta.dc, delta.wa, delta.wb, delta.wc: real

  Methods


    Method1
      begin
         IF ( K .EQ. 0 ) THEN
                A1 = DA + DB*DC - DA*DB*DC
                B1 = DB + DA*DC - DA*DB*DC
                C1 = DC + DB*DA - DA*DB*DC
                WA = DSQRT(B1*C1/A1)
```

```
            WB = DSQRT(A1*C1/B1)
            WC = DSQRT(B1*A1/C1)
        ENDIF
        IF (DABS(WA-WB) .LE. 0.00001) THEN
        ELSE
                WC = (1-DC)*(DB-DA)/(WA-WB)
        ENDIF
    end

Method2
    begin
         ARC(1) = EDGE
            IF  (RFACE(EDGE) .EQ. FACE) THEN
                ARC(2) = FREDGE(EDGE)
                ARC(3) = BREDGE(EDGE)
            ELSE
                ARC(2) = BLEDGE(EDGE)
                ARC(3) = FLEDGE(EDGE)
            ENDIF
        DO  110 J =1,10
            DA =  POS(ARC(1),J)
            DB =  POS(ARC(2),J)
            DC =  POS(ARC(3),J)
            RHO = DA*DB*DC - (DA+DB+DC) + 1
            IF (RHO .GT. 0 ) THEN
               K = 0.0
            ELSE
               K= 1.0
            ENDIF
            POS(ARC(1),J) = WC
            POS(ARC(2),J) = WA
            POS(ARC(3),J) = WB
            DA =  NEG(ARC(1),J)
            DB =  NEG(ARC(2),J)
            DC =  NEG(ARC(3),J)
            RHO = DA*DB*DC - (DA+DB+DC) + 1
            IF (RHO .GT. 0 ) THEN
               K = 1.0
            ELSE
               K= 0.0
            ENDIF
            NEG(ARC(1),J) = WC
            NEG(ARC(2),J) = WA
```

```
                    NEG(ARC(3),J) = WB
                    DA = POINT(ARC(1),J)
                    DB = POINT(ARC(2),J)
                    DC = POINT(ARC(3),J)
                    K = 0.5
                    POINT(ARC(1),J) = WC
                    POINT(ARC(2),J) = WA
                    POINT(ARC(3),J) = WB
               110    CONTINUE
            end

        SubDerivation  {Transform, Delta}


    Object12  is
        Attributes
            initial.i, initial.num  :  integer

        Methods

            Method1
              begin
                DO  10 J= 1,NA
                      I = ARCNUM(J)
                      NUM = BNODE(I)
                      IF (INCDNT(NUM) .NE. 0)  GOTO 20
               20      NUM = FNODE(I)
                      IF (INCDNT(NUM) .NE. 0)  GOTO 30
               30      NUM = LFACE(I)
                      IF (INCDNT(NUM) .NE. 0)  GOTO 40
               40      NUM = RFACE(I)
               10  CONTINUE
              end

        SubDerivation  {Initial}

    Object13  is
        Attributes
            wye.num   :  integer

        Methods

            Method1
```

```
begin
    ARC(1) = EDGE
       IF  (BNODE(EDGE) .EQ. NODE) THEN
          ARC(2) = BREDGE(EDGE)
          ARC(3) = BLEDGE(EDGE)
       ELSE
          ARC(2) = FLEDGE(EDGE)
          ARC(3) = FREDGE(EDGE)
       ENDIF
     DO  10 I = 1,3
         ROT(I) = MOD(I,3) + 1
         IF (BNODE(ARC(I)) .EQ. NODE) THEN
             TEMP(1,I) = FNODE(ARC(I))
             TEMP(2,I) = RFACE(ARC(I))
             TEMP(3,I) = FREDGE(ARC(I))
             TEMP(4,I) = FLEDGE(ARC(I))
         ELSE
             TEMP(1,I) = BNODE(ARC(I))
             TEMP(2,I) = LFACE(ARC(I))
             TEMP(3,I) = BLEDGE(ARC(I))
             TEMP(4,I) = BREDGE(ARC(I))
         ENDIF
             TEMP(5,I) = LABEL(ARC(I))
   10  CONTINUE
       DO   20  I = 1,3
         LFACE(ARC(I)) = TEMP(2,I)
         FNODE(ARC(I)) = TEMP(1,ROT(I))
        FLEDGE(ARC(I)) = TEMP(4,ROT(I))
   20  CONTINUE
        DO  30 I = 1,3
          NUM = LFACE(ARC(I))
    30  CONTINUE
        DO   40 I=1,3
          NUM = FNODE(ARC(I))
    40  CONTINUE
end
```

SubDerivation {Wye}

Object14
    Attributes
        delta.num : integer

Methods

```
Method1
  begin
       ARC(1) = EDGE
       IF  (RFACE(EDGE) .EQ. FACE) THEN
               ARC(2) = FREDGE(EDGE)
               ARC(3) = BREDGE(EDGE)
       ELSE
               ARC(2) = BLEDGE(EDGE)
               ARC(3) = FLEDGE(EDGE)
       ENDIF
       DO  10 I = 1,3
             IF (RFACE(ARC(I)) .EQ. FACE) THEN
                  TEMP(1,I) = FNODE(ARC(I))
                  TEMP(2,I) = LFACE(ARC(I))
                  TEMP(3,I) = FLEDGE(ARC(I))
                  TEMP(4,I) = BLEDGE(ARC(I))
             ELSE
                  TEMP(1,I) = BNODE(ARC(I))
                  TEMP(2,I) = RFACE(ARC(I))
                  TEMP(3,I) = BREDGE(ARC(I))
                  TEMP(4,I) = FREDGE(ARC(I))
             ENDIF
                  TEMP(5,I) = LABEL(ARC(I))
   10   CONTINUE
         DO  20  I = 1,3
             FNODE(ARC(I)) = TEMP(1,I)
             RFACE(ARC(I)) = TEMP(2,ROT(I))
             FREDGE(ARC(I)) = TEMP(4,ROT(I))
   20   CONTINUE
         DO 25 I = 1,3
            NUM =RFACE(ARC(I))
    25 CONTINUE

           DO  30 I = 1,3
             NUM = RFACE(ARC(I))
       30   CONTINUE
        DO  40 I = 1,3
             NUM = FNODE(ARC(I))
       40   CONTINUE
  end
```

```
        SubDerivation  {Delta}


    Object15   is
        Attributes
              wye.arc(i)  :  integer

        Methods

              Method1
                begin
                      ARC(1) = EDGE
                         IF  (BNODE(EDGE) .EQ. NODE) THEN
                            ARC(2) = BREDGE(EDGE)
                            ARC(3) = BLEDGE(EDGE)
                         ELSE
                            ARC(2) = FLEDGE(EDGE)
                            ARC(3) = FREDGE(EDGE)
                         ENDIF
                 end

        SubDerivation  {Wye}


    Object16 is
        Attributes
              delta.arc(i)  :  integer

        Methods
              Method1
                  begin
                     ARC(1) = EDGE
                         IF  (RFACE(EDGE) .EQ. FACE) THEN
                            ARC(2) = FREDGE(EDGE)
                            ARC(3) = BREDGE(EDGE)
                         ELSE
                            ARC(2) = BLEDGE(EDGE)
                            ARC(3) = FLEDGE(EDGE)
                         ENDIF

                end

        SubDerivation  {Delta}
```

Appendix A2
Object Templates for COMMON Variables

Object 17 is

Attributes	fnode, bnode, lface, rface, dnode, dface, label : integer fledge, fredge, bredge, bledge, nn, na, nf, flag, incdnt: integer pos, neg, point, errpos, errneg : real

Methods

```
Method1
  begin
     IF (BNODE(EDGE) .EQ. NODE) THEN
          NODE1 = FNODE(EDGE)
          NUM = FLEDGE(EDGE)
          IF(FREDGE(NUM).EQ. EDGE) THEN
            FREDGE(NUM) = FREDGE(EDGE)
          ELSE
            BLEDGE(NUM) = FREDGE(EDGE)
          ENDIF
          NUM = FREDGE(EDGE)
          IF(FLEDGE(NUM).EQ. EDGE) THEN
            FLEDGE(NUM) = FLEDGE(EDGE)
          ELSE
            BREDGE(NUM) = FLEDGE(EDGE)
          ENDIF
        ELSE
          NODE1 = BNODE(EDGE)
          NUM =BLEDGE(EDGE)
          IF(BREDGE(NUM).EQ. EDGE) THEN
            BREDGE(NUM) = BREDGE(EDGE)
          ELSE
            FLEDGE(NUM) = BREDGE(EDGE)
          ENDIF
                NUM = BREDGE(EDGE)
          IF(BLEDGE(NUM).EQ. EDGE) THEN
            BLEDGE(NUM) = BLEDGE(EDGE)
          ELSE
            FREDGE(NUM) = BLEDGE(EDGE)
          ENDIF
```

```
            ENDIF
         DNODE(NODE1)= DNODE(NODE1)-1
         DFACE(NUM1) = DFACE(NUM1)-2
         DNODE(NODE) = 0
         LABEL(EDGE) = 0
         NN=NN-1
         NA=NA-1
     end


Method2
   begin
         IF (RFACE(EDGE) .EQ. FACE) THEN
               IF(BLEDGE(NUM).EQ. EDGE) THEN
                  BLEDGE(NUM) = BLEDGE(EDGE)
               ELSE
                  FREDGE(NUM) = BLEDGE(EDGE)
               ENDIF
               NUM = BLEDGE(EDGE)
               IF(FLEDGE(NUM).EQ. EDGE) THEN
                  FLEDGE(NUM) = FLEDGE(EDGE)
               ELSE
                  BREDGE(NUM) = FLEDGE(EDGE)
               ENDIF
            ELSE
               NUM = BREDGE(EDGE)
               IF(FREDGE(NUM).EQ. EDGE) THEN
                  FREDGE(NUM) = FREDGE(EDGE)
               ELSE
                  BLEDGE(NUM) = FREDGE(EDGE)
               ENDIF
               NUM = FREDGE(EDGE)
               IF(BREDGE(NUM).EQ. EDGE) THEN
                  BREDGE(NUM) = BREDGE(EDGE)
               ELSE
                 FLEDGE(NUM) = BREDGE(EDGE)
               ENDIF
            ENDIF
            DFACE(FACE1)= DFACE(FACE1)-1
            DNODE(NUM1) = DNODE(NUM1)-2
            DFACE(FACE) = 0
           LABEL(EDGE) = 0
           NF=NF-1
```

```
          NA=NA-1
    end


Method3
    begin
          ARC(1) = EDGE
          IF  (BNODE(EDGE) .EQ. NODE) THEN
                  ARC(2) = BREDGE(EDGE)
                   ARC(3) = BLEDGE(EDGE)
         ELSE
                   ARC(2) = FLEDGE(EDGE)
                   ARC(3) = FREDGE(EDGE)
              ENDIF
         DO  10 I = 1,3
                  ROT(I) = MOD(I,3) + 1
                  IF (BNODE(ARC(I)) .EQ. NODE) THEN
                       TEMP(1,I) = FNODE(ARC(I))
                       TEMP(2,I)= RFACE(ARC(I))
                       TEMP(3,I) = FREDGE(ARC(I))
                       TEMP(4,I) = FLEDGE(ARC(I))
                  ELSE
                       TEMP(1,I) = BNODE(ARC(I))
                       TEMP(2,I) = LFACE(ARC(I))
                       TEMP(3,I) = BLEDGE(ARC(I))
                       TEMP(4,I) = BREDGE(ARC(I))
                  ENDIF
                       TEMP(5,I) = LABEL(ARC(I))
         10   CONTINUE
              DO  20  I = 1,3
                  RFACE(ARC(I)) = NEWFAC
                  BNODE(ARC(I)) = TEMP(1,I)
                  LFACE(ARC(I)) = TEMP(2,I)
                  FNODE(ARC(I)) = TEMP(1,ROT(I))
                  BLEDGE(ARC(I)) = TEMP(3,I)
                  FLEDGE(ARC(I)) = TEMP(4,ROT(I))
                  FREDGE(ARC(I)) = ARC(ROT(I))
                  BREDGE(ARC(I)) = ARC(ROT(ROT(I)))
                LABEL(ARC(I)) = MIN(TEMP(5,I),TEMP(5,ROT(I)))
         20  CONTINUE
                   DFACE(NEWFAC) = 3
              DO 25  I = 1,3
                   NUM = FLEDGE(ARC(I))
                IF(LFACE(NUM) .EQ. TEMP(2,I)) THEN
                  BLEDGE(NUM) = ARC(I)
                ELSE
                  FREDGE(NUM) = ARC(I)
                ENDIF
        25    CONTINUE
               DO 30 I = 1,3
                  NUM = LFACE(ARC(I))
                   DFACE(NUM) = DFACE(NUM) - 1
        30    CONTINUE
               DO  40 I=1,3
                  NUM = FNODE(ARC(I))
                  DNODE(NUM) = DNODE(NUM)+ 1
        40    CONTINUE
               DO  110 J = 1,10
                   DA = 1- POS(ARC(1),J)
                   DB = 1- POS(ARC(2),J)
                   DC = 1- POS(ARC(3),J)
                   POS(ARC(1),J) = 1-WC
                   POS(ARC(2),J) = 1-WA
                   POS(ARC(3),J) = 1-WB
                   ERRPOS(J) =
ERRPOS(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*DA*DB*DC)
                   DA = 1- NEG(ARC(1),J)
                   DB = 1- NEG(ARC(2),J)
                   DC = 1- NEG(ARC(3),J)
                   NEG(ARC(1),J) = 1-WC
                   NEG(ARC(2),J) = 1-WA
                   NEG(ARC(3),J) = 1-WB

ERRNEG(J)=ERRNEG(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*DA*
DB*DC)
                  DA = 1- POINT(ARC(1),J)
                   DB = 1- POINT(ARC(2),J)
                   DC = 1- POINT(ARC(3),J)
                   POINT(ARC(1),J) = 1-WC
                   POINT(ARC(2),J) = 1-WA
                   POINT(ARC(3),J) = 1-WB
        110     CONTINUE
                  DNODE(NODE) = 0
                   NN=NN-1
                   NF=NF+1
      end
```

```
Method4
   begin
        ARC(1) = EDGE
           IF  (RFACE(EDGE) .EQ. FACE) THEN
               ARC(2) = FREDGE(EDGE)
               ARC(3) = BREDGE(EDGE)
           ELSE
               ARC(2) = BLEDGE(EDGE)
               ARC(3) = FLEDGE(EDGE)
           ENDIF
           DO  10 I = 1,3
              ROT(I) = MOD(I,3) + 1
              IF (RFACE(ARC(I)) .EQ. FACE) THEN
                   TEMP(1,I) = FNODE(ARC(I))
                   TEMP(2,I) = LFACE(ARC(I))
                   TEMP(3,I)= FLEDGE(ARC(I))
                   TEMP(4,I) = BLEDGE(ARC(I))
              ELSE
                   TEMP(1,I) = BNODE(ARC(I))
                   TEMP(2,I) = RFACE(ARC(I))
                   TEMP(3,I) = BREDGE(ARC(I))
                   TEMP(4,I) = FREDGE(ARC(I))
              ENDIF
                   TEMP(5,I) = LABEL(ARC(I))
           DO  20  I = 1,3
              BNODE(ARC(I)) = NEWNOD
              FNODE(ARC(I)) = TEMP(1,I)
              LFACE(ARC(I)) = TEMP(2,I)
              RFACE(ARC(I)) = TEMP(2,ROT(I))
              FLEDGE(ARC(I)) = TEMP(3,I)
              FREDGE(ARC(I)) = TEMP(4,ROT(I))
              BREDGE(ARC(I)) = ARC(ROT(I))
              BLEDGE(ARC(I)) = ARC(ROT(ROT(I)))
       LABEL(ARC(I)) = MIN(TEMP(5,I),TEMP(5,ROT(I)))
       20  CONTINUE
            DNODE(NEWNOD) = 3
           DO 25  I = 1,3
               NUM = FREDGE(ARC(I))
               IF(FNODE(NUM) .EQ. TEMP(1,I)) THEN
                  FLEDGE(NUM) = ARC(I)
               ELSE
```

```
                BREDGE(NUM) = ARC(I)
             ENDIF
   25     CONTINUE
          DO 30 I = 1,3
            NUM = RFACE(ARC(I))
            DFACE(NUM) = DFACE(NUM)+ 1
   30     CONTINUE
          DO 40 I = 1,3
            NUM = FNODE(ARC(I))
            DNODE(NUM) = DNODE(NUM) - 1
   40     CONTINUE
          DO 110 J =1,10
          DA = POS(ARC(1),J)
          DB = POS(ARC(2),J)
          DC = POS(ARC(3),J)
          POS(ARC(1),J) = WC
        · POS(ARC(2),J) = WA
          POS(ARC(3),J) = WB

ERRPOS(J)=ERRPOS(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*DA*
DB*DC)
          DA = NEG(ARC(1),J)
          DB = NEG(ARC(2),J)
          DC = NEG(ARC(3),J)
          NEG(ARC(1),J) = WC
          NEG(ARC(2),J) = WA
          NEG(ARC(3),J) = WB
 ERRNEG(J)=ERRNEG(J)+DABS(WA*WB*WC-DA*DB-DA*DC-DB*DC+2*DA*
DB*DC)
          DA = POINT(ARC(1),J)
          DB = POINT(ARC(2),J)
          DC = POINT(ARC(3),J)
          POINT(ARC(1),J) = WC
          POINT(ARC(2),J) = WA
          POINT(ARC(3),J) = WB
   110     CONTINUE
          DFACE(FACE) = 0
          NF=NF-1
          NN=NN+1
      end
```

```
Method5
  begin
      IF (RFACE(EDGE1) .EQ. FACE) THEN
           EDGE2 = FREDGE(EDGE1)
           IF (RFACE(EDGE2) .EQ. FACE) THEN
             FREDGE(EDGE1) = BLEDGE(EDGE2)
             BREDGE(EDGE1) = FLEDGE(EDGE2)
             RFACE(EDGE1) = LFACE(EDGE2)
           ELSE
             FREDGE(EDGE1) = FREDGE(EDGE2)
             BREDGE(EDGE1) = BREDGE(EDGE2)
             RFACE(EDGE1) = RFACE(EDGE2)
           ENDIF
           NUM = FREDGE(EDGE1)
           IF(BREDGE(NUM).EQ. EDGE2) THEN
             BREDGE(NUM) = EDGE1
           ELSE
             FLEDGE(NUM) = EDGE1
           ENDIF
           NUM = BREDGE(EDGE1)
           IF(FREDGE(NUM).EQ. EDGE2) THEN
             FREDGE(NUM) = EDGE1
           ELSE
             BLEDGE(NUM) = EDGE1
           ENDIF
         ELSE
           EDGE2 = FLEDGE(EDGE1)
           IF (RFACE(EDGE2) .EQ. FACE) THEN
             FLEDGE(EDGE1) = FLEDGE(EDGE2)
             BLEDGE(EDGE1) = BLEDGE(EDGE2)
             LFACE(EDGE1) = LFACE(EDGE2)
           ELSE
             FLEDGE(EDGE1) = BREDGE(EDGE2)
             BLEDGE(EDGE1) = FREDGE(EDGE2)
             LFACE(EDGE1) = RFACE(EDGE2)
           ENDIF
           NUM = FLEDGE(EDGE1)
           IF(BLEDGE(NUM).EQ. EDGE2) THEN
             BLEDGE(NUM) = EDGE1
           ELSE
             FREDGE(NUM) = EDGE1
           ENDIF
```

```
                        NUM = BLEDGE(EDGE1)
                        IF(FLEDGE(NUM).EQ. EDGE2) THEN
                           FLEDGE(NUM) = EDGE1
                        ELSE
                           BREDGE(NUM) = EDGE1
                        ENDIF
                     ENDIF
                LABEL(EDGE1) = MIN(LABEL(EDGE1),LABEL(EDGE2))
                     LABEL(EDGE2) = 0
                     NUM = FNODE(EDGE1)
                     DNODE(NUM) = DNODE(NUM)-1
                 1000 IF (RFACE(EDGE2) .EQ.FACE) THEN
                       IF(INCDNT(LFACE(EDGE2)) .GT. 0)
                            INCDNT(LFACE(EDGE2)) = EDGE1
                    ELSE
                       IF(INCDNT(RFACE(EDGE2)) .GT. 0)
                        INCDNT(RFACE(EDGE2)) = EDGE1
                    ENDIF
                    DFACE(FACE)=0
                    DO 110 J = 1,10
POS(EDGE1,J)=
          POS(EDGE1,J)+POS(EDGE2,J)-POS(EDGE1,J)*POS(EDGE2,J)
             POINT(EDGE1,J)=POINT(EDGE1,J)+POINT(EDGE2,J)-
                                  POINT(EDGE1,J)*POINT(EDGE2,J)
   NEG(EDGE1,J) =
      NEG(EDGE1,J)+NEG(EDGE2,J)-NEG(EDGE1,J)*NEG(EDGE2,J)
                 110   CONTINUE
                    NF=NF-1
                    NA=NA-1
            end


          Method6
            begin
                IF (FNODE(EDGE1) .EQ. NODE) THEN
                     EDGE2 = FLEDGE(EDGE1)
                     IF (FNODE(EDGE2) .EQ. NODE) THEN
                        FLEDGE(EDGE1) = BREDGE(EDGE2)
                        FREDGE(EDGE1) = BLEDGE(EDGE2)
                        FNODE(EDGE1) = BNODE(EDGE2)
                     ELSE
                        FLEDGE(EDGE1) = FLEDGE(EDGE2)
                        FREDGE(EDGE1) = FREDGE(EDGE2)
            FNODE(EDGE1) = FNODE(EDGE2)
         ENDIF
         NUM = FLEDGE(EDGE1)
         IF (FREDGE(NUM) .EQ. EDGE2) THEN
            FREDGE(NUM) = EDGE1
         ELSE
            BLEDGE(NUM) = EDGE1
         ENDIF
         NUM = FREDGE(EDGE1)
         IF (FLEDGE(NUM) .EQ. EDGE2) THEN
            FLEDGE(NUM) = EDGE1
         ELSE
            BREDGE(NUM) = EDGE1
         ENDIF
      ELSE
         EDGE2 = BLEDGE(EDGE1)
         IF (FNODE(EDGE2) .EQ. NODE) THEN
            BREDGE(EDGE1) = BREDGE(EDGE2)
            BLEDGE(EDGE1) = BLEDGE(EDGE2)
            BNODE(EDGE1) = BNODE(EDGE2)
         ELSE
            BREDGE(EDGE1) = FLEDGE(EDGE2)
            BLEDGE(EDGE1) = FREDGE(EDGE2)
            BNODE(EDGE1) = FNODE(EDGE2)
         ENDIF
         NUM = BREDGE(EDGE1)
         IF (BLEDGE(NUM) .EQ. EDGE2) THEN
            BLEDGE(NUM) = EDGE1
         ELSE
            FREDGE(NUM) = EDGE1
         ENDIF
         NUM = BLEDGE(EDGE1)
         IF (BREDGE(NUM) .EQ. EDGE2) THEN
            BREDGE(NUM) = EDGE1
         ELSE
            FLEDGE(NUM) = EDGE1
         ENDIF
      ENDIF
LABEL(EDGE1) = MIN(LABEL(EDGE1),LABEL(EDGE2))
      LABEL(EDGE2) = 0
      NUM = RFACE(EDGE1)
      DFACE(NUM) = DFACE(NUM)-1
  1000 IF (BNODE(EDGE2) .EQ.NODE) THEN
```

```
            IF(INCDNT(FNODE(EDGE2)) .GT. 0)
             INCDNT(FNODE(EDGE2)) = EDGE1
          ELSE
            IF(INCDNT(BNODE(EDGE2)) .GT. 0)
             INCDNT(BNODE(EDGE2)) = EDGE1
          ENDIF
          DNODE(NODE) = 0
          DO 110 J= 1,10
               POS(EDGE1,J) = POS(EDGE1,J)*POS(EDGE2,J)
              NEG(EDGE1,J) = NEG(EDGE1,J)*NEG(EDGE2,J)
  POINT(EDGE1,J) = POINT(EDGE1,J)*POINT(EDGE2,J)
      110   CONTINUE
          NN=NN-1
          NA=NA-1
end


Method7
  begin
      READ(4,*)  NN, NA, NF
       DO 15 J=1, NA
      READ(4,*) I,BNODE(I),FNODE(I),LFACE(I),RFACE(I)
READ(4,*) BREDGE(I),BLEDGE(I),FLEDGE(I),FREDGE(I)
            ARCNUM(J) = I
      15   CONTINUE
          DO 16 J=1,NN
              READ(4,*) I, DNODE(I)
              NODNUM(J) = I
      16   CONTINUE
          DO 17 K= 1,NF
              READ(4,*) I, DFACE(I)
              FACNUM(K) = I
      17   CONTINUE
          READ(*,*) PNUM
          DO 110 J = 1,10
            DO 47 I = 1,NA
             POS(I,J) = PNUM + J*0.02
             NEG(I,J) = PNUM + J*0.02
             POINT(I,J) =PNUM + J*0.02
      47      CONTINUE
      110    CONTINUE
          LABEL(START) = 1
```

```
              EBOT = START
               OCURR=START
         1111  IF (BNODE(OCURR) .EQ. SORE) THEN
                  SFACE(OCURR) = RFACE(OCURR)
                  OCURR = BREDGE(OCURR)
               ELSE
                  SFACE(OCURR) = LFACE(OCURR)
                  OCURR = FLEDGE(OCURR)
               ENDIF
               IF (OCURR .NE. START) THEN
                 LABEL(OCURR) = 1
                 NCOUNT = NCOUNT + 1
                 NEXT(EBOT) =OCURR
                 EBOT = OCURR
                 NEXT(EBOT) = 0
                 GOTO 1111
             . ENDIF
         137  ESCAN = ETOP
               CFACE = SFACE(ESCAN)
               WRITE(*,*) 'SCANNING EDGE #', ESCAN
               ECURR = ESCAN
         122  IF (CFACE .EQ. RFACE(ECURR)) THEN
                  NUM = FNODE(ECURR)
                  ECURR = FREDGE(ECURR)
               ELSE
                  NUM = BNODE(ECURR)
                  ECURR = BLEDGE(ECURR)
               ENDIF
               IF (LABEL(ECURR) .EQ. 0) THEN
                   LABEL(ECURR) = LABEL(ESCAN) + 1
                   NCOUNT = NCOUNT + 1
 WRITE(*,*) 'labelling edge #', ECURR,'by label', LABEL(ECURR)
                   IF (NCOUNT .EQ. NA) GOTO 1333
                   SNODE(ECURR) = NUM
                   IF (OTOP .EQ. 0 ) THEN
                       OTOP = ECURR
                   ELSE
                       NEXT(OBOT) = ECURR
                   ENDIF
                   OBOT = ECURR
                   NEXT(OBOT) = 0
               ENDIF
               IF (LABEL(ECURR) .EQ. LABEL(ESCAN)) THEN
```

```
                  ETOP = NEXT(ETOP)
                  IF (ETOP .EQ. 0) THEN
                      IF(OTOP .EQ. 0) THEN
                        GOTO 1333
                      ELSE
                        GOTO 237
                    ENDIF
                  ENDIF
        GOTO 137
           ENDIF
           GOTO 122
  237   OSCAN = OTOP
          CNODE = SNODE(OSCAN)
          OCURR = OSCAN
  222   IF (CNODE .EQ. BNODE(OCURR)) THEN
             NUM = LFACE(OCURR)
             OCURR = BLEDGE(OCURR)
          ELSE
             NUM = RFACE(OCURR)
             OCURR = FREDGE(OCURR)
          ENDIF
          IF (LABEL(OCURR) .EQ. 0) THEN
              NCOUNT = NCOUNT + 1
              LABEL(OCURR) = LABEL(OSCAN) + 1
WRITE(*,*) 'labelling edge #', OCURR,'by label', LABEL(OCURR)
              IF (NCOUNT .EQ. NA) GOTO 1333
              SFACE(OCURR) = NUM
              IF (ETOP .EQ. 0 ) THEN
                  ETOP = OCURR
              ELSE
                  NEXT(EBOT) = OCURR
              ENDIF
              EBOT = OCURR
              NEXT(EBOT) = 0
          ENDIF
          IF (LABEL(OCURR) .EQ. LABEL(OSCAN)) THEN
               OTOP = NEXT(OTOP)
             NEXT(OSCAN) = 0
               IF (OTOP .EQ. 0) THEN
                   IF(ETOP .EQ. 0) THEN
                    GOTO 1333
                   ELSE
                     GOTO 137
                    ENDIF
                 ENDIF
                  GOTO 237
              ENDIF
              GOTO 222
     end


   Method8
     begin
         INCDNT(NUM) = 0
              ABOVE(BELOW(NUM)) = ABOVE(NUM)
              BELOW(ABOVE(NUM)) = BELOW(NUM)
              ABOVE(NUM) = 0
              BELOW(NUM) = 0
     end


   Method9
    begin
        IF (FLAG .EQ. 1) THEN
              FACE = NUM
              IF(DFACE(FACE) .EQ. 1) THEN
                 IF(TOPLOOP .GT. 0) THEN
                    ABOVE(TOPLOOP)=FACE
                    BELOW(FACE) =TOPLOOP
                 ENDIF
                 TOPLOOP = FACE
                 INCDNT(FACE) = ARC
                 GOTO 1000
              ENDIF
              IF(DFACE(FACE) .EQ. 2) THEN
                 IF(TOPPAR .GT. 0) THEN
                    ABOVE(TOPPAR)=FACE
                    BELOW(FACE) =TOPPAR
                 ENDIF
                 TOPPAR = FACE
                 INCDNT(FACE) = ARC
                 GOTO 1000
              ENDIF
              EDGE1= ARC
              IF (RFACE(EDGE1) .EQ. FACE) THEN
```

```
                        EDGE2 = FREDGE(EDGE1)
                        EDGE3 = BREDGE(EDGE1)
                     ELSE
                        EDGE2 = BLEDGE(EDGE1)
                        EDGE3 = FLEDGE(EDGE1)
                     ENDIF
      NCHECK = MIN(LABEL(EDGE1),LABEL(EDGE2),LABEL(EDGE3))
      NSUM =LABEL(EDGE1)+LABEL(EDGE2)+LABEL(EDGE3)
                     IF (NSUM .EQ. (3*NCHECK+2)) THEN
                        INCDNT(FACE) = EDGE1
                        IF (TOPDEL .GT. 0) THEN
                          ABOVE(TOPDEL)=FACE
                          BELOW(FACE) =TOPDEL
                        ENDIF
                        TOPDEL = FACE
                     ENDIF
                     GOTO 1000
                  ENDIF
                     NODE = NUM
                     IF(NODE .EQ. SORE)  GOTO 1000
                     IF(NODE .EQ. TERM1)  GOTO 1000
                     IF(NODE .EQ. TERM2)  GOTO 1000
                     IF(NODE .EQ. TERM3)  GOTO 1000
                     IF(NODE .EQ. TERM4 )  GOTO 1000
                     IF (DNODE(NODE) .EQ. 0) GOTO 1000
                     IF(DNODE(NODE) .EQ. 1) THEN
                        IF(TOPVERT .GT. 0) THEN
                           ABOVE(TOPVERT)=NODE
                           BELOW(NODE) =TOPVERT
                        ENDIF
                        TOPVERT = NODE
                        INCDNT(NODE) = ARC
                        GOTO 1000
                     ENDIF
                     IF(DNODE(NODE) .EQ. 2) THEN
                        IF(TOPSER .GT. 0) THEN
                           ABOVE(TOPSER)=NODE
                           BELOW(NODE) =TOPSER
                        ENDIF
                        TOPSER = NODE
                        INCDNT(NODE) = ARC
                        GOTO 1000
                     ENDIF
```

```
                                EDGE1= ARC
                                IF (BNODE(EDGE1) .EQ. NODE) THEN
                                  EDGE2 = BREDGE(EDGE1)
                                  EDGE3 = BLEDGE(EDGE1)
                                ELSE
                                  EDGE2 = FLEDGE(EDGE1)
                                  EDGE3 = FREDGE(EDGE1)
                                ENDIF
                NCHECK = MIN(LABEL(EDGE1),LABEL(EDGE2),LABEL(EDGE3))
                NSUM =LABEL(EDGE1)+LABEL(EDGE2)+LABEL(EDGE3)
                                IF (NSUM .EQ. (3*NCHECK+2)) THEN
                                  INCDNT(NODE) = EDGE1
                                   IF (TOPWYE .GT. 0) THEN
                                     ABOVE(TOPWYE)=NODE
                                     BELOW(NODE) =TOPWYE
                                   ENDIF
                                   TOPWYE = NODE
                                ENDIF
                    end

            SubDerivation {vertex, loop, wye, delta, parallel, series,remove, positive, main}



            Object18 is
                Attributes
                    topdel, topser, toppar, topvert, toploop :integer

                Methods

                    Method1
                      begin
                          IF (TOPPAR .EQ. NUM) TOPPAR = BELOW(NUM)
                          IF (TOPDEL .EQ. NUM) TOPDEL = BELOW(NUM)
                          IF (TOPLOOP .EQ. NUM) TOPLOOP = BELOW(NUM)
                          IF (TOPSER .EQ. NUM) TOPSER = BELOW(NUM)
                          IF (TOPWYE .EQ. NUM) TOPWYE = BELOW(NUM)
                          IF (TOPVERT .EQ. NUM) TOPVERT= BELOW(NUM)
                      end


                    Method2
```

```
        begin
              IF (FLAG .EQ. 1) THEN
                    FACE = NUM
                   IF (DFACE(FACE) .EQ. 1) THEN
                      TOPLOOP = FACE
                      GOTO 1000
                    ENDIF
               IF(DFACE(FACE) .EQ. 2) THEN
                      TOPPAR = FACE
                      GOTO 1000
                    ENDIF
                    ENDIF
                    IF (NSUM .EQ. (3*NCHECK+2)) THEN
                       TOPDEL = FACE
                    ENDIF
                    GOTO 1000
                 ENDIF
                      TOPVERT = NODE
                      GOTO 1000
                    IF(DNODE(NODE) .EQ. 2) THEN
                      TOPSER = NODE
                      GOTO 1000
                    ENDIF
                    IF (NSUM .EQ. (3*NCHECK+2)) THEN
                       TOPWYE = NODE
                    ENDIF
            1000      RETURN
        end

SubDerivation {remove, positive}


Object19 is
      Attributes
            stprob, upprob, loprob  :  real

      Methods

            Method1
              begin
                IF (SINK1 .EQ. NODE) THEN
                  DO  10  J= 1,10
                    UPPROB(J) = POS(EDGE,J)* UPPROB(J)
              LOPROB(J) = NEG(EDGE,J)* LOPROB(J)
              STPROB(J) = POINT(EDGE,J) * STPROB(J)
        10       CONTINUE
              ENDIF
          end


        Method2
          begin
             DO 110 J = 1,10
               LOPROB(J) = 1.0
               UPPROB(J) = 1.0
               STPROB(J) = 1.0
        110    CONTINUE
          end


        Method3
          begin
                DO 110 J = 1,10
                   LOPROB(J) = LOPROB(J)*POS(ARC,J)
                   UPPROB(J) = UPPROB(J)*NEG(ARC,J)
                   STPROB(J) = STPROB(J)*POINT(ARC,J)
                   WRITE(8,174) OPERPROB, LOPROB(J), STPROB(J),
                                  UPPROB(J)
              174   FORMAT
              110   CONTINUE
          end

    SubDerivation {vertex, main, reduce}


Object20 is
    Attributes
        term1, term2, term3, term4  :  integer

    Methods

        Method1
          begin
               READ(4,*) NTERM, TERM1, TERM2,TERM3,TERM4
          end
```

```
            Method2
               begin
                    NODE = NUM
                     IF(NODE .EQ. SORE)  GOTO 1000
                     IF(NODE .EQ. TERM1)  GOTO 1000
                     IF(NODE .EQ. TERM2)  GOTO 1000
                     IF(NODE .EQ. TERM3)  GOTO 1000
                     IF(NODE .EQ. TERM4 )  GOTO 1000
            1000   RETURN
               end

     SubDerivation { main, positive}

Object21 is
     Attributes
            above, below  : integer

     Methods

            Method1
              begin
                     IF(DFACE(FACE) .EQ. 1) THEN
                        IF(TOPLOOP .GT. 0) THEN
                            ABOVE(TOPLOOP)=FACE
                            BELOW(FACE) =TOPLOOP
                        ENDIF
                   ENDIF
                    IF(DFACE(FACE) .EQ. 2) THEN
                        IF(TOPPAR .GT. 0) THEN
                            ABOVE(TOPPAR)=FACE
                            BELOW(FACE) =TOPPAR
                        ENDIF
                   ENDIF
                     IF (NSUM .EQ. (3*NCHECK+2)) THEN
                          IF (TOPDEL .GT. 0) THEN
                            ABOVE(TOPDEL)=FACE
                            BELOW(FACE) =TOPDEL
                          ENDIF
                   ENDIF
                     NODE = NUM
                     IF(DNODE(NODE) .EQ. 1) THEN
                        IF(TOPVERT .GT. 0) THEN
                            ABOVE(TOPVERT)=NODE
                    BELOW(NODE) =TOPVERT
                 ENDIF
              ENDIF
               IF(DNODE(NODE) .EQ. 2) THEN
                 IF(TOPSER .GT. 0) THEN
                    ABOVE(TOPSER)=NODE
                    BELOW(NODE) =TOPSER
                 ENDIF
              ENDIF
               IF (NSUM .EQ. (3*NCHECK+2)) THEN
                  IF (TOPWYE .GT. 0) THEN
                    ABOVE(TOPWYE)=NODE
                    BELOW(NODE) =TOPWYE
                  ENDIF
              ENDIF
          end

      SubDerivation {positive}

Object22 is
      Attributes
            sink2  :  integer
      Methods

      SubDerivation  {}



Object23 is
      Attributes
            oface  :  integer
      Methods

      SubDerivation  {}



Object24 is
      Attributes
            facnum  :  integer

      Methods
```

```
                Method1
                   begin
                        DO 17 K= 1,NF
                                READ(4,*) I, DFACE(I)
                                FACNUM(K) = I
                       17    CONTINUE
                   end

        SubDerivation {main}

Object25 is
        Attributes
                nodnum  :  integer

        Methods

                Method1
                   begin
                       DO 16 J=1,NN
                         READ(4,*) I, DNODE(I)
                         NODNUM(J) = I
                16    CONTINUE
                   end

        SubDerivation { main}

Object26 is
        Attributes
                nterm  :  integer

        Methods

                Method1
                   begin
                        READ(4,*) NTERM, TERM1, TERM2,TERM3,TERM4
                   end

        SubDerivation {main}

Object27 is
        Attributes
                sore  : integer
        Methods

              Method1
                begin
                  READ(4,*) SORE,SINK1, START
                   PRINT *,'SOURCE:',SORE,'  SINK1:',SINK1
                end

        SubDerivation {main}

  Object28 is
        Attributes
              sink1  :  integer

        Methods

              Method1 ·
                begin
                     IF BNODE(EDGE) .EQ. NODE) THEN
                           NODE1 = FNODE(EDGE)
                     ELSE
                           NODE1 = BNODE(EDGE)
                     ENDIF
                      IF (SINK1 .EQ. NODE) THEN
                        SINK1 = NODE1
                        WRITE(*,11) SINK1
                     11    FORMAT('THE NEW SINK1 NODE IS',I4)
                          ENDIF
                end

              Method2
                begin
                     READ(4,*) SORE,SINK1, START
                      PRINT *,'SOURCE:',SORE,'  SINK1:',SINK1
                end
```

```
Method3
   begin
        NODE = NUM
         IF (NODE..EQ. SINK1) GOTO 1000
        1000 RETURN

   end
```

SubDerivation {vertex, main, positive}

Vita

Bonnie L. Achee received the bachelor of science degree in Computer Science from Southeastern Louisiana University in 1990, where she graduated magna cum laude. She was a Board of Regents Fellow at Louisiana State University during her graduate studies. She received the degree of Doctor of Philosophy in Computer Science in May 1998. She is a member of the Association for Computing Machinery and the IEEE Computer Society. She is currently employed as an Assistant Professor in the Department of Computer Science at Southeastern Louisiana University.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Bonnie Lynn Achee

Major Field: Computer Science

Title of Dissertation: FORM: The FORTRAN Object Recovery Model - A Methodology to Extract Object-Oriented Designs From Imperative Code

Approved:

[illegible]
Major Professor and Chairman

[illegible]
Dean of the Graduate School

EXAMINING COMMITTEE:

James G. Oxley

[illegible]

[illegible]

[illegible]

[illegible]

Date of Examination:

December 9, 1997

IMAGE EVALUATION TEST TARGET (QA-3)