AUTOMATED CONVERSION OF STRUCTURED FORTRAN 77 CODE INTO OBJECT-ORIENTED C++ CODE

by

Malini Kothapalli

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

Master of Science

in

Computer Science

APPROVED:

Dr. Jan Helge Bøhn, Chairman

Dr. James D. Arthur, Co-Chairman

Dr. Stephen H. Edwards

May 8, 2003
Blacksburg, Virginia

Keywords: legacy code, automated conversion, structured, object-oriented

AUTOMATED CONVERSION OF STRUCTURED FORTRAN77 CODE INTO OBJECT-ORIENTED C++ CODE

by

Malini Kothapalli
Department of Computer Science

Jan Helge Bøhn, Chairman
Department of Mechanical Engineering

James D. Arthur, Co-Chairman
Department of Computer Science

ABSTRACT

The maintenance of legacy software systems that were developed using a procedural design approach is becoming increasingly expensive. The procedural approach is often ill suited for complex systems that need to integrate with other codes. Furthermore, these legacy systems are usually written in FORTRAN, for which there is increasingly less personnel available compared to, say, C++. While it would be desirable to convert these legacy systems into object-oriented codes described in C++, such a conversion process is nontrivial. Currently, the structural design must be manually examined, interpreted, and converted into an object-oriented design described in an object-oriented language. Therefore, the conversion process is likely to introduce numerous new inconsistencies and errors, which degrades the software's quality and increases its costs.

The preferred solution would be to automate this conversion process. Automation would promote consistency by eliminating the manual variations in interpretation and implementation. It would therefore maximize the likelihood that the converted code does not introduce new errors relative to the original code.

The work presented here automates the conversion process from procedural design described in the FORTRAN77 language into object-oriented design described in the C++ language. It demonstrates the extraction of object-oriented elements using FORTRAN common block structures and FORTRAN subroutine and function-calling hierarchies. The result is a consistent, first-cut converted design, which enhances

cohesion within classes and reduces coupling between classes. This result is described in the contemporary, broadly used computer language C++, which integrates with adjacent modules that might still remain procedural and described in FORTRAN.

ACKNOWLEDGEMENTS

I would like to thank my advisor, Dr. Jan Helge Bøhn, for his guidance and direction during my research and the writing of this thesis. I would also like to thank my co-advisor, Dr. James D. Arthur, for his advice and encouragement, and committee member Dr. Stephen Edwards for his intellectual feedback about the work.

I would also like to thank AVID LLC, Blacksburg, VA, for funding and providing case studies for this project. I would like to thank Ms. Shalini Akella for being a part of this research work from the beginning till the end of December 2002.

During the initial stages of the research, Ms. Cindy Zhang helped me to understand the details of the ACSYNT software system that is used during this research. I would like to thank her for the support.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1.
INTRODUCTION

The main purpose of this chapter is to introduce the topic of converting legacy code into an object-oriented code. Specifically, it deals with the automated conversion of FORTRAN77 code into C++ code. The motivation for the research and the advantages of the work done are discussed. Also presented is an approach towards solving the problem.

Large software systems, such as those used in aircraft design, including the ACSYNT conceptual aircraft design system [1] that is used for case studies in this thesis, often contain massive amounts of legacy code written in FORTRAN77. The predominant design strategy in these legacy codes is procedural. Maintaining these software systems is not an easy task and does not integrate well with modern object-oriented design practices and additions. This problem is compounded by the fact that these systems typically have undergone so many changes over the years that they are now often in an extremely brittle condition. Also, the availability of personnel and compilers for the maintenance of these legacy systems and languages are increasingly in short supply. Object-oriented design methodologies and programming languages address these problems, in particular, through the efficient reuse of modules. It is, therefore, desirable to move these legacy systems to be represented by an object-oriented design and described in a current object-oriented language, such as C++.

There might be special cases where converting a legacy code into an object-oriented code may not be practical. This can be determined using the Paradigm Selection Process (PSP) suggested by Arthur *et al.* [2]. The result of applying this process to the target system could be a recommendation to either continue the currently existing procedural paradigm or to convert it into an object-oriented paradigm.

Once the decision is made to shift the paradigm, the target system has to be reverse engineered. *Reverse engineering* is the process of analyzing a subject system with two goals in mind: (1) to identify the system's components and their interrelationships, and (2) to create representations of the system in another form or at a higher level of abstraction. *Reengineering* is the examination of a subject system in order to reconstitute it in a new form and subsequently implement the new form. Reengineering a legacy system implies redesigning the system from scratch to meet all the specifications that hold for the legacy system. In order to reengineer the legacy system, both the specification and the documentation of the legacy system are required. Obtaining these sources of information is a challenge. This vital information is often only implicitly available via the source code. Extracting this implicit information by manual reverse engineering is nontrivial when the legacy software system ages, evolves, and grows larger. The manual aspect of such a process makes it difficult to keep the extracted information consistent and orderly, such that intelligent software maintenance decisions can be made.

One approach to reverse engineering for the purpose of increased maintainability is to convert the legacy code into a more maintainable object-oriented C++ code. Today, such conversion processes are largely manual efforts [3]. A major disadvantage to this approach is that new errors are likely to be introduced because of human variability. This makes subsequent maintenance difficult and, hence, costly. The ongoing cost with respect to the manpower and time required for manual conversion is huge and must be applied repeatedly for each and every module or system that has to be converted. In contrast, an automated conversion tool should experience significantly lower ongoing conversion costs in exchange for the one-time upfront cost of tool development. Once the tool is developed, bulk conversion can be achieved with expedience, consistency, and minimal variance. Manual conversion efforts can then be concentrated on specialty issues that are not adequately addressed by the automated tool. This is akin to how a spell- and grammar checker removes the mundane elements of writing and helps the author concentrate on the content of the story, which, still, only a skilled human can do well. The result is a dramatically improved code with respect to maintainability. Therefore, we are motivated to make the following problem statement.

1.1. Problem Statement

This thesis is concerned both with automating the identification and extraction of potential object-oriented classes from structured FORTRAN77 source code, and with creating and describing these classes in the C++ language. In particular, this thesis proposes and demonstrates a solution for fully automated source code conversion where the object-oriented classes are identified and extracted based on the common blocks and the subroutine- and function-calling hierarchy that are present in the FORTRAN77 source code being converted.

1.2. Solution Outline

The software tool *F77toCpp* has been developed to automate the conversion of structured FORTRAN77 into object-oriented C++. It consists of a number of programs, including a parser that is built using the open source software tool *Flex* [4]. The tool identifies object-oriented classes based on the FORTRAN77 common blocks and the subroutine- and function-calling hierarchy. This involves the following tasks:

1. *Identify the potential classes:* The algorithm presented later in this thesis identifies the classes on the basis of the subroutine- and function-calling hierarchy. Only a subset of the subroutines and the functions in this calling hierarchy become classes, subject to a user-selected parameter.
2. *Identify the various member functions and attributes:* Once the classes are identified from the calling hierarchy, the remaining subroutines and functions that are not classes become member functions of their parent classes; that is, the class which is nearest within the hierarchy. If a member function accesses a common block, then an instance of this common block is made an attribute of the member function's class.
3. *Identify the hierarchical relationships between various classes:* When a member function in one class communicates with a member in another class, there exists a relationship between those two classes. When this happens, the class of the initiating member function is given an instance of the second class as an attribute to facilitate this inter-class communication.

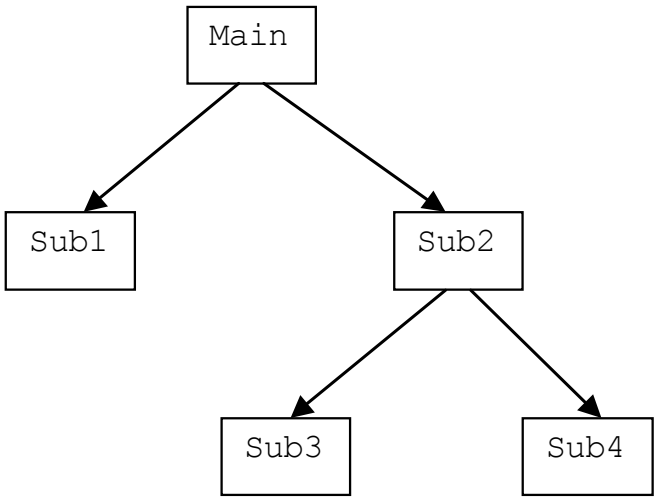


Figure 1-1 Example of a subroutine-calling hierarchy

In addition to turning the FORTRAN77 common blocks into classes, *F77toCpp* creates a tree structure representing the calling hierarchy of the subroutines and functions present in the FORTRAN77 code. This tree structure is then used to combine well-balanced groups of subroutines and functions into classes. For instance, in Figure 1-1, which represents a grossly simplified example, one might choose to group the subroutines `Sub2`, `Sub3`, and `Sub4` into a single class, wherein the three subroutines become members of this class. The logic and details of this grouping process will be discussed in detail in Chapter 3. The quality of the classes identified by *F77toCpp* can be tuned on the basis of the cohesion and coupling metrics. The user has the capability to choose the algorithm parameter in such a way that the resulting classes have the desired cohesion and coupling. The cohesion and coupling metrics that are appropriate to the algorithm presented in this thesis are discussed later in Chapter 3.

After the classes have been identified, *F77toCpp* generates the corresponding *.h* and *.cpp* files for each class. The respective implementation for each of the member functions in these classes is generated by first extracting it from the FORTRAN77 code using the open source program *fsplit* by Burkard [5], and then syntactically converting it to C++ using the open source program *f2c* by Feldman *et al*. [6,7].

The program *f2c* converts FORTRAN77 common blocks into C/C++ `struct` data structures. *F77toCpp* must therefore convert these into `class` data structures in

order to better facilitate an object-oriented design. At the same time, it is important that the FORTRAN77 modules that have not yet been converted to C++ can still access these same data structures without being informed that the conversion has taken place. This is achieved by implementing these classes using a Singleton design pattern [8] that internally references the original `struct` data structure. These changes to the C++ code generated by *f2c* are described in detail in Chapter 3.

The result is *F77toCpp*, a software tool that provides automated conversion of structured FORTRAN77 source code into object-oriented C++ source code. The C++ code generated maintains its data structure access compatibility with those FORTRAN77 source code modules that are not yet converted. The correctness of this conversion process has been tested and validated with the automated conversion of three FORTRAN77 modules within the ACSYNT conceptual aircraft design system [1]: The smallest of these three modules consisted of about 1,500 lines of FORTRAN77 code, while the largest consisted of about 16,500 lines of FORTRAN77 code.

1.3. Thesis Organization

This thesis consists of six chapters, including this introductory chapter.

Chapter 2 provides a literature review that focuses on the problem of maintaining legacy code, and, in particular the problem of migrating FORTRAN77 code to an object-oriented paradigm. This includes the review of language syntax conversion, object-oriented class identification, conversion automation, and program restructuring.

Chapter 3 details the algorithms for identifying and creating classes using the FORTRAN77 subroutine- and function-calling hierarchy. It also details how FORTRAN77 common blocks are converted into classes while maintaining compatibility with remaining unconverted FORTRAN77 modules.

Chapter 4 presents the design and implementation details of the automated conversion tool, *F77toCpp*.

Chapter 5 describes the three case studies within the ACSYNT conceptual aircraft design system [1] with which *F77toCpp* has been tested and validated.

Finally, Chapter 6 reviews the major conclusions of this thesis' research, highlights its contributions, and offers suggestions for further study.

CHAPTER 2.
LITERATURE REVIEW

Legacy codes written in FORTRAN or COBOL are still widely used within the financial, defense, and aircraft design industries, among many others. Many of these codes have become increasingly obsolete, but because their core functionalities remain essential, these codes need to be maintained and moved forward. One option is to maintain these obsolete codes in their current form and modify them minimally as needed to meet requirements. A more attractive option is to convert these codes into a more current language and design, in particular an object-oriented language and design that enjoys more readily available support and that is more suitable for use with modern software technologies. Once converted, the cost of maintaining this code would decrease, as would the cost of integrating it with future systems.

This chapter reviews the literature with respect to converting legacy codes into an object-oriented design, and in particular the conversion of legacy FORTRAN77 code into an object-oriented design described in the C++ programming language. Therefore, the following sections review the literature with respect to syntactical conversion from one programming language into another; the identification of potential object-oriented classes within programs described in a procedural language; methods for converting manually identified classes into object-oriented code; and, finally, code restructuring based on its hierarchical calling structure.

2.1. Syntactical Conversion

A number of conversion tools have been developed to syntactically convert FORTRAN77 code into C or C++ code. Feldman *et al.* developed the *Fortran-to-C*

Converter [6,7], also known as *f2c*, based on the FORTRAN77 compiler by Feldman and Weinberger [9,10]. Its purpose is to enable the compiling of FORTRAN77 code, unaltered, on computers where FORTRAN77 compilers are not available, but where C or C++ compilers are available. It automatically converts FORTRAN77 code syntactically into C code, which in turn can be treated as C++ code by enclosing it with the appropriate compiler statements. Hence, the converted code retains the procedural paradigm that was present in the FORTRAN77 code; it does not represent an object-oriented paradigm.

The open source FORTRAN77 compiler, *g77*, by the GNU Project [11] does FORTRAN77-to-C syntactical conversion internally before it is fed into the general-purpose GNU Compiler Collection (GCC) system. It does not generate C or C++ code as output for manual inspection or use elsewhere.

Laffra [12] developed *C2J++* to convert C and C++ source to Java source. *C2J++* transliterates C source into a syntactically mostly correct Java program. It fails to recognize and convert comma expressions and `goto` statements. Data type conversions are done in a trivial way, usually by removing address and dereference operators. While *C2J++* can assist a developer in the integration of C code into Java programs, extensive manual efforts are necessary to transliterate large volumes of source code. The different assignment and parameter passing semantics of C and Java are ignored, with the result that the transliterated program has a very different behavior than the original.

Novosoft's *C2J* [13] is similar to *C2J++* in that it transliterates C source to Java, but it does not handle C++. It solves many of the problems of *C2J++*. It also transliterates control flow features of C that are not supported by Java, such as comma expressions and goto statements. *C2J* shares some of the disadvantages of the Java backend for GCC: Data structures are stored in a large array, thus circumventing Java's type checking and runtime security checks. The array access requires many operations and the necessary extra type conversions create a runtime overhead. They also make the code difficult to read. In this sense, *C2J* provides only a slight advantage over the Java Backend for GCC: Debugging might be easier with the transliterated Java source code available, but on the other hand, some C control flow structures may be more efficiently implemented by compiling the C source code directly to machine language for the Java Virtual Machine (JVM).

Salopek *et al.* [14] present a brief overview of Recycler™ by BAE SYSTEMS Mission Solutions, Inc., a commercial product that facilitates automated software language translation and re-hosting of existing test programs onto new test systems. Its purpose is to provide a generic syntactical conversion between languages. In essence, it is a lexical analyzer with a graphical user interface (GUI) where the user first builds a model of the two languages and a translation model between these two languages. The language conversion can then be executed automatically, or step by step, with feedback to the user in the form of various metrics and translation output. It is expected that the user will repeat this conversion process while enhancing the three models until a satisfactory conversion result is achieved. The system does not address the conversion from a procedural to an object-oriented paradigm.

2.2. Identification of Classes in Procedural Languages

Though the above-mentioned tools concentrate on syntactic conversion, there has been some efforts towards the paradigm shift from procedural to object-oriented design. Many have investigated methods for identifying classes from procedural languages like COBOL and C [15,16,17]. For instance, the manual approach proposed by Cimitile *et al.* [16], identifies persistent data stores as objects, while programs and subroutines are candidates for object methods. To identify the data structure of objects, a static analysis of the system together with an analysis of the available documentation is performed. These results are refined by using available documentation to assign a concept of the application domain to each of the identified objects. Once the data stores defining the state of the candidate objects have been identified, the identification of the object services entails the assignment of programs and groups of subroutines to persistent data stores. This is achieved in two steps: First programs are assigned to data stores; then they are analyzed to evaluate if they can be decomposed into simpler operations.

Liu and Wilde [18] propose two important factors for characterizing objects: global data, and the types of formal parameters and return values. Their first approach is based on global and persistent data and establishes links to the routines that manipulate

such data. Their second approach is based on data types and establishes relationships between such types and the routines that use them for formal parameters or return values.

In a COBOL system, many fields within the records, which are the starting point for many object identification approaches [15,19], are often unrelated. The above-mentioned object identification approaches do not solve the record-fields problems that often arise in COBOL legacy systems. Duersen and Kuipers [20] suggested a method to identify objects by clustering record fields into coherent groups based on the actual usage of these fields in procedural code; thus solving the records-field's problem. The solution is based on cluster and concept analysis. Concept analysis and cluster analysis both start with a table indicating the features of a given set of items. Cluster analysis then partitions the set of items in a series of disjoint clusters by means of a numeric distance measure between items indicating how many features they share. Concept analysis differs in two respects. First, it does not group items, but, rather, it builds up so-called *concepts*, which are maximal sets of items sharing certain features. Second, it does not try to find a single optimal grouping based on numeric distances. Instead it constructs all possible concepts via a concise lattice representation.

Sahraoui *et al.* [21] proposed a quality-based approach for automating the migration of legacy code towards an object-oriented paradigm. They present two techniques which consider object identification as a grouping problem: genetic algorithms, and conceptual clustering. This approach is different from previously mentioned work in that it explicitly takes into account the quality of the targeted object-oriented system. In particular, the solutions are evaluated on the basis of two important properties of software design: coupling, and cohesion.

The above-mentioned methodologies can be applied to extract some objects from FORTRAN code, but they do not cover all the cases that might arise in FORTRAN. For this we need methodologies that are specific to FORTRAN code. Hence, efforts have been made to identify objects specifically from FORTRAN legacy code [3,22]. For example, Subramaniam and Byrne [3] proposed a nine-step process for deriving an object model from an existing unstructured FORTRAN code. The reverse engineering is done using the application domain information and the existing documentation and code. The resulting object model contains sufficient detail to guide the reimplementation of a

system into an object-oriented language such as C++. However, they only consider reverse engineering and not reimplementation. To produce a more suitable object model than that produced by a single strategy, Subramaniam and Bryne presents a reengineering strategy which is a combination of several strategies. This includes the use of both top-down and bottom-up strategies to help detect meaningful objects and classes.

Achee and Carver [22] describe an algorithm that uses a greedy approach to extract objects from FORTRAN77 code using the concepts of graph theory. First, the data is grouped together to form attribute sets; then methods are affixed to the sets of attributes to determine the objects. The algorithm seeks to obtain the smallest set of parameters needed to obtain the strongest cohesive unit. The cohesive strength of a pair of parameters is measured by determining the frequency in which they are both necessary to perform various functions, where the unit of functionality is the subroutine. Each statement in the source code is evaluated to identify the methods. The identified methods are attached to those attribute sets whose state is changed by the method in consideration.

The processes suggested in [3,22] are both manual and therefore time consuming, expensive, and error prone. For this reason, automated assistance is necessary for these conversion processes to become practical. Subramaniam and Byrne [3] recognize this by concluding that an algorithm for bottom-up object identification and for identifying potential methods is required.

2.3. Semi-automated Conversion

Fanta and Rajlich [23] proposed the automation of restructuring legacy C code into C++. The restructuring is accomplished by restructuring tools and scenarios. They presented a transformation tool-set and the design of the individual tools, but with the legacy source code comprehension process being manual and without any automated support. Once the classes are identified, the tool sets are used for encapsulation. For example, the function encapsulation tool is used to encapsulate a sequence of consecutive statements or expressions into a new function and replaces the original code fragment with an appropriate function call. Likewise, the function insertion tool inserts a standalone function into a target class. Though the tools are efficient and produce good results, the

initial step of identifying classes remains manual. Hence their process of restructuring legacy C code into C++ is partly manual and partly automated.

Penteado *et al*. [24] suggested a reengineering approach that first applies Fusion/RE [25] to a legacy system, and then applies segmentation that preserves the functionality and programming language, while changing the paradigm from procedural to object-oriented. Next, using this segmented procedural code, the object-oriented code is automatically generated using the Draco-Put transformation system [26]. The initial phase of applying Fusion/RE to analyze legacy code and extracting objects from it is a manual step. The same is true for the segmentation process, which uses a text editor. The Draco-Put system is labor-intensive to set up for transforming from one programming language to another. Hence, only one of three steps in this procedure is automated.

2.4. Restructuring based on hierarchical structure

The work presented in this following section differs from the work presented above in the sense that the subroutine- and function-calling hierarchy is used for identifying classes. The result of clustering subroutines and functions based on their inter-communication will tend to be a highly cohesive set of classes with minimized coupling between them. However, thus far, this hierarchical structure has not been used as the basis for converting procedural code into object-oriented code.

Choi and Scacchi [27] suggest an approach to provide a simplified view of a system by identifying clusters of routines or procedures that belong together. The objective of clustering is normally to identify subsystems that will be meaningful to the programmer rather than to specifically understand the system's data organization and structure. In their case, their restructuring algorithm first constructs a hierarchical structure from an implementation description based on the relationship between modules in terms of the resources they exchange. This hierarchical structure is derived automatically by analyzing the functions, procedure calls, and data items that are given as parameters to a module. Here, a module refers to each source-code file in the system. Once the hierarchical structure is identified, it is mapped into a resource-structure

diagram, which shows the control relationship between the modules and subsystems. This diagram can then be restructured based on minimizing the module coupling and the alteration distance.

Module coupling is defined as the number of modules under a subsystem [27]. Coupling is minimized by reducing the fan-out factor of a subsystem by requiring the number of modules attached to a subsystem be kept low. Alteration distance between modules is a measure of the distance between an altered module and the affected module. The alteration distance is zero if both the altered and affected module is part of one subsystem, which is desirable.

Figure 2-1 shows a Resource Flow Diagram, in which a line indicates a resource exchange between modules. Figure 2-2 shows three resource-structure diagrams: Figure

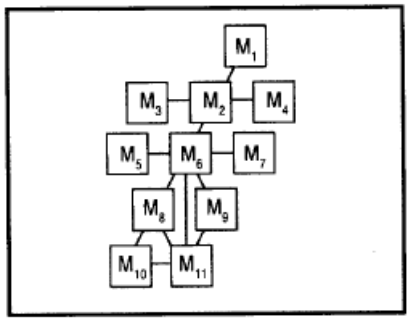


Figure 2-1 A resource-flow diagram of modules (from [27])

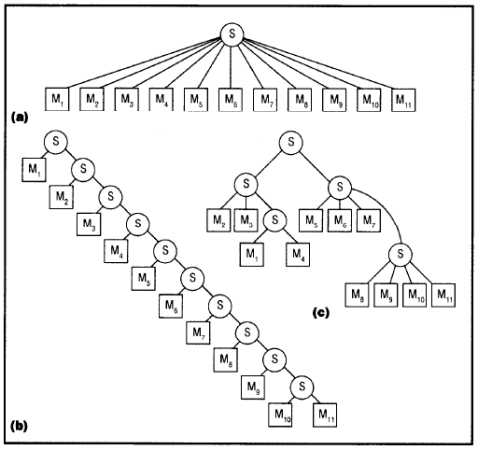


Figure 2-2 A resource-structure diagram based on (a) maximum control; (b) minimum coupling; and (c) using restructuring algorithm (from [27])

2-2a illustrates restructuring based on maximum control visibility (minimum alteration distance); Figure 2-2b illustrates minimum coupling; and Figure 2-2c illustrates the resource-structure diagram after the restructuring algorithm was applied [27]. This approach to extract and restructure a system design can be applied to large systems that previously were too difficult to understand and modify.

2.5. Observations

With regard to reverse engineering FORTRAN77 legacy code into object-oriented C++ code, the literature suggests the following:

(a) There are tools that convert from one language to another, but they perform only syntax conversion.

(b) There are methodologies proposed for identifying objects from general legacy code. However, these solutions do not handle all the different cases that are observed in FORTRAN77 code.

(c) There are methodologies proposed to identify objects from FORTRAN code, but these methodologies are mostly manual.

(d) There are methodologies proposed that are semi-automated and that can convert procedural codes into object-oriented codes, but they require at least 50% manual effort.

(e) Finally, there have been restructuring efforts based on identifying the hierarchical structure within a code. Their primary purpose for identifying this hierarchical structure is to restructure the code to provide a better understanding of large systems that were previously too complex to comprehend.

The approach proposed in this thesis to convert structured FORTRAN77 code into object-oriented C++ code is similar to the restructuring approach taken by Choi and Scacchi [27]. However, instead of restructuring the system by identifying a hierarchy, we use the hierarchy to cluster subroutines and functions into classes, thus capturing the paradigm shift. The advantages of our approach are two fold: Firstly, the functional cohesion and coupling metrics can be used to choose the appropriate clustering of subroutines [28]. Secondly, we have practically eliminated manual

intervention from the process of converting structural FORTRAN77 code into object-oriented C++ code.

CHAPTER 3.
ALGORITHM TO IDENTIFY CLASSES

This chapter discusses the algorithm to identify classes in the structured FORTRAN77 code. The algorithm is developed with automated conversion of the FORTRAN77 code into C++ code as the main objective. It practically eliminates manual intervention from the process of converting structural FORTRAN77 code into object-oriented C++ code. The algorithm identifies potential classes by generating a subroutine- and function-calling hierarchy. Following the identification of classes, the system generates the corresponding *.cpp* and *.h* files. The respective implementation for each of the functions in these classes is extracted from the FORTRAN77 source code and syntactically converted to C++ and inserted into these *.cpp* files.

The process of identifying classes in the FORTRAN77 legacy code can be divided into three different phases: (1) the identification of the subroutine- and function-calling hierarchy, (2) the generation of classes, and (3) the processing of common blocks. The following sections discuss these different phases and present the issues that were identified during the process of identifying classes in the FORTRAN77 code. They also present examples of the subroutine- and function-calling hierarchy organized into a tree structure, and the generation of classes and member functions of these classes from this tree in accordance with the algorithm. Note that in the discussion that follows, FORTRAN77 subroutines and FORTRAN77 functions will not be differentiated, but will both be referred to as subroutines.

3.1. Identifying the Subroutine-Calling Hierarchy

Identifying the subroutine-calling hierarchy is the first phase in the process of identifying classes in the structured FORTRAN77 code. The subroutine-calling hierarchy represents the inter-relationship between subroutines and functions that call each other. In other words, this is the "call/called-by" list. This hierarchy is eventually used to identify the classes.

The subroutine-calling hierarchy is extracted from the legacy code by parsing the FORTRAN77 source code for the keywords *SUBROUTINE*, *FUNCTION*, *PROGRAM*, and *CALL*. The *SUBROUTINE*, *FUNCTION*, and *PROGRAM* keywords represent program units in a FORTRAN77 code, and the *CALL* keyword is used to identify the relationship between these program units. Each such unique program unit is stored as a node in the tree data structure that is used to represent the subroutine-calling hierarchy. Every node stores the name of the FORTRAN77 subroutine, the subroutine's arguments, and references to the calling and called subroutines.

This hierarchical tree structure is used to group subroutines into different classes. In order to avoid replication of code within the classes, each subroutine is made a member of only one class. Hence, the subroutines are not duplicated within the tree; that is, any subroutine forms only one node in the tree.

3.2. Generating Classes

Once the subroutine-calling hierarchy is identified, this hierarchy is used to generate classes. Each class contains one or more subroutines as members, with every subroutine belonging to only one class. *F77toCpp* generates these classes from the subroutine-calling hierarchy tree discussed previously using a clustering algorithm. The algorithm clusters subroutines into classes based on a user defined parameter. Different results can be obtained by varying this parameter's value. To choose an appropriate value for the parameter, object-oriented design metrics can be used to measure the quality of the results. The parameter value can be fine tuned to get the desired results. Cohesion and coupling are the two metrics used in this thesis to measure the quality of the results

generated by *F77toCpp*. These metrics are widely used to define the quality of a software design [28].

The clustering algorithm presented here provides a solution to group related nodes in a tree. Here the nodes represent subroutines, and the relationship between them is the corresponding invocation between subroutines. The algorithm uses the subroutine-calling hierarchy to measure the following variables during the process of generating classes:

- Height of the subroutine-calling hierarchy, *Maxheight*; and
- Height of each node within the subroutine-calling hierarchy, n_h.

Based on a node's height when compared to the height of the subroutine-calling hierarchy tree, the node either forms a class or belongs to an already formed class. When a subroutine does not meet the criteria to form a class, it is made a member function of a class which is its ancestor in the hierarchy tree. Once the algorithm identifies the classes and the subroutines that belong to each of those identified classes, it generates class files.

The algorithm determines the height of each node, n_h, within the subroutine-calling hierarchy, where the height of a node is the maximum distance between the node and its leaf nodes, assuming each leaf node is at a height 1. In this context, a node in the calling hierarchy refers to a subroutine or a function. The height of the calling hierarchy tree, *Maxheight*, is the maximum distance between the root node and the leaf nodes. All the nodes at a height, n_h, greater than or equal to the ceiling of maximum height divided by the factor *x,* i.e., $n_h >= ceil(Maxheight/x)$, are considered as classes, and the remainder of the nodes become member functions of their respective parent classes. The parent class can be either an immediate parent node or an ancestor node in the calling hierarchy. The value of factor *x* is set by the user, and the number of subroutines that form each of these classes is determined by this factor.

For instance, in Figure 3-1, the maximum tree height, *Maxheight*, is three. If the user selects the factor *x* to be three, then all the subroutines whose height is greater than or equal to the value of $ceil(Maxheight/x) = ceil(3/3) = 1$, become classes. Hence, in this case, `Main`, `Sub1`, `Sub2`, `Sub3`, and `Sub4` become classes because their height is equal to or greater than 1.

In Figure 3-2, the maximum height of the tree, *Maxheight* is six. If the value of *x* is two, then the value of *ceil*(*Maxheight/x*) is three. Hence, the subroutines `Main`, `Sub2`, `Sub5`, and `Sub6` become classes because their height is greater than or equal to three. The subroutines `Sub1`, `Sub3`, and `Sub4` become members of the class `Main`; the subroutines `Sub8` and `Sub9` become members of the class `Sub6`; and the subroutine `Sub7` becomes a member of the class `Sub5`. The lines grouping subroutines as shown in Figure 3-1 and Figure 3-2 illustrates the clustering of subroutines into a class. There are two important observations that can be made from the diagram. The first observation, from Figure 3.2, is that although the parent of the `Sub9` subroutine (i.e., the subroutine that calls the `Sub9` subroutine) is `Sub8`, its parent class in the hierarchy is `Sub6`. Therefore, the `Sub9` subroutine becomes a member of the `Sub6` class. The second observation is that any two classes communicate through only one function in one of the classes. For example, the `Main` and `Sub2` classes communicate only through the `Sub2` member function of the `Sub2` class (Figure 3.1), and the `Sub5` and `Sub6` classes communicate only through the `Sub6` member function of the `Sub6` class (Figure 3.2). The second observation plays a vital role in measuring the quality of the generated classes. In particular, it is used to measure the coupling between classes across the entire system as discussed in Section 3.2.2.

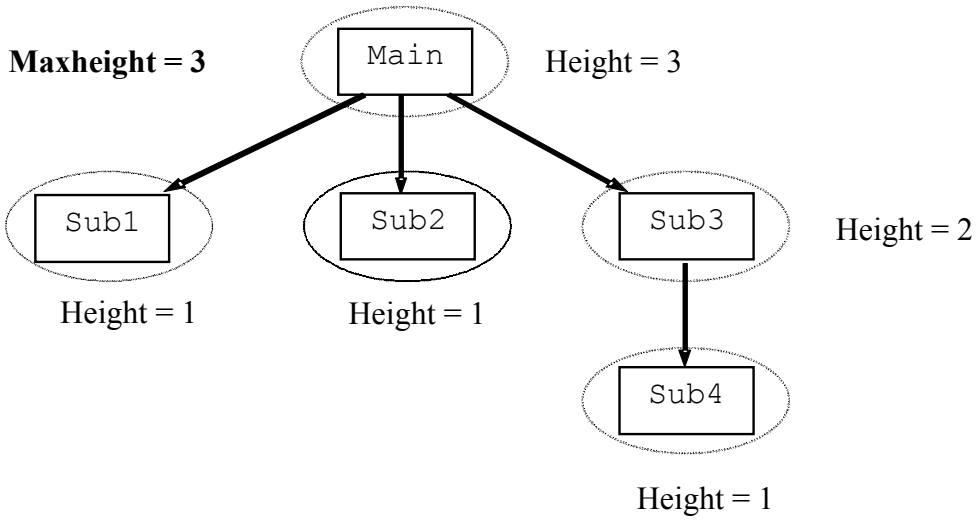


Figure 3-1 First example of a subroutine-calling hierarchy

Both the cases illustrated in this section have subroutines with only one parent. This makes for a simpler scenario where there is only one communication path to any subroutine. When there are multiple communication paths to a subroutine, i.e., when a subroutine has more than one parent, the naive choice is to include the subroutine in all the parent classes. This choice however, results in code replication, and the maintenance of such code is difficult. There are different object-oriented solutions to this problem, and the feasibility of each of these solutions is discussed in the next sub-section.

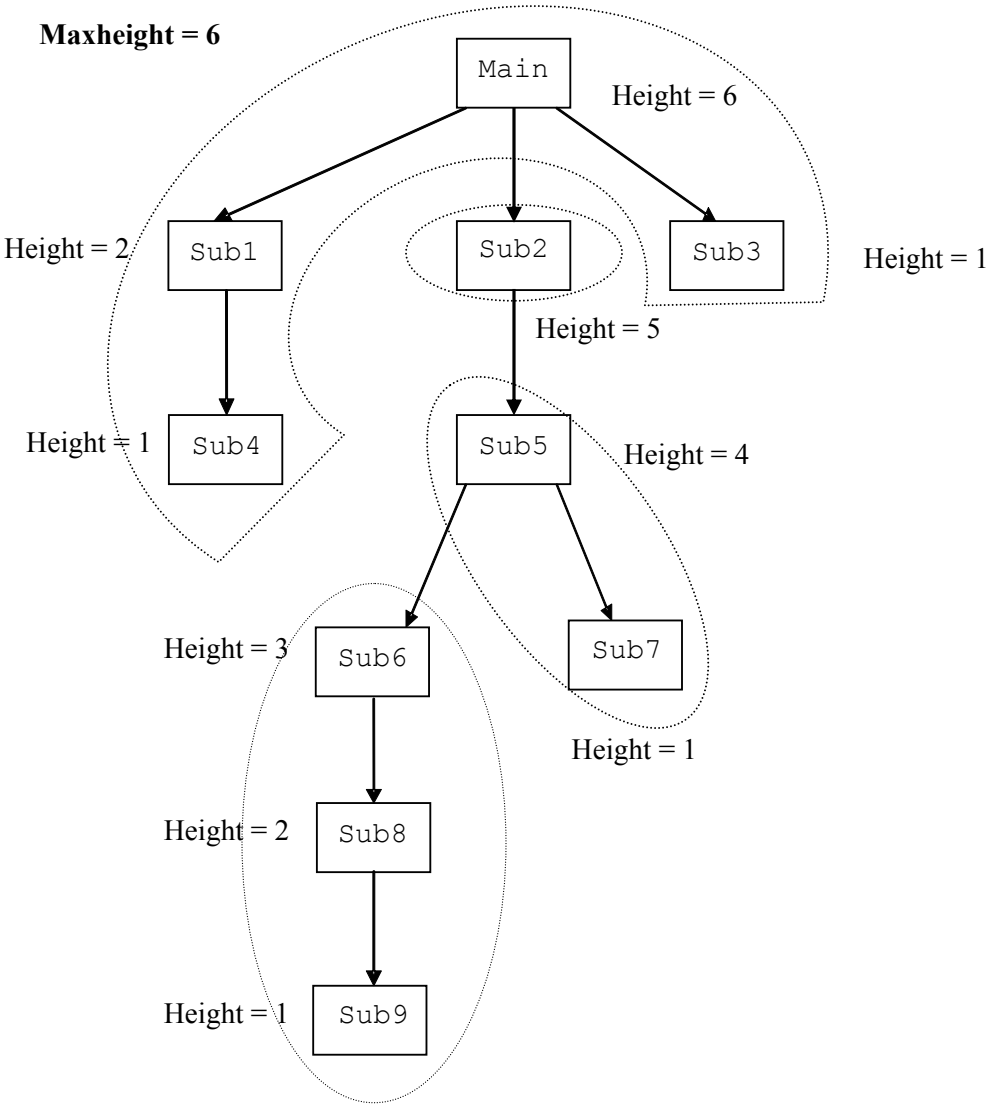


Figure 3-2 Second example of a subroutine-calling hierarchy

3.2.1. Inheritance Versus Composition

This subsection discusses a special case of the aforementioned algorithm. The scenario shown in Figure 3-3 illustrates a common situation in which a node has multiple parents (e.g., a subroutine is called by two or more subroutines). In this case, `Sub4` is called by `Sub2` and `Sub3`. This scenario has to be dealt with differently, because there are multiple subroutines communicating with one subroutine. Since clustering of subroutines is based on this communication, a decision has to be made about which cluster group this subroutine belongs to. This situation can be addressed in one of the three ways:

1. explicitly replicate the code of `Sub4` as two independent member functions: one for `Sub2` and another for `Sub3`;
2. use C++ inheritance, where `Sub4` is a base class that is inherited by the `Sub2` and `Sub3` classes; or
3. use C++ composition, where `Sub4` is a class that is included into the `Sub2` and `Sub3` classes, respectively.

The first method is undesirable because it would cause multiple instances of source code that would have to be maintained perfectly identical. The failure to maintain them identical would likely cause unreliable code. The other two methods, which are

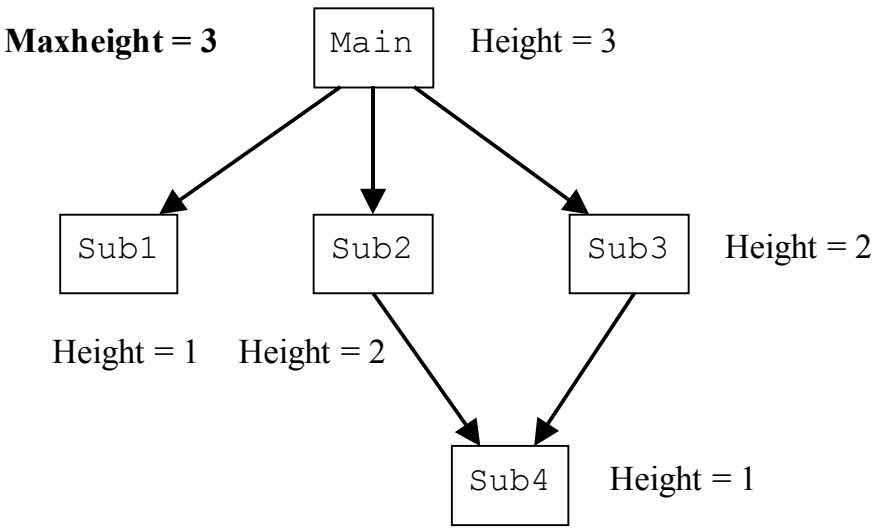


Figure 3-3 Sample subroutine-calling hierarchy: a node with two parents

known as inheritance and composition, respectively, provide for code reuse without code duplication. While inheritance can produce an elegant code, it tends to be more difficult to manage during design evolutions that change the fundamental structure of the design. This difficulty is particularly noticeable for multiple-inheritance. For that reason, it is a good object-oriented design practice to employ inheritance only when there is an overwhelmingly permanent and clear *is-a* relationship between classes (i.e., in all existing and conceivable cases for these classes), and when it will not cause multiple-inheritance. In all other cases, composition is typically preferred.

Being able to automate the reliable and appropriate detection of *is-a* relationship is non-trivial. The automated class extraction methodology that is being developed will therefore initially—and perhaps for quite some time—not have the ability to identify an *is-a* relationship. For that reason, the use of inheritance in the class identification algorithm is always avoided and composition is always applied in its place.

Once the classes are identified, the clustering algorithm generates the respective class files. The resulting classes can be tuned for a better quality with respect to coupling and cohesion by clustering of subroutines into classes. This can be achieved by varying the value of the *x* factor, the parameter on which the clustering algorithm depends.

3.2.2. Effect of Varying the *x* Factor

This subsection discusses the two software attributes that are affected by the class identification algorithm. These two software attributes are coupling and functional cohesion. Coupling is a measure of the strength of association established by a connection from one module (or subroutine) to another. That connection is defined by a reference to some label or address defined elsewhere [29]. Reduction in coupling among modules is recommended, although it cannot be totally eliminated. The degree of coupling between modules depends on a number of factors that can be manipulated by the programmer, such as the complexity of the connection, the type of connection, the size of connection, and the type of communication via the connection [30].

With coupling, one is more concerned with inter-module relatedness or connections. Cohesion is more concerned with intra-modular relatedness [30]. Cohesion is described as the measure of the strength of functional association among elements

within a module [29]. Unlike coupling concepts, with cohesion, each module is considered in isolation. The consideration from now on is how tightly the module's internal elements are bound or related to one another [30].

Functional cohesion is the desired type of cohesion within a module. In a functionally cohesive module, all of the components are related directly to the performance of a single function. In other words, every element is an integral part of, and is essential to, the performance of a single function. Cohesion is a desirable attribute; that is, one should strive to maximize cohesion in the code. This maximizing can be achieved by striving toward functionally cohesive modules [30].

By combining the subroutines that call each other into classes, we generate functionally cohesive classes. The degree of cohesion within a class depends on the factor x in that, as mentioned previously, the variable x is user defined and *ceil*(*Maxheight/x)* determines the number of classes in the system. The integer x can range from 1 to *Maxheight*. At the one extreme, with x set to *Maxheight*, each subroutine is classified into an individual class with maximum coupling between classes. At the other extreme, with x set to 1, all the subroutines are clustered into one single class with maximum cohesion within this class. These are both extremely undesirable solutions. Hence, we need a balanced solution that lies in between these two extremes. Currently, the user manually selects the values for x, and then subjectively evaluates the results.

In order to make an appropriate decision for the clustering of subroutines, the following metrics can be used. The Coupling Between Object Classes (CBO) metric [31] measures the quality of an object-oriented code. CBO is defined as the number of other classes whose methods are used by methods of this class. This particular metric is recommended because it is a measure of non-inheritance based coupling. Since the algorithm is designed in such a way that inheritance is totally avoided, this metric is a good measure. Furthermore, since there is only one communication path between any two classes identified in the calling hierarchy, the coupling value will be the same between any two classes identified. Hence, coupling measurement can be done on a system-wide basis.

The Information-flow based Cohesion (ICH) metric [32] measures cohesion based on the information flow through method invocations within a class. For a method `m`

implemented in class c, the cohesion of m is the number of invocations to other methods implemented in class c, weighted by the number of parameters of the invoked methods. The more parameters an invoked method has, the more information is passed, and the stronger the link is between the invoking and invoked methods. The cohesion of a class is therefore the sum of the cohesion of its methods:

$$ICH^{c}(m) = \sum_{m' \in M_{NEW}(c) \cup M_{OVR}(c)} (1 + |Par(m')|).NPI(m,m'),$$

$$ICH(c) = \sum_{m \in M_{I}(c)} ICH^{c}(m)$$

These two metrics can be used to analyze the quality of the classes identified by *F77toCpp*. For example, in Figure 3.1, assuming that no parameters are passed between methods during method invocation,

Total number of subroutines	=	5
Total number of classes	=	5
Total number of communication paths between classes in the entire system	=	4
CBO value for the hierarchy	=	4
ICH (CMain)	=	0
ICH (CSub1)	=	0
ICH (CSub2)	=	0
ICH (CSub3)	=	0
ICH (CSub4)	=	0

In Figure 3.2, assuming that no parameters are passed between methods during method invocation,

Total number of subroutines	=	10
Total number of classes	=	4
Total number of communication paths between classes in the entire system	=	3
CBO value for the hierarchy	=	3
ICH (CMain)	=	3

ICH (CSub2)	=	0
ICH (CSub5)	=	1
ICH (CSub6)	=	2

High cohesion and low coupling are desired in any object-oriented system. In Figure 3.1, the cohesion is non-existent in all classes, and the coupling is high when compared to the total number of subroutines present in the FORTRAN77 system. This system can be improved in terms of quality of the classes generated, by reducing the coupling in the system and increasing the cohesion within the classes generated. This can be achieved by increasing the value of factor *x*, which will then reduce the number of classes generated. In Figure 3.2, the coupling is not very high when compared to the number of subroutines present in the FORTRAN77 system. This makes it a reasonable system in terms of quality. Consequently, the ability to influence the grouping of classes (by varying the *x* factor) and the ability to measure the result via the CBO and ICH metrics provides the operator of *F77toCpp* program with the deliberate means to strive towards high cohesion and low coupling in the resulting object-oriented system.

3.3. Converting Common Blocks

In FORTRAN77, a common block is used as a placeholder for global variables shared across subroutines. A common block is a storage area that two or more scoping units can share, allowing them to define and reference the same data and to share storage units. Therefore, the common block is a prime candidate for a C++ class.

Each common block present in the FORTRAN77 code is therefore converted by *F77toCpp* into a class. The variables of the common block become public member variables of the class. The data types used in the FORTRAN77 source code are also maintained in these C++ classes. Unless specified in the original source code, the data type of a variable follows the default FORTRAN77 language variable convention. That is, a variable starting with letters *i, j, k, l, m,* or *n* is implicitly defined as type `int` and the rest are defined as of type `float`.

Since common blocks represent global variables, there can be only one instance of classes representing each of these common blocks. The common blocks are therefore converted into classes using the Singleton design pattern [8]. The Singleton design pattern ensures that a class has only one instance, and it provides a global point of access to this instance. It works by having a special method that is used to instantiate the desired object. When this method is called, it checks to see if the object has already been instantiated. If it has been instantiated, the method simply returns a reference to that object. If not, the method instantiates it and returns a reference to the new instance. To ensure that this is the only way to instantiate an object of this type, the constructor of this class is made either protected or private (see Appendix A).

When all the modules of the FORTRAN77 system are converted into object-oriented C++ classes, the interaction between these modules in the system becomes straightforward. They interact using message passing between classes. Since this is a homogeneous system (i.e., all the modules in the system are implemented using only one language), all objects in the system accessing a particular common block refer to the same memory location.

In contrast, consider the case where only some of the modules in the FORTRAN77 system are converted into object-oriented C++. To function properly, the different modules, implemented using different paradigms, have to interact and generate the same results as the all-FORTRAN77 system. Hence, the FORTRAN77 modules sharing common blocks with C++ classes must refer to the same memory addresses. At the outset this will not happen. The FORTRAN77 modules will use one memory location for a common block, while the C++ class implementing the same common block will be allocated a different memory location.

There are however, some convenient guarantees concerning the memory usage of FORTRAN77 common blocks represented by *f2c* and *g77* using the C/C++ data type `struct`, and the C++ data type `class`. Provided that the following conditions hold, these two data types will occupy memory identically, including the order, offset, and sizing of their data members [33]:

- The class has no virtual member functions, including inherited virtual functions of a base class;

- The class has no virtual base classes in the entire inheritance chain;
- The class has no member objects that have either a virtual base class or virtual member functions; and
- All the data members of the class are declared without an intervening access specifier.

C++ treats non-static member functions as static functions. In other words, member functions are ordinary functions. They are no different from global functions, except that they take an implicit `this` argument, which ensures that they are called on an object and that they can access their data members. An invocation of a member function is transformed to a function call, whereby the compiler inserts an additional argument that holds the address of the object.

In the C++ code generated by *F77toCpp*, inheritance is totally avoided. The memory alignment of class objects representing common blocks is therefore same as the memory alignment of the corresponding C structures. The common block objects (C++) can therefore be made to reference the address of the corresponding `C struct`'s used by *f2c* and *g77*. This can be achieved by making changes to the C++ classes representing the common blocks such that they reference the FORTRAN77 common block data structures in place of its own private data members. Hence, the common block data can be shared between yet-unconverted FORTRAN77 modules and the object-oriented C++ classes. Section 5.5 describes this solution in detail.

CHAPTER 4.
THE DESIGN OF *F77TOCPP*

The *F77toCpp* tool has been developed based on the algorithm presented in Chapter 3. This tool is used to convert FORTRAN77 code into C++ code containing object-oriented design elements. *F77toCpp* has a command line interface, and it has been implemented using the standard UNIX utility *Flex*.

F77toCpp initially generates skeletons for the classes identified; that is, classes with member functions, but without function bodies or member variables. The respective implementation for each function in these classes is extracted from the FORTRAN77 code and is syntactically converted into C using *f2c* [6]. *F77toCpp* takes advantage of the fact that C++ supports the C language. Hence, the member-function bodies of the respective classes are filled with the C code that has been converted using *f2c*. Global data is handled by converting common blocks into classes using the Singleton design pattern [34]. The Singleton design pattern ensures that a class has only one instance and a single global point of access (see Appendix A for an example using the Singleton design pattern). Thus, *F77toCpp* generates two types of classes: data-only classes that are used to represent common blocks in C++, and function-only classes that are used to represent groups of program units within the FORTRAN77 system that are being converted.

This chapter presents the design of *F77toCpp*. First it describes the overall code flow of this system. Then it details its design, including its use of directory structures and scripts to complete the conversion process.

4.1. F77toCpp Code Flow

This section presents the sequential steps that *F77toCpp* follows to generate the C++ classes. Note that *F77toCpp* currently accepts only a single FORTRAN77 file. Hence, if the FORTRAN77 system has multiple files, then these need to be manually concatenated into a single file for use as an input to *F77toCpp* as shown here:

```
% cat filea.f fileb.f filec.f > singular.f
% ./F77toCpp singular.f results
```

Figure 4-1 presents an overview of the conversion process used by *F77toCpp* to convert structured FORTRAN77 into object-oriented C++. The following five steps are used to generate these object-oriented C++ classes:

1. Generate skeleton classes: The subroutines are clustered into classes using the subroutine-calling hierarchy. The *.cpp* and *.h* files of these classes are generated with empty member-function bodies.
2. Split FORTRAN files: The program *fsplit* [5] is used to split the program units within the FORTRAN77 source files into individual FORTRAN files.
3. Syntax Conversion: The program *f2c* is used to convert these individual FORTRAN files into C code acceptable to a C++ compiler.
4. Fill the member function body: The C code generated by *f2c* is inserted into the

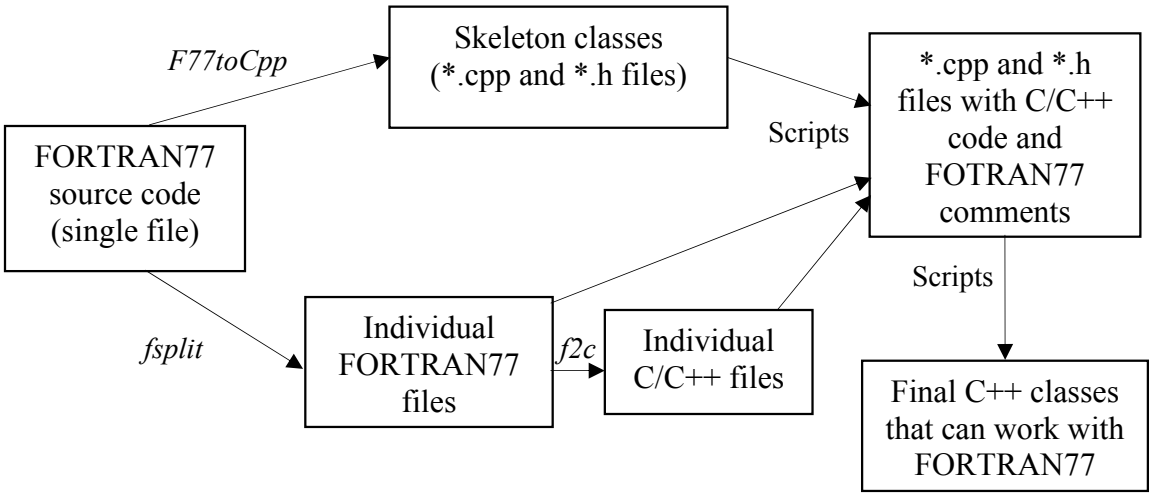


Figure 4-1 Overview of the conversion process used by *F77toCpp*

respective member functions of classes. This C code is encapsulated within the C++ classes, and it needs to be modified to support C++ syntax.

5. Run Scripts: Scripts are used to modify the code in the *.cpp* files. At this point, the classes generated are ready to be used.

As part of generating the skeleton classes, the comments are stripped from the source file and copied to a temporary flat file called *F77toCpp_comments*. The remainder of the source file without the comments is placed in the file *F77toCpp_temp.f*. Each continuous comment block is given a unique comment index that is inserted back into FORTRAN77 code. This produces a FORTRAN77 code that is significantly easier to parse, while at the same time preserving the comments for subsequent insertion into C++ code. To illustrate this comments management, Figure 4-2 shows a block of comments in the file *F77toCpp_comments* that was extracted from *control.f*, a FORTRAN77 source file from within the ACSYNT software system.

The common block and other potential class elements are identified using the file *F77toCpp_temp.f* as input, and the corresponding classes are generated as *.cpp* and *.h* files. During the creation of the *.cpp* and *.h* class files for the function-only classes, keywords with the corresponding member function names are inserted into these files. These keywords are used to identify the member functions, and to insert code from the corresponding FORTRAN source file or C file that is generated by *f2c*, respectively, as explained below. For example, in Figure 4-3, the keyword

```
@#@#@#LINES: 4
COMMENTINDEX: 2
FILE: control.f

C ROUTINE TO CYCLE THROUGH ACSYNT.
ICALC=1: READ ACSYNT CONTROL INPUT AND MODULE INPUT
ICALC=2: CALL MODULES FOR EXECUTION
ICALC=3: CALL MODULES FOR OUTPUT


@#@#@#LINES: 3
COMMENTINDEX: 3
FILE: control.f

------------------------------------------------------------
GO TO (10,20,30), ICALC
-------------------- INPUT -------------------------------
```

Figure 4-2 The *F77toCpp_comments* file

```
#include <iostream.h>
using namespace std;
#include "CAcscyc.h"


//F77CPP_INSERT_FORTRAN  acscyc.f
//F77CPP_INSERT_C        acscyc.c

//F77CPP_INSERT_FORTRAN  dataio.f
//F77CPP_INSERT_C        dataio.c

//F77CPP_INSERT_FORTRAN sumprt.f
//F77CPP_INSERT_C sumprt.c

//F77CPP_INSERT_FORTRAN sumcvt.f
//F77CPP_INSERT_C sumcvt.c
```

Figure 4-3 The keywords show where in this skeleton *.cpp* file the FORTRAN code (comments) and C code (member functions) are to be inserted.

F77CPP_INSERT_FORTRAN is used to show where to place the FORTRAN77 code as a comment in the *.cpp* file, while the keyword F77CPP_INSERT_C is used to show where to insert the corresponding C code that is generated by *f2c*.

The file *F77toCpp_temp.f* is next provided as input to the program *fsplit* [5]. This program identifies the subroutines and functions in a source file, and, for each of these program units, generates an individual FORTRAN77 file with the same name as that of the program unit, but with the *.f* file extension, and places the corresponding original FORTRAN77 source code in that file. Splitting these program units into individual files greatly simplifies subsequent processing. The program *f2c* can now convert each program unit into a separate *.c* file, which then can easily be inserted into the appropriate *.cpp* class file. The same is true for the *.f* file that is inserted into this same *.cpp* file as a comment. These file insertions are performed by a script (see Appendix B for an example). Note also how *f2c* generates C code that is acceptable to a C++ compiler by enclosing the code within an `#extern "C"` compiler directive. Another important reason to split each program unit into individual files is that it avoids the complex union structure for common blocks that *f2c* generates when there are two or more program units in a file accessing the same common block. Hence, the resulting C++ code is simplified.

Though *f2c* handles the syntactic conversion, there are issues that have to be considered when using the *f2c*-converted code within the classes generated by *F77toCpp*.

One such issue is how to handle FORTRAN77 equivalence. There are two equivalence scenarios that might occur within FORTRAN77 codes. Those are when two local variables are equivalenced to each other, and when a local variable is equivalenced to a common block variable. The first scenario is not of concern because it only involves local variables. However, in the second scenario, where a local variable is equivalenced to a common block variable, *f2c* defines the local variable via a compiler directive such that it maps to the address of that particular common block variable within the `struct` data structure that *f2c* and *g77* use to represent that common block in C. Since *F77toCpp* converts this `struct` data structure into a `class` data structure in C++, this compiler directive must be modified accordingly. Appendix C illustrates this modification, which is made automatically via a script. This modification contains two elements. The first is to change the reference from a struct to a class (Appendix D illustrates this change in general). The second is to change from *f2c*'s numeric index into the data structure to address the target variable, to addressing the variable using its name. While the latter is not strictly required, it makes for code that is easier to read and maintain. The following brief example of these changes is taken from the complete example shown in Appendix C:

f2c: `#define irow ((integer *) &skinc_1 + 3)`

F77toCpp: `#define irow ((integer*) &pSkinc->is)`

In this case, the local variable `irow` is made equivalent to the variable `is` within the common block `skinc`, where `is` is the 4th variable in this common block (hence the numeric index 3).

Another issue that must be considered when using *f2c*-converted code in an object-oriented C++ setting is that *f2c* generates calls to independent functions. Once *F77toCpp* converts these functions to class member functions, then their corresponding invocations must be changed as well. A script within *F77toCpp* makes this change, as illustrated by the example in Appendix E.

Finally, the signature of the functions generated by *f2c* differs from what is actually identified by *F77toCpp*. For example, for any argument that is of type character, an argument giving the length of the value is also passed by *f2c*. *f2c*-generated function

signatures are used in *.cpp* files. To maintain consistency between function signatures in *.cpp* and *.h* files, the function signatures identified by *F77toCpp* are not placed in the *.h* files during their creation. Instead, once the *f2c*-converted code is copied into the *.cpp* files, a script is used to parse these *.cpp* files, identify the signatures of member functions, and copy them into corresponding header files. See Appendix F for an example. At this point, the resulting classes can be compiled, and executed.

4.2. Implementation Details

This section describes the implementation details of *F77toCpp*. In particular, the identification of C++ classes using subroutine-calling hierarchy is described. Also, the design details of *F77toCpp*, the directory structure generated and the various scripts used towards automating the conversion process are presented.

Critical to *F77toCpp* is the identification of the subroutine-calling hierarchy. It is from this hierarchy that the function-only C++ classes are generated. The following three class data structures are used to describe this hierarchy: `CTree`, `CTreeNode`, and `CChildNode`. These three classes represent the calling hierarchy which is basically a tree structure, each node in the tree and the links between these nodes in the tree, respectively. They are represented here as conventional C `struct` data types, for simplicity in presentation—though in actual implementation, they are implemented as C++ *class* data types.

```
struct tree {
     struct treenode*    currentNode;
     struct treenode*    root;
     struct treenode*    prevNode;
     int                 maxHeight;
}
```

This structure, corresponding to the `CTree` class, represents the calling hierarchy that is identified from the FORTRAN77 source code. This `tree` data structure is used to implement this hierarchy. Its attributes signify the following:

`currentnode`: Points to the node in the tree which represents the subroutine that is currently being parsed in the FORTRAN77 source file;

`root:` Head of the tree. This is a dummy node, which has references to all the nodes with no parents;

`prevNode`: When a subroutine is being parsed, the first step is to check if it has already been parsed. If it has been parsed, it implies that a node representing this subroutine is already created in the tree. `prevNode` refers to this node in the tree. It points to NULL if this is the first time the subroutine is parsed; and

`maxHeight:` Maximum height of the tree without considering the dummy root node.

```
struct treenode {
     char                *name;
     char                *arguments;
     struct childnode    *children;
     struct childnode    *parent;
}
```

This structure, corresponding to the `CTreeNode` class, represents the individual nodes that are created in the hierarchical tree structure. Each node is a subroutine that is present in the FORTRAN77 source code. It stores references to its parents (subroutines that call this particular subroutine) and its children (subroutines that are called by this particular subroutine). Its attributes signify the following:

`name:` Text string holding the name of the FORTRAN77 subroutine;

`arguments:` Text string holding this subroutine's arguments;

`children:` Linked list of this node's children; that is, the list of all the subroutines that this subroutine calls; and

`parent:` Linked list of the node's parents; that is, the list of all the subroutines that call this subroutine.

```
struct childnode {
     struct treenode     *child;
     struct childnode    *nextChild;
}
```

This structure, corresponding to the `CChildNode` class, is a linked list used by a parent to keep track of all its children; that is, for a subroutine to keep track of all the subroutines it calls. Its attributes signify the following:

`child:` Points to the actual tree node in the subroutine-calling hierarchy, which represents the subroutine that a parent is calling; and

`nextChild:` Points to the next child in the linked list.

Once the subroutine-calling hierarchy is identified from the FORTRAN77 source code, the algorithm mentioned in Chapter 3 is used to generate the C++ classes. This algorithm is based on the height of each node relative to the height of the entire tree.

4.2.1. *F77toCpp* Class Diagram

The class diagram for *F77toCpp* is shown in Figure 4.4. The following are the classes that were implemented as part of *F77toCpp*:

comments.l - This is a lex file for parsing the FORTRAN77 source file and removing comments from the source file. The comments are dumped into the

Figure 4-4 Class diagram for *F77toCpp*

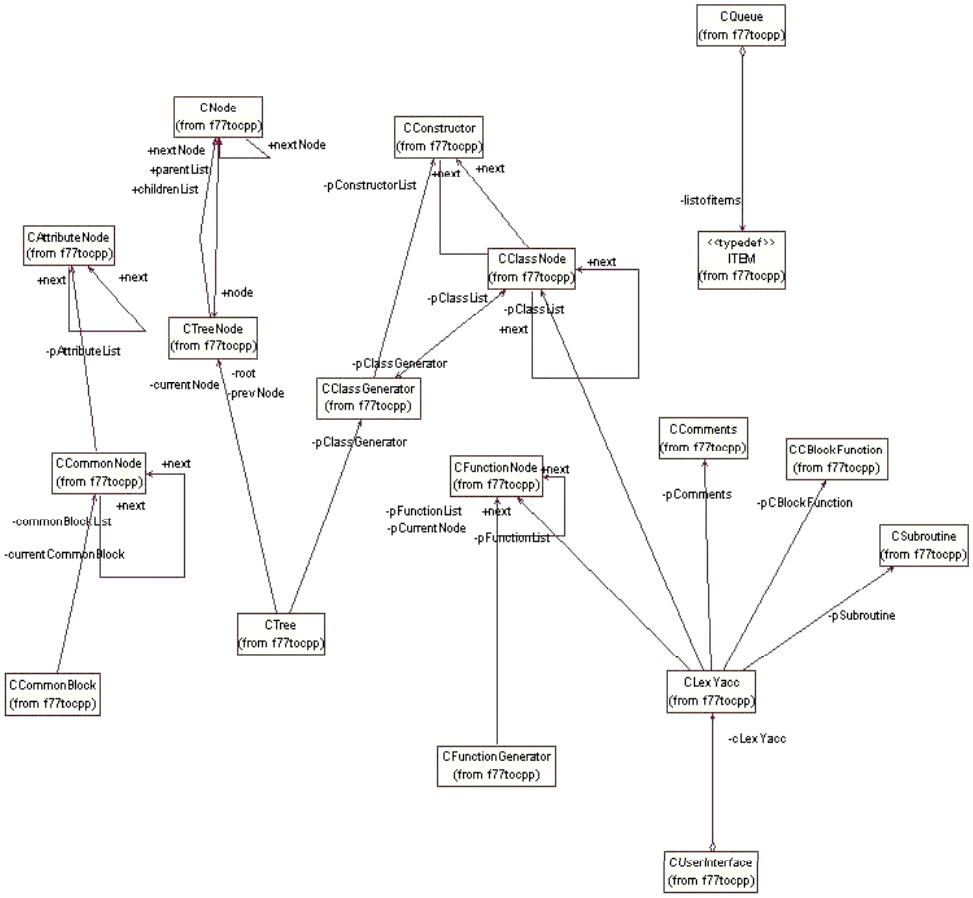
f77cpp_comments file in the current directory, and the rest of the code is copied to the *f77cpp_temp.f* file. This lex file represents the `CComments` class.

processcblock_function.l - This is a lex file for parsing the FORTRAN77 source code and collecting information about common blocks and functions in the code. This file represents the `CCBlockFunction` class.

subroutine.l - This is a lex file for parsing the FORTRAN77 source code and generating the subroutine/function-calling hierarchy. This file represents the `CSubroutine` class.

CTree.cpp - This class represents the tree used to store the details about the calling hierarchy.

CTreeNode.cpp - This class represents each node in the subroutine-calling hierarchy (tree). Each node maintains the program unit name, the list of program units that call



this subroutine/function, and the list of subroutines/functions that are called by this program unit.

CNode.cpp - This class represents each node in the list of parents and the list of children that a `CTreeNode` object maintains.

CQueue.cpp - This class implements the Queue abstract data type. It is used to display the subroutine-calling hierarchy in level-order traversal.

CClassGenerator.cpp - This class is the main controller class. It maintains the list of classes that needs to be generated by the tool, and it maintains, for each class, the list of arguments that needs to be added to the constructor of that particular class.

CClassNode.cpp - This class represents the node in a linked list maintained by the `CClassGenerator` object. Each of these nodes maintains the information about each subroutine and function that has been parsed from the FORTRAN77 source code. Information such as the name of program unit, the name of the program unit

that invoked this particular program unit, and whether this program unit is going to be converted into a class or a function in a class, etc; are stored here.

CConstructor.cpp - This class represents the node in the linked list of constructors for all the classes generated by the tool. Each node maintains the class to which the constructor belongs, and the list of arguments for this constructor.

CCommonBlock.cpp - This class maintains the linked list of common blocks present in the FORTRAN77 source code and generates class files for each of those common blocks.

CCommonNode.cpp - This class represents a node in the linked list of common blocks. Each node has the common block name and the linked list of attributes that belong to the common block.

CAttributeNode.cpp - This class represents a node in the linked list used, and it is to store the attributes of a common block.

CFunctionGenerator.cpp - This class is the controller class that maintains the linked list of functions present in the FORTRAN77 source code.

CFunctionNode.cpp - This class represents the node in the linked list of functions that are maintained by the `CFunctionGenerator` object. Each node maintains the function name, the function arguments, and the return type of the function.

CLexYacc.cpp - This is the controller class that calls the respective classes in sequence to perform the following:

a) Process comments using the `CComments` object;

b) Process common blocks and functions using the `CCBlockFunction` object;

c) Process subroutines using the `CSubroutine` object.

CUserInterface.cpp - This class represents the command line interface, which is used by the tool to interact with the user.

4.2.2. Generated Directory Structure

In the process of generating classes from the subroutine-calling hierarchy and modifying and adapting them to the C++ environment, different directories are created and used. The adaptation of the classes generated by *F77toCpp* is done in an incremental manner,

and for each increment a different directory is created. At each increment, the class files in the earlier directory are processed, adapted and copied into the subsequent directory, until the desired result is obtained. The following are the seven sub-directories that are generated by *F77toCpp* under the user-specified main directory:

- **`F77CPP_ClassesA`**: This directory holds the skeleton *.h* and *.cpp* files of the classes that are identified by *F77toCpp*. These classes have empty member functions.
- **`F77CPP_ClassesB`**: This directory holds the *.cpp* files containing FORTRAN77 subroutine source code as comments followed by the respective member functions. These member functions have their implementation generated by *f2c*. The *.h* files in this directory are duplicates of those in the `F77CPP_ClassesA` directory. The references to structures in the `#define` statements representing common blocks are replaced with references to the appropriate classes.
- **`F77CPP_ClassesC`**: This directory holds the class files once their common block structures are deleted and the references to these structures are replaced with references to the appropriate classes representing these common blocks. Also, the member functions of these classes are given class scope.
- **`F77CPP_ClassesD`**: This directory holds the final class files, ready to be used. The `extern` function declarations are removed from *.cpp* files. Function calls are replaced with appropriate calls to the member functions of the classes to which these functions belong.
- **`F77CPP_FORTRAN`**: This directory holds the individual *.f* files that are generated by *fsplit* from FORTRAN77 input file.
- **`F77CPP_f2c`**: This directory holds the individual *.c* files that are generated by *f2c*, using the individual *.f* files from the `F77CPP_FORTRAN` directory.
- **`F77CPP_Scripts`**: This directory contains the script files that are used for the automated conversion process.

4.2.3. Scripts

Once the skeleton classes are identified and generated by *F77toCpp*, a series of UNIX scripts are used to add functionality to these classes and adapt their code to the C++ environment. Section 4.1 discussed a number of issues that need to be addressed during this conversion. Once these scripts have been run, the resulting classes are ready to be used. The following outlines the scripts that are used to add functionality to the skeleton classes generated by *F77toCpp* and to modify them to incorporate appropriate C++ coding notations:

- RUN_insert_comment: This script inserts the keyword "F77CPP: FUNCTION" in the FORTRAN77 files present in the `F77CPP_FORTRAN` directory. This keyword is used to identify a function.
- RUN_f2c: This script uses *f2c* to convert all the FORTRAN77 files in the `F77CPP_FORTRAN` directory into C files. These C files are saved to the `F77CPP_f2c` directory.
- RUN_insert_f: This script inserts the original FORTRAN77 code as comments, and the C code generated by *f2c,* into the *.cpp* files in the `F77CPP_ClassesA` directory. The resulting files are stored in `F77CPP_ClassesB` directory.
- RUN_replace_define: This script replaces the `#define` statements that make a local variable equivalent to a common block variable, with references to classes representing the respective common blocks. These `#define` statements refer to structures representing common blocks. The resulting files are stored in the same directory; namely, the `F77CPP_ClassesB` directory.
- RUN_replace_struct: For each `*.C` file in `F77CPP_ClassesB` directory, this script removes all the structures that are used to reference, represent, or store these common blocks, and replace the references to the common block structures with references to classes that correspond to the respective common blocks. It also inserts class names prior to the subroutine definition and removes the "_" after the subroutine definition that was generated by *f2c*. The resulting files are placed in the `F77CPP_ClassesC` directory.

- RUN_replace_signature: This script identifies and copies the member function signatures into the respective header files for each of the `*.C` files in the `F77CPP_ClassesC` directory. The resulting files are placed in the same directory; namely, the `F77CPP_ClassesC` directory.
- RUN_append_class: This script removes all the `extern` function declarations, record the names of each such function, and the class to which this function belongs for each `*.C` file in the `F77CPP_ClassesC` directory. The function references are updated and the results are stored as `*.C` files in the `F77CPP_ClassesD` directory.

The resulting class files in the `F77CPP_ClassesD` directory are the final files which can then be used as is. To illustrate the result of this conversion process, Appendix G shows part of a FORTRAN77 code as it is converted into C++ using *F77toCpp*.

CHAPTER 5.
RESULTS

The AirCraft SYNThesis computer program, ACSYNT, was originally developed by the NASA Ames Research Center, Moffet Field, California, starting in the early 1970's, for conceptual design studies of advanced aircraft. In particular, it was developed to fulfill the need to study the effects of advanced technologies on future aircraft [1]. Though there were other aircraft synthesis codes, ACSYNT was one of the first to be designed in such a modular and robust fashion that a non-linear optimization code could be included to enhance the conceptual design process.

The ACSYNT code was designed to have the flexibility to analyze a wide range of civil and military aircraft including fighters, bombers, and transports. The ACSYNT program is an interdisciplinary aircraft synthesis program for application in the early design stages of an aircraft. The program uses the following modules: (1) Geometry, (2) Trajectory, (3) Aerodynamics, (4) Propulsion, (5) Stability, (6) Weights and Structures, (7) Takeoff performance, (8) Mission performance, (9) Cost, and (10) Sonic boom. The code includes methodology for converging to a vehicle design, and for optimizing the vehicle design for a particular objective function (such as gross takeoff weight), subject to various constraints using the COPES/CONMIN optimization program. When combined with a non-linear optimization code, it is important for every aspect of the synthesis process to be correctly modeled so that the optimization will seek feasible and realistic conclusions. For this reason, the discipline modules of ACSYNT are parameter-driven with equations which are derived largely from theory, as opposed to using table look-up methods. Numerous correlation studies with existing aircraft have been performed, and they have shown ACSYNT to be approximately 80 percent accurate.

Hence, ACSYNT is a good program for determining the technology items that are most important for research, and for showing the trends of future aircraft [1].

When using ACSYNT, the first step is to execute the Geometry module to define the aircraft dimensions and physical shape. Then the Trajectory module is run to execute this aircraft at an initial weight estimate along a specified mission. The Trajectory module, in turn, calls the Aerodynamics and Propulsion modules to determine the performance at each mission point. After the mission is finished, the weights are calculated and compared to the initial estimate, and an iteration scheme is then used to converge upon the appropriate weight. Once the aircraft is converged, additional modules are executed which do not affect the mission analysis, such as the Takeoff and Stability modules. Additional modules can be run on the side, such as the Advanced Aero module, to compare the results of the simpler and more advanced methods. These results can then, if required, be matched using the simple method to keep execution time reasonably rapid. For example, a single pass through a mission analysis might require 2500 calls to the Aerodynamics module. If an advanced method module were called each time, then the result would be a set of extremely long execution times such as during optimization. ACSYNT permits quick estimations of configurations, and as knowledge of the conceptual design increases, it also permits more accurate predictions. If a large amount of data is available on a design, then the many matching parameters permit accurate matching of the configuration so that precise mission studies can be performed [1].

The ACSYNT system was chosen as a source for the case studies used in this thesis research for the following reasons: Its code is large and complex; it is written in a variety of languages including FORTRAN77, C, and C++; and it remains important to maintain and evolve this code for the benefit of the aircraft industry. And, as with many other old scientific codes, the availability of FORTRAN77 programmers to work on this code is dwindling while the availability of C++ programmers is strong and growing. In short, ACSYNT represents the ideal target for an automated structured FORTRAN77 to object-oriented C++ conversion system.

This chapter presents the results obtained from converting three modules within the ACSYNT system into object-oriented C++ code using *F77toCpp*; namely, the

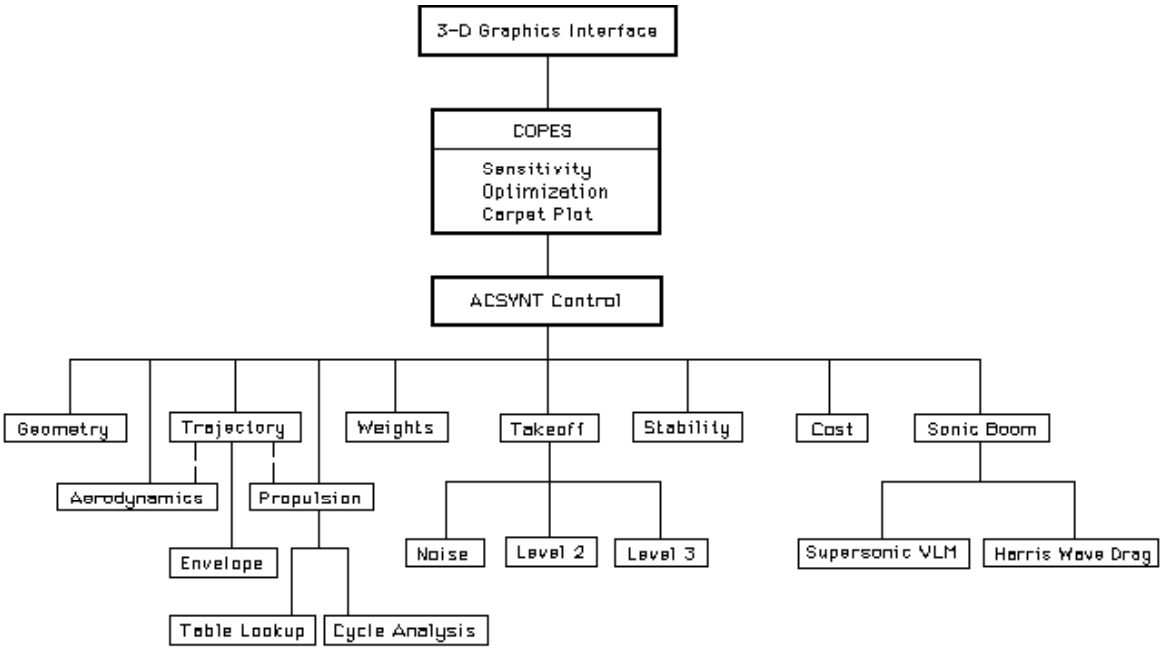


Figure 5-1 ACSYNT control structure [1]

modules: Weights, Control, and Aerox (identified as Aerodynamics in Figure 5.1). Initially, the Weights module and the Control module were studied and manually converted into object-oriented C++ code. The results of these conversions are presented here. Also presented are the results of the automated conversion, which was done subsequently using *F77toCpp*. A comparison is made between the automated conversion results and the manual conversion results. Next, the automated conversion results of the Aerox module are presented. Aerox was considered as a test case at a later stage when the *F77toCpp* had already been developed and tested for the Weights and Control modules. Hence, the Aerox module was never converted manually. The Weights and Control modules were integrated with the ACSYNT system by creating an interface between the FORTRAN77 and C++ codes. The integration process, the issues that arose during this process and the results are discussed. Finally, a comparison is made between the results of original FORTRAN77 code and the results obtained after the integration of the FORTRAN77 and C++ codes.

5.1. Software Development Environment

F77toCpp was developed using the following resources:

- Linux operating system (Red Hat 7.1)
- C compiler (GCC v2.96)
- C++ compiler (GCC v2.96)
- FORTRAN77 compiler (GCC v2.96)
- f2c (May 10, 2000) [6,7]
- fsplit (May 12, 2000) [5]
- flex (v2.5.4)
- awk (GNU v3.0.6)

5.2. Weights Module

Initial vehicle configuration along with the estimated gross weight is given as input to the ACSYNT program. Based on this input, the Weights module is executed and a new gross weight is determined. If the estimated and calculated weights are the same within a prescribed tolerance (e.g., |west/wcal-1| < 0.0001), the vehicle weight is said to be converged. Although a tolerance of 0.0001 may seem too precise to be meaningful, it is necessary from an analytical point of view, because the sensitivity of the weight and the performance to small changes in vehicle parameters are used to guide the design process.

Because the calculated weight is dependent on the estimate, this is necessarily an iterative process. The converged weight is determined by linear and quadratic interpolation. This process usually requires calculating the gross weight (one iteration through the program) from three to six times, depending on the accuracy of the initial estimate. If no upper bound on the weight can be found, then the vehicle cannot be converged (i.e., the vehicle cannot perform the mission at any weight), and the process is terminated when the weight has reached a maximum value specified by the user.

The Weights module was selected as one of three case studies used for this thesis research. It was selected as the first case study because it was relatively trivial module within the ACSYNT system. Figure 5-2 illustrates its subroutine-calling hierarchy. The module consists of 3,238 lines of FORTRAN77 source code across seven subroutines and

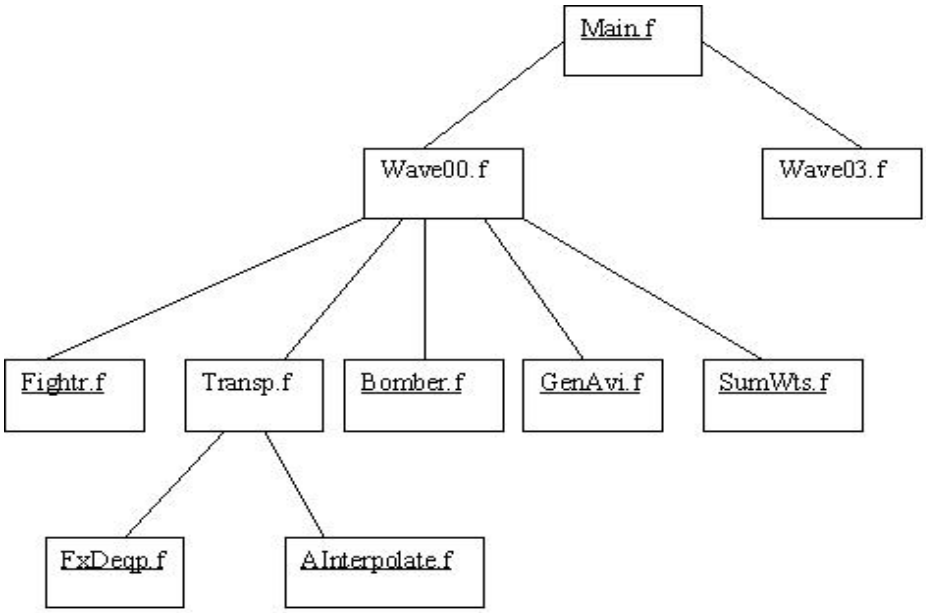


Figure 5-2 Subroutine-calling hierarchy of Weights module

three common blocks. Note that `FxDeqp.f` and `Ainterpolate.f` in Figure 5-2 are not a part of Weights module. Instead, they are external subroutines that are called from within the module.

The following subsections present the manual conversion process and the automated conversion process as they apply to the conversion of the Weights module from structured FORTRAN77 into object-oriented C++. Their results are compared and discussed.

5.2.1. Manual Conversion Process

The manual conversion of the chosen sub systems was initially carried out to prepare a test bed, which was used as a reference to compare the results from the automated conversion process. The quality of the code obtained from *F77toCpp* can thus be compared and analyzed and *F77toCpp* can be further enhanced to generate better quality results.

```
      IF(ITYPE .EQ. 1) CALL TRANSP(NERROR,KGPRNT)
      IF(ITYPE .EQ. 2) CALL FIGHTR(NERROR,KGPRNT)
      IF(ITYPE .EQ. 3) CALL BOMBER(NERROR,KGPRNT)
      IF(ITYPE .EQ. 4) CALL GENAVI(NERROR,KGPRNT)
```

Figure 5-3 FORTRAN77 code to determine the top-level code flow within the Weights module

```
if (strcmp(type, "TRANSPORT") == 0)
{
    pAircraft = new CTransport(pWtstemp, pWtsnewtemp);
    int itype = 1;
    pAircraft->set_itype(&itype);
}
else if (strcmp(type, "FIGHTER") == 0)
{
    pAircraft = new CFighter(pWtstemp, pWtsnewtemp);
    int itype = 2;
    pAircraft->set_itype(&itype);
}
else if (strcmp(type, "BOMBER") == 0)
{
    pAircraft = new CBomber(pWtstemp, pWtsnewtemp);
    int itype = 3;
    pAircraft->set_itype(&itype);
}
else if (strcmp(type, "AVIATION") == 0)
{
    pAircraft = new CGAviation(pWtstemp, pWtsnewtemp);
    int itype = 4;
    pAircraft->set_itype(&itype);
}
pAircraft->CalculateWeights (nerror, kgprnt);
```

Figure 5-4 Manually converted code corresponding to the FORTRAN77 code in Figure 5-3

The Weights module was analyzed thoroughly before it was redesigned using object-oriented design techniques. The main flow in the Weights module was based on the aircraft type. Hence, the code flow was selected depending on the aircraft type. This is illustrated in Figures 5-3 and 5-4. In Figure 5-3, the various subroutines are called depending on the variable `ITYPE`, which represents the aircraft type. This code was manually converted into C++ code using the *strategy* design pattern [34], as shown in Figure 5-4.

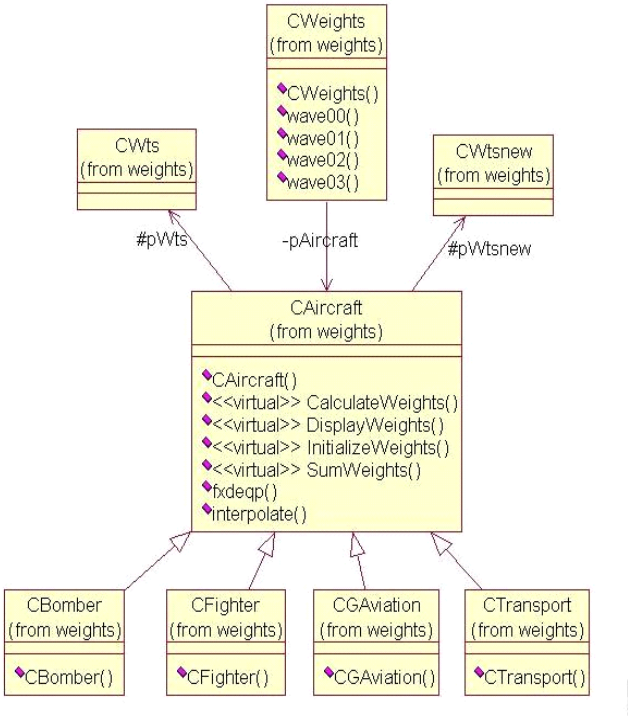


Figure 5-5 UML class diagram for the manually converted Weights module

During the design phase, all the non-local variables that were part of the `sumwts.f`, `wave03.f`, `ainterpo.f`, and `fxdeqp.f` files, were declared in base class `CAircraft`. The non-local variables that were used in more than one of the files `bomber.f`, `fightr.f`, `transp.f`, and `genavi.f`, were also declared in the `CAircraft` class.

At run-time, the object of class `CWeights` initializes `pAircraft` (a pointer to the object of class `CAircraft`) with any one of the aircraft types *bomber*, *transport*, *fighter*, or *general aviation*, depending on the user's choice. When a function call is made on this `pAircraft` pointer, the actual derived type of this object, to which this pointer points, is retrieved, and the function call is made on that object. This is possible because of the run-time polymorphism. The derived class, in turn, calls the base class function. The `main.cpp` file instantiates an object of class `CWeights` and calls its methods. Figure 5-5 shows the class diagram for manually converted Weights module.

5.2.2. Automated Conversion Process

The FORTRAN77 Weights code was given as input to *F77toCpp*. The value of *x* was set to 2 because the height of the Weights subroutine-calling hierarchy, *Maxheight*, is 3. *F77toCpp* then converted the common blocks into classes. All the subroutines in the hierarchy whose height >= (3 + 1)/2 became classes; that is, the subroutines `wave00` and `transp` were converted into classes. Figure 5-6 shows the resulting UML class diagram.

During this automated conversion of the FORTRAN77 Weights module into C++, there were instances when manual intervention was needed to generate the C++ code, to compile it, and to run it successfully. For example, the FORTRAN77 Weights code had the `INCLUDE WTSGLO.INC` statement in some of the subroutines. This statement had to be manually replaced by the contents of the file WTSGLO.INC, because *F77toCpp*

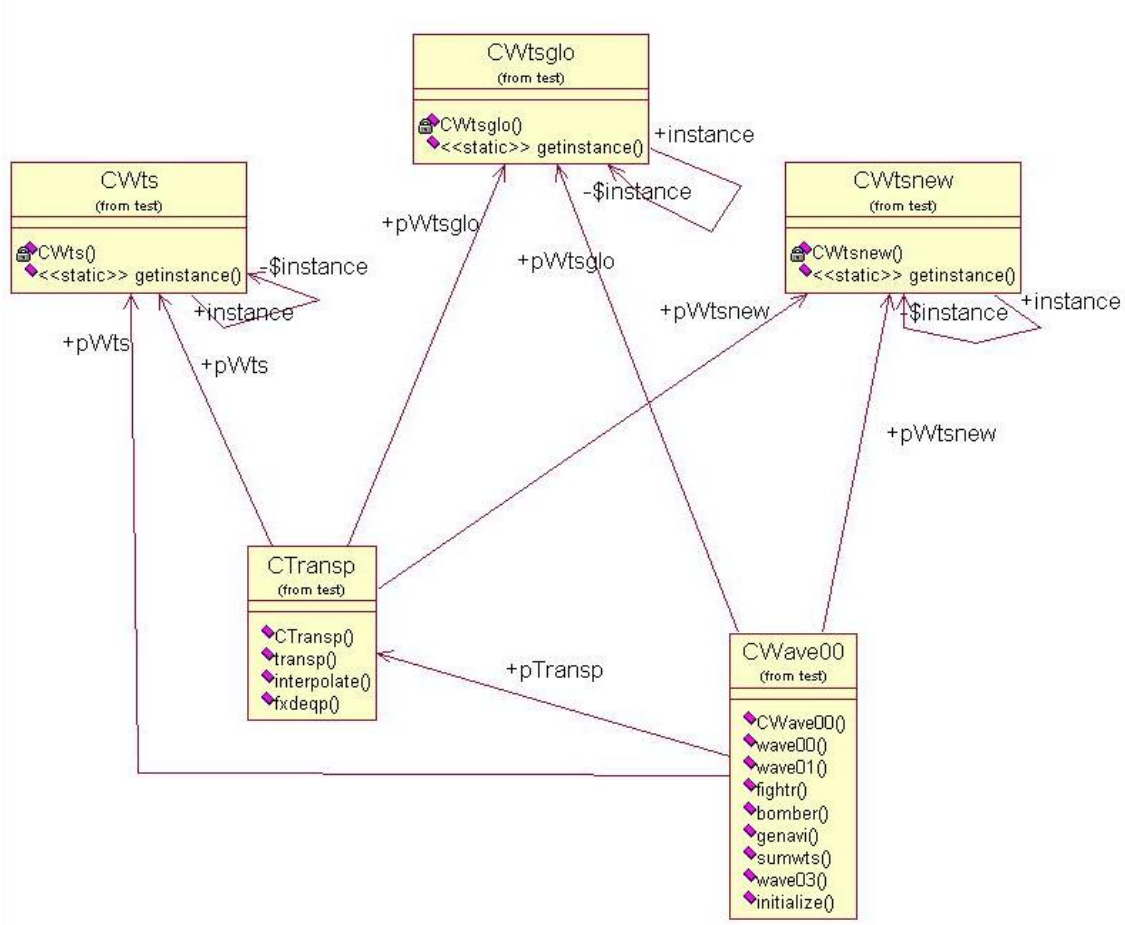


Figure 5-6 UML Class diagram for Weights module that was converted using *F77toCpp*

does not currently have the ability to process include files.

Furthermore, *f2c* renames some variables by appending underscores to avoid conflict with keywords. For example, the variable `TYPE` that belongs to `WTS` common block, was converted into `type__`. *F77toCpp*, on the other hand, identifies it as `type` and declares it in the `CWts` class, which represents the `WTS` common block. This mismatch error had to be corrected manually. Some of the *.cpp* files used math library functions and hence, the math header file *math.h* had to be included manually where needed.

Each FORTRAN77 subroutine was converted individually into C++ functions by *f2c*. Hence, some of the static variables that were used by *f2c* to declare constants in C++ code were replicated across these C++ functions. When this converted code was inserted into class files, the static variable re-declaration resulted in compilation error. These re-declaration statements had to be removed manually. Once *F77toCpp* was done generating the object-oriented C++ classes, it took approximately 30 to 50 minutes of final manual modifications to get the code compiled.

5.2.3. Validation

To validate the numerical correctness of the converted codes, the original FORTRAN77 Weights module was integrated into an independent stub program where the common block variables were populated from a file containing values that has been obtained from ACSYNT at runtime. Next, the converted C++ versions of the Weights module, first the manually converted and then automation converted, were each integrated into equivalent, independent stub programs. The outputs of these three programs were identical to the corresponding output generated by ACSYNT for the one input data set that they were tested against.

One of the major differences between the manually converted code and the automated conversion code is that the manual version of the C++ Weights subsystem identifies an *is-a* relationship and implements it using inheritance. Reusability of code, which is the essence of object-oriented design, is thus achieved. Currently, *F77toCpp*

does not have the capability to identify *is-a* relationships, and, thus, it cannot support code reuse in the form of inheritance.

The quality of the C++ code generated by *F77toCpp* is determined by measuring the coupling and cohesion metrics within the Weights sub-system. The details of these metrics are presented in Section 3.2.2. These metrics are measured for the C++ Weights module and are presented in the Section 5.5.

5.3. Control Module

The ACSYNT Control module organizes the various design and analysis functions and exercises the disciplinary modules. All data is transferred between modules via a single common block labeled `global`, which contains only that information which is transferred across modules. The general-purpose optimization program COPES may be applied to problems other than vehicle synthesis if the appropriate modules are provided. Together, this design allows for straightforward program expansion and provides a control program for general applications [1].

Two primary size limitations are imposed on ACSYNT: First, the amount of data transferred through the `global` common block, and, second, the minimum amount of memory required for the program. Currently, the data transfer management system contains approximately 1600 variables which correspond to about 5 Megabytes of data. The Control module is fairly simple, with it containing 1,440 lines of FORTRAN77 source code in a singular source file, 11 subroutines, and 17 common blocks [1]. The subroutine-calling hierarchy is illustrated in Figure 5-7.

The Control module was selected as the second case study used to analyze the results generated by *F77toCpp*. This module controls the flow in the ACSYNT code. The Control module calls different subroutines and each subroutine, once executed, returns control back to this module. Thus, this module is functionally very different from the Weights module.

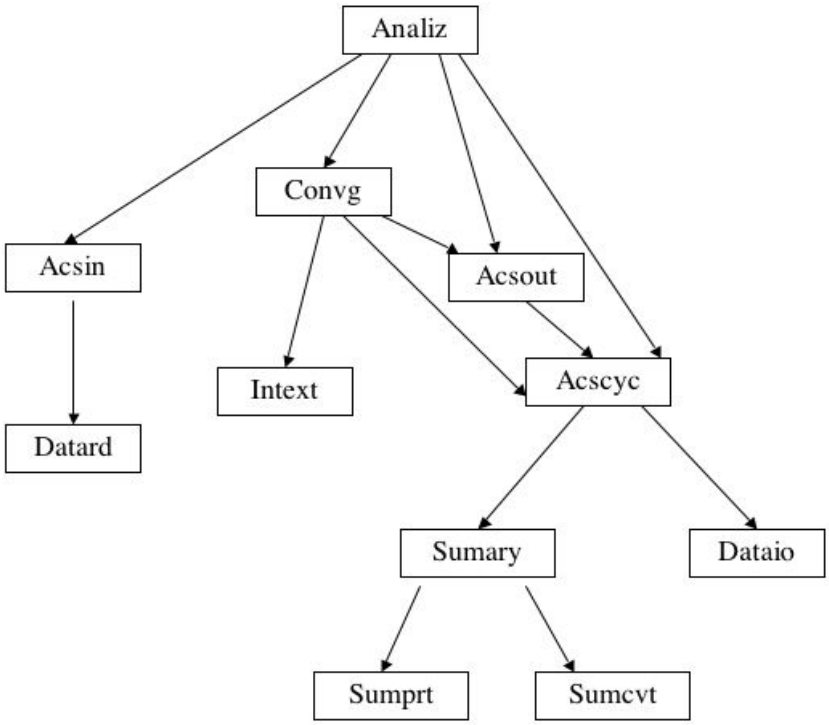


Figure 5-7 The subroutine-calling hierarchy of the Control module

The following subsections present the manual conversion process and the automated conversion process as they apply to the conversion of the Control module from structured FORTRAN77 into object-oriented C++. Their results are compared and discussed.

5.3.1. Manual Conversion Process

The Control module was analyzed and redesigned using object-oriented design techniques. The main concept of the Control module is that, depending on the input, one of the subroutines `Acsin`, `Convg` and `Acscyc`, or `Acsout` is called from the subroutine `Analiz`. The Control code is illustrated in Figure 5-8. Since there was no common functionality between these subroutines and there was no *is-a* relationship between them, inheritance and polymorphism were not considered as a solution. Instead,

```
      GO TO (10,20,30), ICALC

      C -------------------- INPUT ---------------------
      10 CALL ACSIN
      RETURN

      C ----------------- EXECUTION ------------------
      20 IF (MEXEC.LE.0) RETURN
      CALL CONVG
      CALL ACSCYC (ICALC,NERROR,MEXEC,NEXEC,IGEO,IPDBG,KGPRNT)
      RETURN

      C ------------------- OUTPUT -------------------
      30 CALL ACSOUT (NERROR)
```

Figure 5-8 Example code from Control module

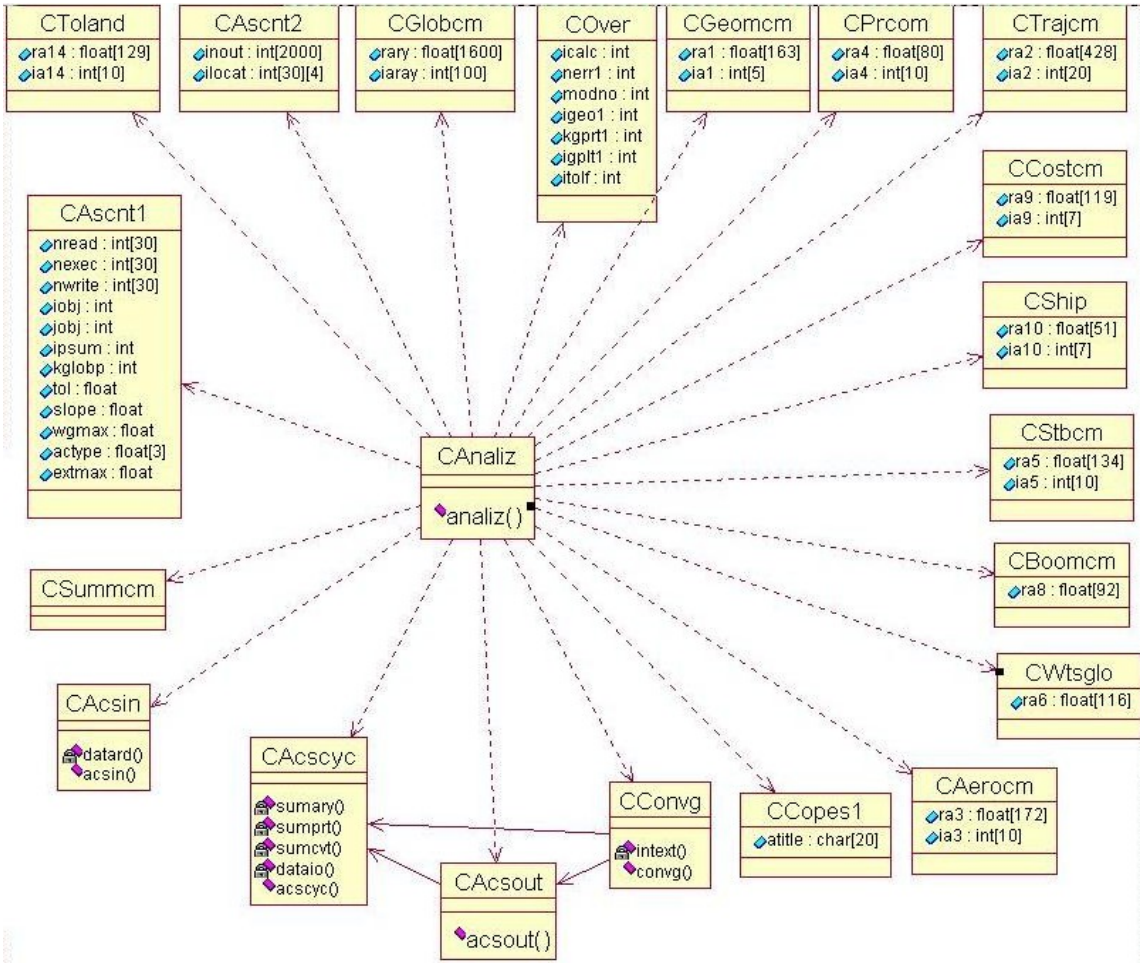


Figure 5-9 Class diagram for the manually converted Control module, including the classes that represent the FORTRAN77 common blocks

the FORTRAN77 code was divided into five different classes, each representing one of the above-mentioned subroutines. The subroutines that were invoked by these five top-level subroutines were made member functions of their respective classes. The design of the C++ code is illustrated in Figure 5-9. Note how the five top-level subroutines are made into the classes `CAnaliz`, `CAcsin`, `CConvg`, `CAcsout`, and `CAcscyc`; the subroutine `Datard` is made a member function of the class `CAcsin`; the subroutine `Intext` is made a member function of the class `Cconvg`; and so on, in accordance with the hierarchy shown in Figure 5-7.

5.3.2. Automated Conversion Process

The automated conversion process was applied to the Control module source code. As mentioned in Section 3.2, the grouping of subroutines into classes and the quality of the classes generated by the clustering algorithm depends on a user-set parameter *x*. The cohesion and coupling metrics for the classes generated depends on the value set for this factor *x*. For the Control module, the value of *x* was chosen as 2, selecting this choice from 2 or 3 because the M*axheight* of Control subroutine-calling hierarchy was 6. This caused the subroutines whose height >= 6/2 to become classes. In addition, all 17 common blocks were converted into classes. Figure 5-10 illustrates the resulting class diagram. Note how the C++ classes generated by *F77toCpp* are same as those that were generated manually. The resulting C++ code was used to integrate with the ACSYNT system and the output was verified against the output of the original FORTRAN77 system. The details of this integration process are presented in Section 5.5.

5.4. Aerox (Aerodynamics) Module

The third case study used to test the capability of *F77toCpp* was the Aerox module. This module was fairly complex to the extent that it consisted of 17,215 lines of FORTRAN77 source code when merged into a single file, 132 subroutines and functions, and 63 common blocks. Due to its large size and complexity, this module was not manually converted into object-oriented C++ code; hence, only the results of the automated conversion process are presented here.

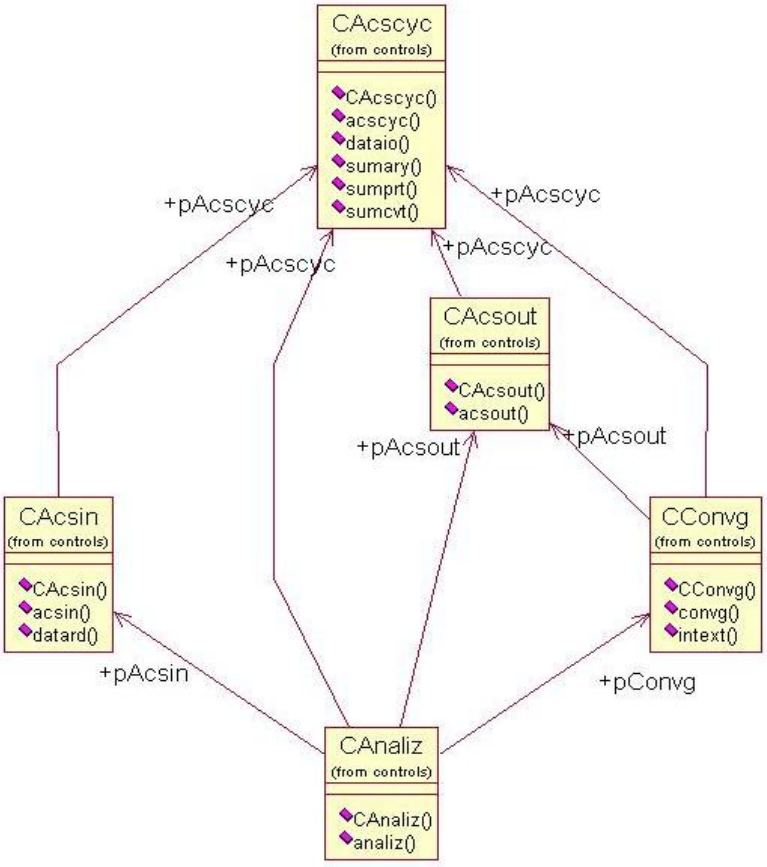


Figure 5-10 Class diagram of Control module

5.4.1. Automated Conversion Process

As discussed in Section 3.2, *F77toCpp* has a key parameter *x*, which controls the grouping of subroutines and functions into classes. This parameter is based on the maximum height of the module's subroutine-calling hierarchy, in that the resulting classes would at most span a specified number of calling levels. In the case of the Aerox module, the maximum height (*MaxHeight*) of the calling hierarchy is 10. Hence, we ran three case studies using the values 2, 3, and 4 for *x*, which corresponds to the values *MaxHeight/2*, *MaxHeight/3*, and *MaxHeight/4*, respectively. The statistics for these three cases are listed in the Table 5-1. Specifically, it shows the number of classes that were generated based on subroutines, functions, and common blocks, respectively. For instance, in the case of *MaxHeight/2*, 27 subroutines and 12 functions became classes,

Table 5-1 Statistics for the Aerox module

	5 = MaxHeight/2	3 = MaxHeight/3	2 = MaxHeight/4
SUBROUTINES	27	35	49
FUNCTIONS	12	12	14
COMMON BLOCKS	63	63	63
CLASSES	102	108	126

with the remaining 93 subroutines and functions being assigned as members of these 39 classes.

These three sets of C++ classes, excluding those generated from the common blocks, were then separately fed to Rational ROSE where they were reverse engineered into three separate UML class diagrams. The classes generated from the common blocks were not included because the resulting complexity caused Rational ROSE to crash. Based on a subjective analysis of these images, we determined that *MaxHeight/4*

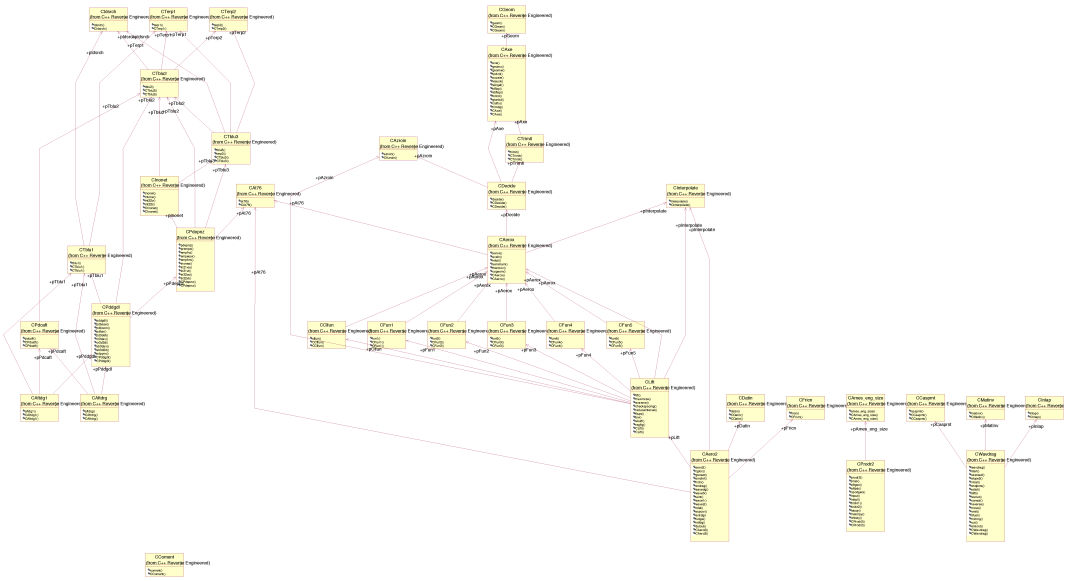


Figure 5-11 (A) UML class diagram of the Aerox module for *Maxheight/2*

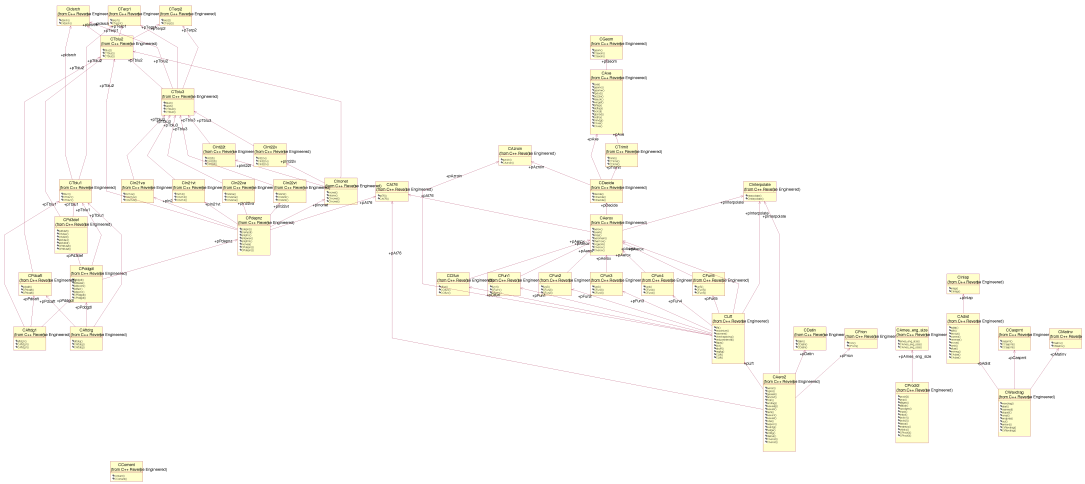


Figure 5.11 (B) UML class diagram of the Aerox module for *Maxheight/3*

grouping best represented our preferred final solution (Figures 5-11 A, B and C). The reason behind this choice is it reduced the number of large classes with respect to the number of lines of code within a class. Also, it is easier to manually merge smaller classes than to split a large class within the current implementation of the test bed. Finally, with *Maxheight/4*, some of the classes that were generated had similar names, and subjectively it was decided that these should be merged manually into one class.

Figures 5-12a and 5-12b, with an actual case shown in Figure 5-13, illustrate a typical example of how we chose to manually merge a number of small classes into a single larger class to avoid needless proliferation of classes. In the case shown in Figure 5-12a, there are three small classes (B, C, and D) that are all called by only a single class

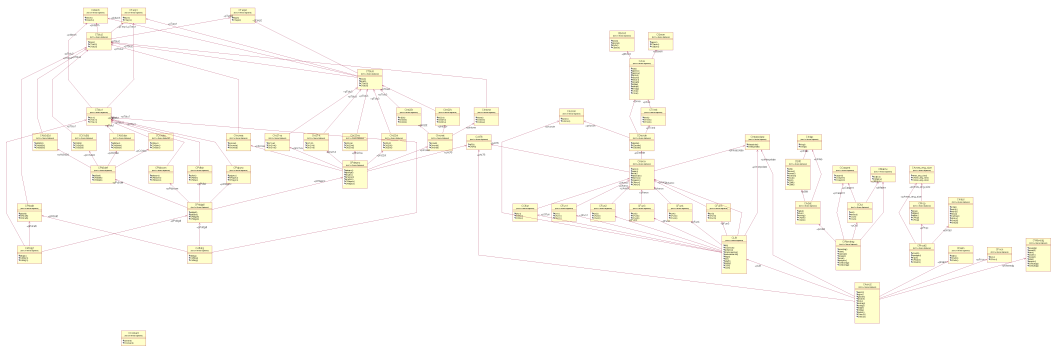


Figure 5.11 (C) UML class diagram of the Aerox module for *Maxheight/4*

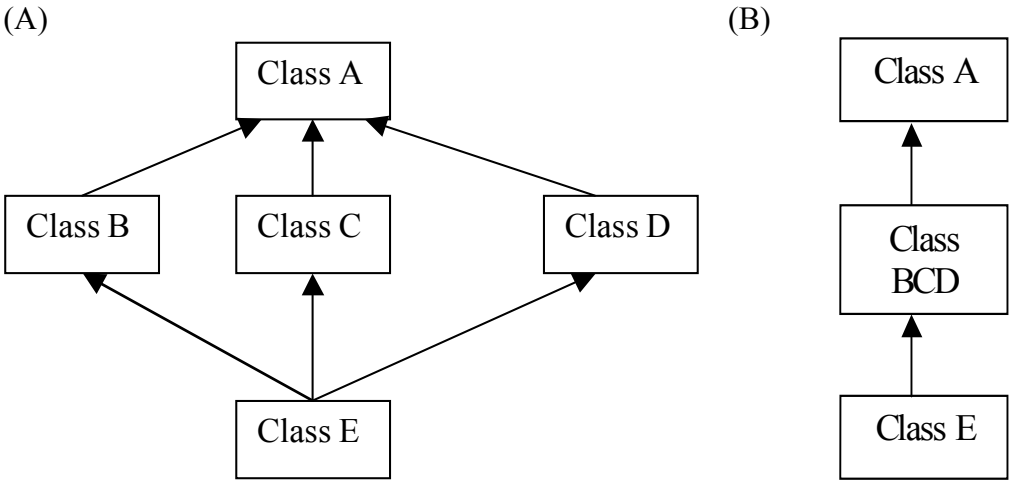


Figure 5-12 (A) Original class structure; (B) Merged class structure

(E), and that all call only a single class (A). We chose to merge the classes B, C, and D into a single class BCD, as shown in Figure 5-12b, because maintaining the three original separate classes does not appear to provide for an improved design understanding or subsequent maintenance advantage. Correspondingly, in Figure 5-13, `CClfun`, `CFun1`, `CFun2`, `CFun3`, `CFun4`, and `CFun5` were merged into once single class `CFun`. In all, we identified six such cases in the Aerox module for the parameter setting $x = 4$. This reduced the number of classes by 13. In addition we merged a few more classes based on their small sizes and member function name similarities. Thus, the total number of classes was manually reduced from 126 to 109; a total reduction of 17 classes. It took about one hour to analyze and identify the classes that could be merged, and about three hours to manually merge these classes.

It is interesting to note that Aerox module consisted of five disconnected sub-modules as illustrated in Figure 5-14. This insight was not apparent until we reviewed the UML class diagrams that were generated via Rational ROSE (Figure 5-11), but it was clearly an important observation from a future software maintenance point of view.

It was not clear if this insight would have been as readily available from output generated by conventional FORTRAN calling hierarchies, such as those generated by the commercial available programs. We believe the key benefit of *F77toCpp* in this regard, when attempting to gain an overall understanding of a 132-subroutine/function module, is the complexity reduction that is achieved by grouping multiple-related and closely-

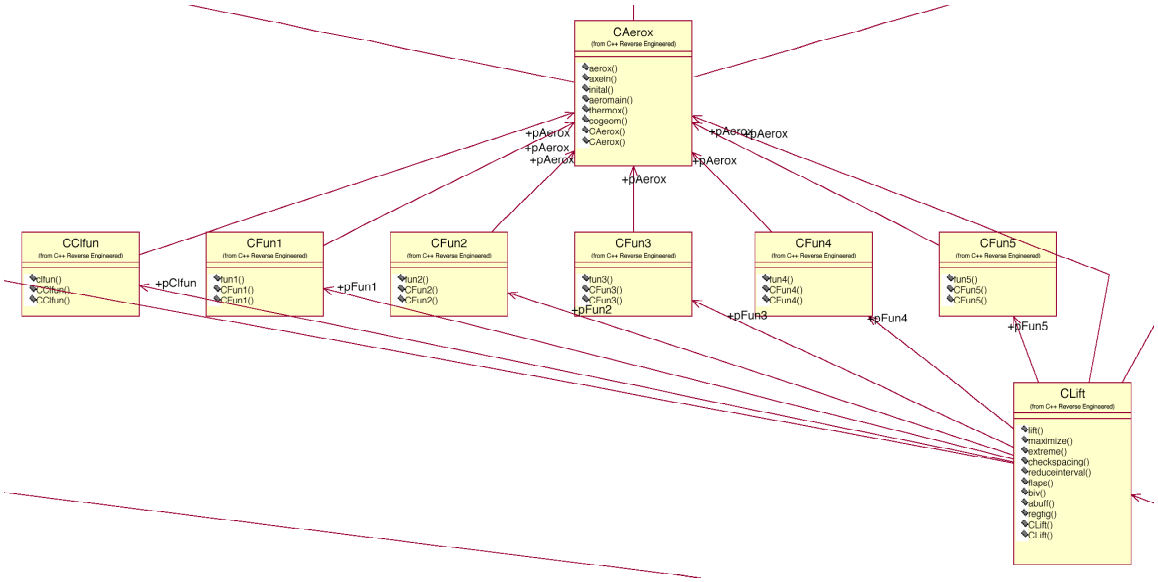


Figure 5-13 The merge of the six classes CClfun, CFun1, CFun2, CFun3, CFun4, and CFun5 into one class.

connected subroutines and functions into a single class. In particular, as we examine the effect of changing *x*, it is similar to a dynamic analysis function adding a new visual dimension to aid the understanding of the code's inherent structure.

5.5. Coupling and Cohesion Metrics for the Case Studies

Section 3.2 identifies two metrics that can be used to analyze the quality of the classes identified by *F77toCpp*; namely, Coupling Between Object classes (CBO) and Information-flow based Cohesion (ICH). CBO is defined as the number of other classes whose methods are used by methods of this class. Since there is only one communication path between any two classes identified, the coupling metric between any two classes is a unit. The ICH metric measures cohesion based on the information flow through method invocations within a class. For a method `m` implemented in class `C`, the cohesion of `m` is the number of invocations to other methods implemented in class `C`. These metrics are used to analyze the quality of the classes generated by *F77toCpp* for the Weights, Control, and Aerox modules.

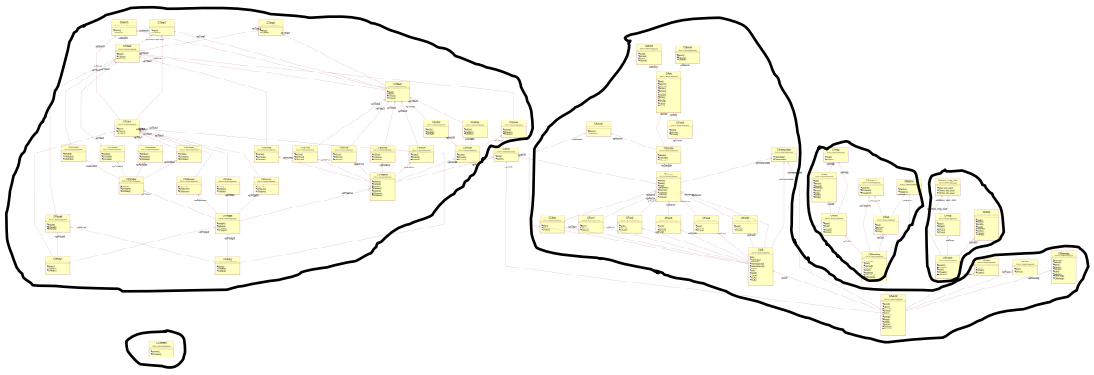


Figure 5-14 The UML diagram provides a visual depiction of the five disconnected sub-modules within the Aerox module for *Maxheight/4*

For the Weights module as shown in Figure 5-6, assuming that no parameters are passed between methods during method invocation,

Total number of subroutines	=	10
Total number of classes	=	2
Total number of communication paths between classes in the entire system	=	1
CBO value for the hierarchy	=	1
ICH (CWave00)	=	6
ICH (CTransp)	=	2

For the Control module as shown in Figure 5-10, assuming that no parameters are passed between methods during method invocation,

Total number of subroutines	=	11
Total number of classes	=	5
Total number of communication paths between classes in the entire system	=	8
CBO value for the hierarchy	=	8
ICH (CAnaliz)	=	0
ICH (CAcsin)	=	1
ICH (CConvg)	=	1
ICH (CAcsout)	=	0
ICH (CAcscyc)	=	4

High cohesion and low coupling is desired in any object-oriented system. For the Weights module in Figure 5-6, coupling is low with high cohesion within classes. This is a desired quality system. For the Control module in Figure 5-10, coupling is very high when compared to the total number of subroutines. Except for one class, cohesion is non-existent. The system quality can be improved by reducing the coupling in the system and increasing the cohesion within the classes generated. This can be achieved by increasing the value of factor *x*, which will then reduce the number of classes generated.

5.6. Integration of C++ Code with FORTRAN77 Code

To validate the results obtained from *F77toCpp*, the C++ code generated by *F77toCpp* was integrated with the ACSYNT system by replacing two of the FORTRAN77 modules individually with their respective converted C++ modules. In particular, the Weights module and the Control module were individually tested, by integrating the C++ code with the ACSYNT system. The results of the original FORTRAN77 code and the results after integrating the C++ code with the FORTRAN77 code were compared. This subsection presents the details of this integration of C++ code into the ACSYNT system, and it discusses and analyses the problems faced during this integration.

When the FORTRAN77 subsystems are interacting with the C++ subsystems, to function properly, the different modules, implemented using different paradigms, have to interact and generate the same results as the all-FORTRAN77 system. As mentioned in Section 3.3, changes are made to the C++ classes representing the common blocks such that they reference the FORTRAN77 common block data structures in place of its own private data members. Hence, the common block data can be shared between still-unconverted FORTRAN77 modules and the object-oriented C++ classes. This avoids duplication of memory associated with the common blocks. To have the FORTRAN77 code interact with the C++ code, an interface was implemented between the FORTRAN77 code and the C++ code [35]. Only a few additional statements were required for the C++ code to interact with the FORTRAN77 code; no additional interface was required. The primary concern while implementing this interface was the memory allocation for the common blocks, which is global data that has to be consistently

accessed across the entire system. Without any modification, both the FORTRAN77 and C++ subsystems would allocate memory for the common block that they should have shared. To avoid this double allocation and non-sharing of memory as intended, the C++ subsystem common block instances were made to reference the memory of the respective common blocks in the FORTRAN77 sub system.

This redirected common block reference is facilitated by that the converted code is translated from FORTRAN77 into C using *f2c*. Hence the resulting C code contains the fundamental references to the shared common block. All that is needed is to modify these references to point to the FORTRAN77 common blocks instead of those created by *f2c*. *f2c* converts common blocks into structures by appending an underscore to the common block names if they do not contain any underscores, or have a pair of underscores appended if they do contain underscores [6]. The C++ object is made to reference the corresponding instance of the C structure representing the common block as illustrated in Figure 5-15. Figure 5-15 illustrates the interface between the FORTRAN77 ACSYNT sub system and the C++ Weights sub system.

A major point to keep in mind while mixing different languages is that, C and C++ default to passing arguments by value; whereas, FORTRAN77 defaults to passing arguments by reference. In other words, the normal way that FORTRAN77 subroutines or functions are called allows them to modify their argument variables inside the subroutine code, while C and C++ do not. Hence, in the interface all variables are passed by reference as is the normal way in FORTRAN77 [29].

Since *f2c* renames the FORTRAN77 procedures and appends an underscore if they do not contain any underscores, when a FORTRAN77 module calls the `WAVE00` subroutine, it is converted into a call to the `wave00_` subroutine internally. This subroutine is implemented in the interface as shown in Figure 5-15 (line 16). The `wave00_` subroutine in the interface instantiates the common blocks used in the Weights sub-system (lines 18-23), and calls the appropriate function on the `CWave00` object, which encapsulates the `wave00` subroutine (lines 24-25). For the FORTRAN77 subroutines that were invoked from the C++ code, an `extern` declaration was used to declare these subroutines with an underscore appended to the FORTRAN77 subroutine

```
1       extern struct wts_1_ wts_;
2       extern struct wtsglo_1_ wtsglo_;
3       extern struct wtsnew_1_ wtsnew_;

4       struct wts_1_ {
5       real title[10], fr, stress, dens, slope[20], k1, k2, k3, k4, k5,
k6, kb, kp1, kp2, cargo, afmach, wfrwg, wfr, type, fliftf;
6       integer iprint, idelt, iobliq, itail, igraph, iwaf, iwairc, iwapu,
iwbody, iwcrew, iwe, iwelt, iwep, iwfs, iwhdp, iwht, iwinst, iwlg, iwna,
iwpa, iwpl, iwps, iwsc, iwtsum, iwvt, iwwing, iwammu, iwbomb, iwcand,
iwcarg, iwetan, iwmiss, iwfeq, iwarm, iwbag, iwbb1, iwca, iwgear,
iwpass, iwenvp, iwpiv, iliftf, iwbb2, kbody, kwing, kerror, maxit,
itype, iwthrv, ithrv, ihsct;};

7       struct wtsglo_1_ {
8       real arcn, arht, arvt, arwg, bdmax, bodl, crew, en, enwing, esf,
pass, qmax, svt, swg, swetb, swetwg, swetcn, swetht, swetvt, swpcn,
swpht, swpvt, swpwg, tcrcn, tcrht, tcrvt, tcrwg, trwg, deslf, ultlf,
vtno, wbb2, weng, wfuel, wgto, techi[9], techg, werr, wfext, tctwg,
fufrac, ws, waf, wairc, wapu, wbody, wcrew, we, welt, wep, wfs, whdp,
wht, winst, wlg, wna, wpa, wpl, wps, wsc, wtsum, wvt, wwing, wammun,
wbomb, wcand, wcargo, wetank, wmiss, wfeq, wftot, spanht, wempty, wthrv,
wpiv, warm, wbag, wbb1, wca, wgear, wpass, wenvp, amprwt, wliftf,
aendia, aenle, engba, tsls, wftrap, swstr, swafex, stewfc, stewcc,
coach, rootwg, rootht, rootvt, rootcn, xqcbwg, xqcbht, xqcbvt, xqcbcn,
sfcsls, crmach, blrang, woil, vsto, clmxto, swetp, wopit; };

9       struct wtsnew_1_ {
10      real wppas, wbppas, wcppas, wempop, wcabg, wattd, wpassv, wcrgct,
wpacfc, wgally, wfmisc, wffurn, wfemer, wcghnd, strut, dpcabn, pyllen,
englen, sflap, sle;
11      integer igear, jfltyp, imnten, imntgr;
12      real flapn, wpcrew, wbpcrew, wpattd, wbpattd, wcabgxtra; };

13      extern "C" {
14      void wave00_(long int *icalc, long int *nerror,
15      long int *igeo, long int *kgprnt, long int *igplt);}

16      void wave00_(long int *icalc, long int *nerror, long
17      int *igeo, long int *kgprnt, long int *igplt) {
18      if (CWts::instance == NULL)
19              CWts::instance = (CWts*) &wts_;
20      if (CWtsglo::instance == NULL)
21              CWtsglo::instance = (CWtsglo*) &wtsglo_;
22      if (CWtsnew::instance == NULL)
23              CWtsnew::instance = (CWtsnew*) &wtsnew_;
24      CWave00 wave;
25      wave.wave00 (icalc, nerror, igeo, kgprnt, igplt);}
```

Figure 5-15 Interface between the FORTRAN77 system and the C++ Weights module

name. It took 4-5 days to complete the integration between FORTRAN77 subsystem and C++ subsystem.

Once the C++ code was integrated with the FORTRAN77 code, the new integrated ACSYNT program was run and its output was compared with that of the original ACSYNT code. In both the cases, the two programs were compiled and run on the same machine using the same compiler systems. For the Weights module, most of the values that were generated by the integrated code and the original code were similar. Only a few values deviated from the original results: In total, eight values differed from the original values from the FORTRAN77 code. These values had a very small deviation. Table 5-2 illustrates a sample of the differences between the original code and the integrated code.

For instance, in Table 5-2, the value of gross weight was **465879.43750** in the FORTRAN77 system, whereas it was **465879.46875** in the integrated system. This small deviation was the result of using different libraries for the intrinsic functions in the two systems. For example, the FORTRAN77 Weights subsystem uses the FORTRAN77-

Table 5-2 Sample difference between results generated by the original ACSYNT code and the ACSYNT code integrated with the C++ Weights code

Original ACSYNT code with FORTRAN77 Weights Module	ACSYNT code integrated with C++ Weights module
Estimated Gross Weight = 530000.0 Calculated Gross Weight = 499522.3 Slope of Wcalc vs. West line = 0.60 Delta between Wcalc and West = -30477.7	Estimated Gross Weight = 530000.0 Calculated Gross Weight = 499522.3 Slope of Wcalc vs. West line = .60 Delta between Wcalc and West = -30477.7
Estimated Gross Weight = 453805.8 Calculated Gross Weight = 465879.4 Slope of Wcalc vs. West line = 0.60 Delta between Wcalc and West = 12073.6	Estimated Gross Weight = 453805.8 Calculated Gross Weight = 465879.5 Slope of Wcalc vs. West line = .60 Delta between Wcalc and West = 12073.6
Estimated Gross Weight = 475425.3 Calculated Gross Weight = 474865.3 Slope of Wcalc vs. West line = 0.44 Delta between Wcalc and West = -560.0	Estimated Gross Weight = 475425.4 Calculated Gross Weight = 474865.3 Slope of Wcalc vs. West line = .44 Delta between Wcalc and West = -560.1
Estimated Gross Weight = 474450.7 Calculated Gross Weight = 474471.7 Slope of Wcalc vs. West line = 0.42 Delta between Wcalc and West = 21.0	Estimated Gross Weight = 474450.7 Calculated Gross Weight = 474471.7 Slope of Wcalc vs. West line = .42 Delta between Wcalc and West = 21.1
** End Vehicle Convergence ** 3 Convergence Iterations Required	** End Vehicle Convergence ** 3 Convergence Iterations Required

provided `cos` function, whereas the integrated system uses the C math-library `cos` function. The initial difference in the value of these variables due to the usage of different intrinsic functions is propagated to all the other variables that use them subsequently.

For example, in the FORTRAN77 code, the `SUBROUTINE BOMBER` has the following code:

```
WHT=(5.38*((ULTLF*WGTO*EXPHT*EXPHT*ARHT)/(100000.*SWG*TCRHT
*(COS(SWPHTR)**1.8)))**0.555)*0.7
```

F77toCpp converts the above code into the following C++ code:

```
d__2 = (double) cos(swphtr);
d__1 = (double) (pWtsglo->ultlf * pWtsglo->wgto * expht *
expht * pWtsglo->arht / (pWtsglo->swg * (float)1e5 * pWtsglo-
>tcrht * pow_dd(&d__2, &c_b3)));
pWtsglo->wht = pow_dd(&d__1, &c_b2) * (float)5.38 * (float).7;
```

Notice how, after executing these codes, the value of the cosine calculation, and subsequently the value of `WHT` differ for the same input:

FORTRAN77			C++		
ULTLF	=	3.75	pWtsglo->ultlf	=	3.75
WGTO	=	530000	pWtsglo->wgto	=	530000
EXPHT	=	957.39849854	expht	=	957.39849854
ARHT	=	4.38999987	pWtsglo->arht	=	4.38999987
SWG	=	4605	pWtsglo->swg	=	4605
TCRHT	=	0.11	pWtsglo->tcrht	=	0.11
SWPHTR	=	0.715585350990	swphtr	=	0.715585350990
COS(SWPHTR)	=	0.754709362984	cos(swphtr)	=	0.754709345568
WHT	=	7657.45068359	wht	=	7657.45019531

For the Control module, there was no deviation between the results generated by the original ACSYNT system and the integrated system. The results of the original code and the integrated code were identical.

5.7. Observations

The following observations were made during the process of developing the automated conversion tool and executing the test cases using this tool:

1. The conversion time is minimal. To convert a FORTRAN77 code into an object-oriented C++ code, the manual effort requires a time span in the range of months. The automated conversion process is completed in minutes when using the *F77toCpp*.
2. Once the classes are generated by *F77toCpp*, a manual effort is required for some minor modifications. This amounts to a negligible time when compared to the time spent on manually converting the FORTRAN77 code into object-oriented C++ code. The manual adjustments for the Weights module required a couple of days because it was the first module that was tested. For Control module it took only a couple of hours.
3. Once the classes are manually adjusted and compiled, they are integrated with the FORTRAN77 system. For Weights module, it took 4-5 days to integrate with the FORTRAN77 system. For Control module, it took couple of days to integrate with the FORTRAN77 system.
4. Automated conversion process promoted a standard and consistent naming convention. Hence, human error introduction is reduced. Due to the invariable human error factor, the manual conversion might result in new erroneous code. Thus, the code generated using the automated conversion process should be more maintainable in the long term than the code converted manually.

CHAPTER 6. CONCLUSIONS AND CONTRIBUTIONS

The goal of this research is to move towards automating the conversion of FORTRAN77 legacy code into object-oriented C++ code. Given that most software professionals today focus on object-oriented technologies, it is significantly easier to staff the maintenance of object-oriented code than that of legacy FORTRAN77 code. Furthermore, object-oriented programming itself facilitates software maintenance. Hence, the software developed as part of this thesis research, *F77toCpp*, represents an important a step towards improving the maintenance of legacy FORTRAN77 codes. *F77toCpp* eliminates just about all manual intervention associated with the conversion of procedural FORTRAN77 code into object-oriented C++ code. This automation therefore dramatically reduces the time, effort, and errors that are inherent in a manual conversion process. It furthermore promotes coding consistency, which enhances the maintainability of the converted legacy code.

6.1. Concluding Remarks

This thesis identifies potential classes from the FORTRAN77 code using the FORTRAN77 common block structures and the FORTRAN77 subroutine- and function-calling hierarchy. Though it does not increase the coupling between function-only classes, this approach enhances the cohesion within a function-only class by grouping subroutines that call one another. By choosing an appropriate value for the class-depth parameter, as in *MaxHeight/x*, the coupling within function-only classes can be reduced.

According to good software engineering practices [28], low coupling and high cohesion is desirable in any object-oriented design. An appropriate value for the class-depth parameter *x* can be chosen such that the clustering of the subroutines and functions results in an object-oriented code with an acceptable level of cohesion and coupling. Using the metrics suggested by Chidamber and Kemerer [31] and Lee *et. al.* [32] to determine cohesion and coupling, the Aerox case study (Section 5.3) demonstrates the effect of varying the class-depth factor, *x*, on the resulting cohesion and coupling of the converted object-oriented code.

6.2. Contributions

This thesis has addressed the need for automating the conversion of structured FORTRAN77 code into object-oriented C++ code. In particular, it has demonstrated a methodology for full automation of the analysis of the inherent characteristics of the FORTRAN77 code, based on its subroutine- and function-calling hierarchy and common block structure, to produce a robust, consistent, and fully functional first-cut object-oriented design and associated C++ code implementation. The benefits of this work are as follows:

(1) Reduces conversion time: The automation enables a robust, first-cut conversion to be completed in hours instead of months. In particular, through two case studies, it demonstrated a reduction in conversion time by 99.16% (240 hours vs. 2 hour) and 99.4% (160 hours vs. 1 hour) for the Weights and Control modules, respectively, by employing automation instead of manual conversion.

(2) Promotes coding consistency and reduces the introduction of new errors: The automation removes human variability, and associated introduction of new errors, by enforcing a consistent naming and coding conventions. This speeds up the subsequent verification of the converted code, and it facilitates subsequent maintenance and evolution of this code.

(3) Preserves inherent designer's intent: The clustering of classes based on the common blocks and the subroutine- and function-calling hierarchy

preserves parts of the original designer's intent that is implicitly embedded in these structures. This designer's intent often becomes more clear when comparing the UML class diagrams produced as the class-depth factor, *x*, is varied. Indeed, it was found that these UML class diagrams provided new insights into the software system structure that had been lost to the original system designers.

(4) Preserves compatibility with non-converted code: The use of class structures and their member functions is designed such that converted code may co-exist, operate, and compile with legacy modules that have not yet been converted. This enables a module-by-module conversion process without requiring that all code modules be converted at once, and it facilitates the integration with legacy modules that will most likely never be converted.

6.3. Recommendations for Future Work

This thesis provides an important contribution towards maintaining legacy codes that have been undergoing significant changes over time. However, the methodology presented contains two serious issues that should be addressed in future work. The first is that there is a high degree of interaction between classes when function-only classes interact with data-only classes. From an object-oriented point of view, with an emphasis on ease of maintenance, this high level of coupling is clearly undesirable.

The second issue is that the FORTRAN77 global data, in the form of common blocks, are placed into classes with the global data as member variables with public access. Since any class can access a class's public members, such access might produce undesirable results should an outsider use them. Indeed, these classes contain only data and no methods other than a method to instantiate its Singleton class (Appendix A). From an object-oriented point of view, with an emphasis on maintenance robustness, these data-only classes are clearly undesirable.

One approach to reduce the interaction between the function-only classes and the data-only classes, and the presence of data-only classes, is to use data flow analysis. This

would facilitate combining at least parts of these data-only classes with the appropriate, corresponding function-only classes to more effectively support the ongoing software maintenance process. The FORTRAN77 source code can be analyzed to collect statistics about the subroutines and functions and their use of the global data. For instance, matrices can be generated to describe the number of times a subroutine or function alters or observes the data in a common block, perhaps with the resolution of the individual data instead of the common block itself. Once the statistics have been gathered, the global data might be placed in the already identified function-only classes, for instance according to the following rules [16]:

1. The common block should be placed in the function class that alters it the most number of times.
2. If more than one function class alters a common block the same maximum number of times, then the common block should be placed in the function class (among those having the same number of maximum alterations) where it is being observed the most number of times.
3. If more than one function class alters and observes a common block the same number of times, then the common block should be placed in any of these function classes on a random basis.

Then, after identifying the parent class in which a common block will be placed, all the other classes that use this common block can refer to the common block through a reference to that parent class. Hence, the data within common blocks is kept together within a single class. The rationale behind not splitting the common blocks is that a common block is often used in FORTRAN77 code as a placeholder for related global variables. Splitting a common block across multiple classes would undo this frequent implicit relationship. Hence, it would generally not be desirable, from an object-oriented point of view, to split a common block across multiple classes. Adding these capabilities would further enhance the quality of the object-oriented code generated by *F77toCpp*. It would increase the cohesion, reduce the coupling, and bring out more of the implicit designer's intent to help make the code more maintainable over time.

REFERENCES

[1] G. N. Vanderplaats, "ACSYNT Program Overview," *Proc., NASA Ames Computer-Aided Design Workshop*, NASA Ames Research Center, Moffett Field, CA, January 1975.

[2] J. D. Arthur, R. E. Nance, and B. J. Keller, "The Paradigm Selection Guide: A Product of Collaborative Research Between NSWCDD SLBM Personnel and VTSRC Personnel," Technical Report SRC-03-001, Systems Research Center, Virginia Tech, January 2003.

[3] G. V. Subramaniam and E. J. Byrne, "Deriving an Object Model from Legacy Fortran Code," *Proc., International Conference on Software Maintenance*, IEEE, Monterey, CA, November 4-8, 1996, pp. 3-12.

[4] J. R. Levine, T. Mason, and D. Brown, *lex & yacc*, 2nd edition, O'Reilly & Associates, 1992.

[5] J. Burkard, *fsplit*, source code, revised 12 July 2000, verified April 8, 2003, `http://www.psc.edu/~burkardt/src/fsplit/fspit.html`

[6] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, "A Fortran-to-C Converter," *Computing Science Technical Report No. 149*, AT&T Bell Laboratories, Murray Hill, NJ 07974, May 16, 1990 (revised March 22, 1995). `http://achille.cs.bell-labs.com/cm/cs/cstr/149.ps.gz`

[7] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer, *f2c*, source code, June 25, 2002, `http://www.netlib.org/f2c/`

[8] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley, New York, 1995, pp. 127-134.

[9] S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler," in: *Unix Programmer's Manual*, Volume II, Holt, Rinehart, and Wilson, AT&T Bell Laboratories, Murray Hill, NJ, 1983.

[10] S. I. Feldman and P. J. Weinberger, "A Portable Fortran 77 Compiler," in: *Unix Time Sharing System Programmer's Manual*, Volume 2, AT&T Bell Laboratories, 10th edition, Saunders College Publishing, Murray Hill, NJ, 1990, ISBN 0-03-047529-5.

[11] *g77*, part of the GNU Compiler Collection (GCC), the GNU Project, `http://www.gnu.org/software/gcc/gcc.html`

[12] C. Laffra, "Translating C++ to Java," in: *Advanced Java*, Prentice Hall, Inc., New Jersey, 1997, pp. 253-262.

[13] Novosoft, *C2J-C to Java translator*, November 2001, `http://tech.novosoft-us.com/product_c2j.jsp`

[14] P. R. Salopek, T. J. Timcho, and W. V. Barnishan, "Migration of Legacy Test Programs to Modern Programming Environments," *Proc., AUTOTESTCON*, IEEE, Anaheim, CA, September 18-21, 2000, pp. 217-222.

[15] H. M. Sneed, "Object Oriented COBOL Recycling," *Proc., 3rd Working Conference on Reverse Engineering*, IEEE, Monterrey, CA, November 8-10, 1996, pp. 169-178.

[16] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino, "Identifying objects in legacy systems," *Proc., 5th International Workshop on Program Comprehension*, IWPC'97, IEEE, Dearborn, MI, May 28-30, 1997, pp. 138-147.

[17] R. Millham, "An investigation: reengineering sequential procedure-driven software into object-oriented event-driven software through UML diagrams," *Proc., 26th Annual International Conference on Computer Software and Applications,* IEEE, Oxford, England, August 26-29, 2002, pp. 731-733.

[18] S. S. Liu and N. Wilde, "Identifying Objects in a conventional Procedural Language: An Example of Data Design Recovery," *Proc., Conference on Software Maintenance*, San Diego, CA, November 26-29, 1990, pp. 266-271.

[19] A. Cimitile, A. De Lucia, G. A. Di Lucca, and A. R. Fasolino, "Identifying objects in legacy systems using design metrics," *Journal of Systems and Software*, vol. 44, no. 3, January, 1999, pp. 199-211.

[20] A. Van Deursen and T. Kuipers, "Identifying Objects using Cluster and Concept Analysis," *Proc., International Conference on Software Maintenance*, Los Angeles, CA, May 16-22, 1999, pp. 246-255.

[21] H. Sahraoui, P. Valtchev, I. Konkobo, and S. Shen, "Object identification in legacy code as a grouping problem," *Proc., 26th Annual International Conference on Computer Software and Applications,* IEEE, Oxford, England, August 26-29, 2002, pp. 689-696.

[22] B. L. Achee and D. L. Carver, "A Greedy Approach to Object Identification in Imperative Code," *Proc., 3rd Workshop on Program Comprehension*, IEEE, Washington, D.C., November 14-15, 1994, pp. 4-11.

[23] R. Fanta and V. Rajlich, "Restructuring legacy C code into C++," *Proc., International Conference on Software Maintenance*, Oxford, England, IEEE Computer Society Press, August 30-September 3,1999, pp. 77-85.

[24] R. D. Penteado, P. C. Masiero, A. F. Do Prado, and R. T. V. Braga, "Reengineering of Legacy Systems based on Transformations Using the Object Oriented Paradigm," *Proc., Fifth Working Conference on Reverse Engineering*, Honolulu, HI, October 12-14, 1998, pp. 144-153.

[25] R. D. Penteado, F. Germano, P. C. Masiero, "An Overall Process Based on Fusion to Reverse Engineer Legacy Code," *Proc., 3rd Working Conference on Reverse Engineering*, IEEE, Monterey, CA, November 8-10, 1996, pp. 179-188.

[26] A. F. Prado, "Strategy of Reengineering of Domain Oriented Software," Sc.D Thesis, Rio de Janeiro/RJ, Brazil, 1992.

[27] S. C. Choi and W. Scacchi, "Extracting and Restructuring the Design of Large Systems," *IEEE Software*, vol. 7, no. 1, January 1990, pp. 66-71.

[28] J. Krauskopf, "Elemental concerns [Software Design]," *IEEE Potentials*, vol. 9, no. 1, February 1990, pp. 13-15.

[29] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, 1974, pp. 115-139.

[30] A. V. Dandekar, "A Procedural Approach to the Evaluation of Software Development Methodologies," Master's thesis, Department of Computer Science, Blacksburg, VA, September 1987.

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

[32] Y. S. Lee, B. S. Liang, S. F. Wu, and F. J. Wang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," *Proc., International Conference on Software Quality*, Maribor, Slovenia, November 6-9, 1995, pp. 81-90.

[33] D. Kalev, *The ANSI/ISO C++ Professional Programmer's Handbook*, QUE Professional, 1999.

[34] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*, Addison Wesley, New York, 1995, pp. 315-323.

[35] Tennessee Technological University, Computer Aided Engineering Laboratory, October 21, 2003, "Mixing Code in C, C++, and FORTRAN on Unix," `http://www.cae.tntech.edu/help/programming/mixed_languages/view`

APPENDIX A. Singleton class for common block access

The C++ Singleton design pattern ensures that a class has only one instance, and it provides a global point of access to this instance. It works by having a special method that is used to instantiate the desired object: When this method is called, it checks to see if the object has already been instantiated. If it has, the method simply returns a reference to the object. If not, the method instantiates it and returns a reference to this new instance. To ensure that this is the only way to instantiate an object of this type, the constructor of this class is made either protected or private.

In the following example, all three FORTRAN77 program-units (`MAIN`, `SUB1`, and `SUB2`) refer the same memory representing the common block SHARED (lines 103, 113, and 122):

FORTRAN77 FILE: `main.f`

```
101           PROGRAM MAIN
102
103           COMMON /SHARED/ I
104
105           CALL SUB1
106           WRITE(*,*) I
107           CALL SUB2
108           WRITE(*,*) I
109
110           END
```

FORTRAN77 FILE: `test.f`

```
111           SUBROUTINE SUB1
112
113           COMMON /SHARED/ I
114
115           I = 10
116
117           END
118
119
120           SUBROUTINE SUB2
121
122           COMMON /SHARED/ I
123
124           I = 20
125
126           END
```

The following C++ code corresponds to the above FORTRAN77 code, including the common block `SHARED`. Note how, in the files `CShared.h` and `CShared.cpp`, the constructor for this common block is made `private` (line 207 and 218-221) so that external classes are denied permission to create an instance of the `CShared` object (line 206). Instead, it is the responsibility of the `getinstance` method (lines 210 and 223-228) to create an instance of the `CShared` object if it has not yet been created, or, to return the existing instance. Hence, there is never more than one instance of the `CShared` object. Finally, note how the flag `instance` is static (line 206) so it can be initialized at compile time (line 216):

(1-A) C++ HEADER FILE: `CShared.h`

```
201    #ifndef CSHARED_INCLUDED
202    #define CSHARED_INCLUDED
203
204    class CShared {
205           private:
206                  static CShared* instance;
207                  CShared();
208           public:
209                  int i;
210                  static CShared* getinstance();
211    };
212
213    #endif
```

(1-B) C++ CLASS FILE: `CShared.cpp`

```
214    #include "CShared.h"
215
216    CShared* CShared::instance = 0;    // initialize pointer
217
218    CShared::CShared()
219    {
220           // perform necessary instance initializations
221    }
222
223    CShared* CShared::getinstance()
224    {
225           if (instance == 0)
226                  instance = new CShared();
227           return instance;
228    }
```

The remaining files replicate the above sample FORTRAN77 program units, including their access of the `CShared` object which functions like a FORTRAN77 common block:

(2-A) C++ HEADER FILE: `CSub1.h`

```
#ifndef sub1_included
#define sub1_included

#include "CShared.h"

class CSub1
{
       public:
              CShared* pcommon;
              CSub1();
              CSub1(CShared* acommon);
};

#endif
```

(2-B) C++ CLASS FILE: `CSub1.cpp`

```
#include "CSub1.h"

CSub1::CSub1()
{
}

CSub1::CSub1(CShared* acommon)
{
       pcommon = acommon;
       pcommon->i = 10;
}
```

(3-B) C++ HEADER FILE: `CSub2.h`

```
#ifndef sub2_included
#define sub2_included

#include "CShared.h"

class CSub2
{
       public:
              CShared* pcommon;
              CSub2();
              CSub2(CShared* bcommon);
};

#endif
```

(3-A) C++ CLASS FILE: `CSub2.cpp`

```
268    #include "CSub2.h"
269
270    CSub2::CSub2()
271    {
272    }
273
274    CSub2::CSub2(CShared* acommon)
275    {
276           pcommon = acommon;
277           pcommon->i = 20;
278    }
```

MAIN C++ FILE: `main.cpp`

```
279    #include "CSub1.h"
280    #include "CSub2.h"
281    #include "CShared.h"
282    #include <iostream.h>
283    using namespace std;
284
285    int main()
286    {
287           CShared* acommon = CShared::getinstance();
288           CSub1 asub1(CShared::getinstance());
289           cout << acommon->i << endl;
290           CSub2 asub2(CShared::getinstance());
291           cout << acommon->i << endl;
292           return 0;
293    }
```

Note how an object of class `CShared` is instantiated automatically when referenced via the call to `getinstance` (line 287). Subsequent call to `getinstance` (lines 288 and 290) do not instantiate additional `CShared` "common blocks" objects, but merely return the pointer to the previously instantiated object. The two output statements demonstrate this sharing of common data space: Line 289 will print "`10`" because the `CShared` "common block" variable `i` is set to `10` by the `CSub1` constructor (line 252), whereas line 291 will print "`20`" because `CShared` "common block" variable `i` is set to `20` by the `CSub2` constructor (line 277).

APPENDIX B. Inserting *f2c*-generated code into the class files

F77toCpp first parses the FORTRAN77 source code to identify the various classes, their attributes, and member functions. From this, it generates the corresponding skeleton classes, but with empty member function bodies (lines 201-208). The program *fsplit* extracts the subroutines and functions from the original FORTRAN77 source file and saves them as individual FORTRAN77 files with the same name as that of the respective program unit, with the *.f* file extension (lines 101-140). Splitting these program units into individual files greatly simplifies subsequent processing. The program *f2c* can now convert each program unit into a separate *.c* file, which then can easily be inserted into the appropriate *.cpp* class file. The same is true for the *.f* file that is inserted into this same *.cpp* file as a comment. A script performs these file insertions. The result is a C++ class file complete with functional C++ code and the original FORTRAN77 source code included as a comment for reference (lines 301-370).

The following example illustrates this process. It shows the source code for the FORTRAN77 subroutine, `SUB1`, after it has been extracted from the original FORTRAN77 source code by the program *fsplit*:

```
101           SUBROUTINE SUB1
102
103           COMMON /SKINC/ TT, RNOL, PT, IS, JS
104
105           TT = 100
106           RNOL = 2000
107           MAX = 92
108           N = 88
109           DETERM = 89.9
110
111           CALL MATINV(MAX,N,DETERM)
112
113           RETURN
114           END
115
116
117           SUBROUTINE MATINV(MAX,N,DETERM)
118
119           COMMON /SKINC/ TT,RNOL,PT,IS,JS
120
121           DIMENSION IWK(10,2)
122
123           EQUIVALENCE (IROW,IS), (ICOLUM,JS)
124
125 C INITIALIZATION
126
127 C SEARCH FOR PIVOT ELEMENT
```

```
128           AMAX=0.0
129
130           IROW=J
131           ICOLUM=K
132           AMAX=TMAX
133
134   C INTERCHANGE ROWS TO PUT PIVOT ELEMENT ON DIAGONAL
135
136           IWK(I,1)=IROW
137           IWK(I,2)=ICOLUM
138
139           RETURN
140           END
```

Continuing this example, the following shows the `CSub1` class file that was generated by *F77toCpp* to eventually represent the `SUB1` subroutine. Note how it contains indicators that show where to insert the original FORTRAN77 code that was generated by *fsplit* in the form of a C/C++ comment, and where to insert the corresponding C++ code that was generated by *f2c*:

```
201     //F77CPP_INSERT_FORTRAN    sub1.f
202     //F77CPP_INSERT_C          sub1.c
203
204     CSub1::CSub1()
205     {
206           pMatinv = new CMatinv();
207           pSkinc = CSkinc::getinstance();
208     }
```

Next, the FORTRAN77 source code and the corresponding C++ source code are inserted into this class file. The result is shown here:

```
301     #include <iostream.h>
302     using namespace std;
303     #include "CSub1.h"
304
305     //COMMENTED FORTRAN SOURCE CODE
306     /*
307     #if 0  // BEGIN INSERT OF ORIGINAL FORTRAN CODE: sub1.f
308
309           SUBROUTINE SUB1
310
311           COMMON /SKINC/ TT, RNOL, PT, IS, JS
312
313           TT = 100
314           RNOL = 2000
315           MAX = 92
316           N = 88
317           DETERM = 89.9
318
319           CALL MATINV(MAX,N,DETERM)
320
321           RETURN
```

```
322           END
323     #endif // END INSERT OF ORIGINAL FORTRAN CODE: sub1.f
324     */
325
326     #if 0  // BEGIN INSERT OF ORIGINAL f2c CODE: sub1.c
327     /* ../F77CPP_FORTRAN/sub1.f -- translated by f2c (version 20020621).
328        You must link the resulting object file with the libraries:
329           -lf2c -lm   (in that order)
330     */
331
332     #ifdef __cplusplus
333     extern "C" {
334     #endif
335     #include "f2c.h"
336
337     /* Common Block Declarations */
338
339     struct {
340         real tt, rnol, pt;
341         integer is, js;
342     } skinc_;
343
344     #define skinc_1 skinc_
345
346     /* Subroutine */ int sub1_()
347     {
348         integer n, max__;
349         real determ;
350         extern /* Subroutine */ int matinv_(integer *, integer *, real *);
351
352         skinc_1.tt = (float)100.;
353         skinc_1.rnol = (float)2e3;
354         max__ = 92;
355         n = 88;
356         determ = (float)89.9;
357         matinv_(&max__, &n, &determ);
358         return 0;
359     } /* sub1_ */
360
361     #ifdef __cplusplus
362     }
363     #endif
364     #endif // END INSERT OF ORIGINAL f2c CODE: sub1.c
365
366     CSub1::CSub1()
367     {
368             pMatinv = new CMatinv();
369             pSkinc = CSkinc::getinstance();
370     }
```

At this point, the C++ code will not compile properly. A number of automated scripted modifications must be applied to complete the conversion process. Appendices C through F illustrate these modifications, with the final result being shown in Appendix G.

APPENDIX C. Replacing EQUIVALENCE statements

When a local variable is made equivalent to a common block variable, the program *f2c* defines the local variable via a compiler directive such that it maps to the address of that particular common block variable within the `struct` data structure that both *f2c* and *g77* use to represent that common block in C. Since *F77toCpp* converts this `struct` data structure into a `class` data structure in C++, this compiler directive must be modified accordingly. A script is used to perform these modifications. This modification contains two elements. The first is to change the reference from a `struct` to a `class` (Appendix D illustrates this change in general). The second is to change from *f2c*'s numeric index into the `struct` data structure to address the target variable, to addressing the variable using its name. While the latter is not strictly required, it makes the code easier to read and maintain.

The following C code was generated by the program *f2c*. In the original FORTRAN77 code, local variable `IROW` is made equivalent to the variable `IS`, the fourth variable within the common block `SKINC`. As can be observed below, *f2c* translates this equivalence by making a compiler directive named `irow` (line 114) that points to the memory location corresponding to the variable `is` in the common block `skinc` (line 103). Hence, the syntax of this compiler directive (line 114) must be updated when *F77toCpp* changes the `struct` data structures into `class` data structures.

```
101     struct {
102         real tt, rnol, pt;
103         integer is, js;
104     } skinc_;
105
106     #define skinc_1 skinc_
107
108     /* Subroutine */ int matinv_(integer *max__, integer *n, real *determ)
109     {
110         /* Local variables */
111         integer i__, j, k, iwk[20]    /* was [10][2] */;
112         real amax, tmax;
113
114     #define irow ((integer *)&skinc_1 + 3)
115     #define icolum ((integer *)&skinc_1 + 4)
116
117     #define iwk_ref(a_1,a_2) iwk[(a_2)*10 + a_1 - 11]
118
119         amax = (float)0.;
120         *irow = j;
```

```
121         *icolum = k;
122         amax = tmax;
123
124     /*       INTERCHANGE ROWS TO PUT PIVOT ELEMENT ON DIAGONAL */
125
126         iwk_ref(i__, 1) = *irow;
127         iwk_ref(i__, 2) = *icolum;
128         return 0;
129     } /* matinv_ */
130
131     #undef iwk_ref
132     #undef icolum
133     #undef irow
```

Continuing this example, the `struct` data structure (lines 101-104) has now been replaced with a `class` data structure (line 251 and 204-215). Consequently, the compiler directives (lines 114-115) have been changed to reflect this new data structure (lines 259-260). In particular, note how the directives no longer references the variables via a numeric offsets, but instead use the class variable names to facilitate human interpretation.

C++ HEADER FILE: `CSkinc.h`

```
201     #ifndef CSkinc_H_INCLUDED
202     #define CSkinc_H_INCLUDED
203
204     class CSkinc {
205           private:
206                 static CSkinc* instance;
207                 CSkinc();
208           public:
209                 static CSkinc* getinstance();
210                 float tt;
211                 float rnol;
212                 float pt;
213                 long int is;
214                 long int js;
215     };
216     #endif
```

C++ CLASS FILE: `CSkinc.cpp`

```
251     #include "CSkinc.h"
252
253     /* Subroutine */ int matinv_(integer *max__, integer *n, real *determ)
254     {
255         /* Local variables */
256         integer i__, j, k, iwk[20]    /* was [10][2] */;
257         real amax, tmax;
258
259     #define irow ((integer *)&pSkinc->is)
260     #define icolum ((integer *)&pSkinc->js)
```

```
261
262     #define iwk_ref(a_1,a_2) iwk[(a_2)*10 + a_1 - 11]
263
264         amax = (float)0.;
265         *irow = j;
266         *icolum = k;
267         amax = tmax;
268
269     /*       INTERCHANGE ROWS TO PUT PIVOT ELEMENT ON DIAGONAL */
270
271         iwk_ref(i__, 1) = *irow;
272         iwk_ref(i__, 2) = *icolum;
273         return 0;
274     } /* matinv_ */
275
276     #undef iwk_ref
277     #undef icolum
278     #undef irow
```

APPENDIX D. Replacing references to structures with references to classes

The program *f2c* represents FORTRAN77 common blocks using `struct` data structures in C (lines 101-104). *F77toCpp* converts these data structures to `class` data structures in C++. It is therefore necessary for *F77toCpp* to replace the references to the `struct` structures that were generated by *f2c* with corresponding references to the `class` structures. The following example shows code that was generated by *f2c*. Note the syntax for referencing the `skinc` common block variables `tt` and `rnol` (lines 114-115):

```
101    struct {
102        real tt, rnol, pt;
103        integer is, js;
104    } skinc_;
105
106    #define skinc_1 skinc_
107
108    int sub1_()
109    {
110        integer n, max__;
111        real determ;
112        extern /* Subroutine */ int matinv_(integer *, integer *, real *);
113
114        skinc_1.tt = (float)100.;
115        skinc_1.rnol = (float)2e3;
116        max__ = 92;
117        n = 88;
118        determ = (float)89.9;
119
120        matinv_(&max__, &n, &determ);
121        return 0;
122    }
```

Continuing this example, the `struct` data structure (lines 101-104) has now been replaced with a `class` data structure (line 251 and 204-215). Consequently, the references to the common block variables (lines 114-115) have been changed to reflect this new data structure (lines 258-259). In particular, note how `pSkinc` is a pointer to an object of class `CSkinc` representing the common block `skinc`.

C++ HEADER FILE: `CSkinc.h`

```
201    #ifndef CSkinc_H_INCLUDED
202    #define CSkinc_H_INCLUDED
203
204    class CSkinc {
205           private:
206                  static CSkinc* instance;
207                  CSkinc();
208           public:
209                  static CSkinc* getinstance();
210                  float tt;
211                  float rnol;
212                  float pt;
213                  long int is;
214                  long int js;
215    };
216    #endif
```

C++ CLASS FILE: `CSub1.cpp`

```
251    #include "CSkinc.h"
252
253    /* F77CPP: SUBROUTINE */ int CSub1::sub1(){
254        integer n, max__;
255        real determ;
256        extern /* Subroutine */ int matinv_(integer *, integer *, real *);
257
258        pSkinc->tt = (float)100.;
259        pSkinc->rnol = (float)2e3;
260        max__ = 92;
261        n = 88;
262        determ = (float)89.9;
263        matinv_(&max__, &n, &determ);
264        return 0;
265    }
```

APPENDIX E. Replacing function calls with calls to member functions in appropriate classes

The program *f2c* generates C-style function calls. However, once *F77toCpp* converts these functions to class-member functions, then their corresponding invocations must be changed as well. The following code segment shows a function invocation as generated by *f2c* (line 113):

```
101    int CSub1::sub1(){
102        integer n, max__;
103        real determ;
104        extern /* Subroutine */ int matinv_(integer *, integer *, real *);
105
106        pSkinc->tt = (float)100.;
107        pSkinc->rnol = (float)2e3;
108
109        max__ = 92;
110        n = 88;
111        determ = (float)89.9;
112
113        matinv_(&max__, &n, &determ);
114
115        return 0;
116    }
```

The following is the code after replacing the above C-style function invocation (line 113) with a C++ class-member function invocation (line 212). Note how it is no longer necessary to define the subroutine `matinv` as external (line 104) since it is now a class-member function.

```
201    int CSub1::sub1(){
202        integer n, max__;
203        real determ;
204
205        pSkinc->tt = (float)100.;
206        pSkinc->rnol = (float)2e3;
207
208        max__ = 92;
209        n = 88;
210        determ = (float)89.9;
211
212        pMatinv->matinv(&max__, &n, &determ);
213
214        return 0;
215    } /* sub1_ */
```

APPENDIX F. Copying function signatures into header files

When *F77toCpp* generates the skeleton class files based on the FORTRAN77 calling hierarchy, it does not yet know what the function signatures of the class members will look like. This information will only be revealed once the program *f2c* has completed its translation of the source code from FORTRAN77 into C/C++. Hence, the skeleton *.cpp* and *.h* files are generated without these function signatures (lines 101-109).

```
101    class CSub1 { //Parents: root
102        public:
103            //COMMON BLOCK:
104            CSkinc* pSkinc;
105            //CHILD CLASS:
106            CMatinv* pMatinv;
107
108            CSub1();
109    };
```

In the case shown here, it is therefore necessary to add the function signature for the member function `sub1` of the class `CSub1` to the C++ header file `CSub1.h` once its C/C++ code has been generated by *f2c* and inserted into the C++ class file `CSub1.cpp`. A script performs this addition, and its result is shown in line 209.

```
201    class CSub1 { //Parents: root
202        public:
203            //COMMON BLOCK:
204            CSkinc* pSkinc;
205            //CHILD CLASS:
206            CMatinv* pMatinv;
207
208            CSub1();
209            int sub1();
210    };
```

APPENDIX G. Difference between FORTRAN77 source code and the C++ code generated by *F77toCpp*

This appendix illustrates the complete conversion of a FORTRAN77 source code into the corresponding C++ code as generated by *F77toCpp*. The following FORTRAN77 subroutine is extracted from the example shown in Appendix B:

```
101           SUBROUTINE SUB1
102
103           COMMON /SKINC/ TT, RNOL, PT, IS, JS
104
105           TT = 100
106           RNOL = 2000
107           MAX = 92
108           N = 88
109           DETERM = 89.9
110
111           CALL MATINV (MAX, N, DETERM)
112
113           RETURN
114           END
```

The following codes represent the final result of using *F77toCpp* to convert the above FORTRAN77 code into an object-oriented C++ code, including the use of scripts to adapt the *f2c*-generated C code to a C++ environment. Note how the class `CSub1` (lines 301-320) is used to represent the `SUB1` subroutine (lines 101-114). This class references the class `CSkinc` (line 313), which represents the FORTRAN77 common block `SKINC` (line 103); and it references the class `CMatinv` (line 315), which represents the FORTRAN77 subroutine `MATINV` that is called on lines 111 (FORTRAN77 version) and 390 (C++ version).

C++ HEADER FILE: `CSkinc.h`

```
201   #ifndef CSkinc_H_INCLUDED
202   #define CSkinc_H_INCLUDED
203
204   class CSkinc {
205
206       private:
207
208           static CSkinc* instance;
209
210           CSkinc();
211
212       public:
213
```

```
214             static CSkinc* getinstance();
215             float tt;
216             float rnol;
217             float pt;
218             long int is;
219             long int js;
220
221     };
222     #endif
```

C++ HEADER FILE: CSkinc.C

```
251     #include <stdio.h>
252     #include "CSkinc.h"
253
254     CSkinc* CSkinc::instance = NULL;
255
256     CSkinc::CSkinc() {
257
258     }
259
260     CSkinc* CSkinc::getinstance() {
261
262             if (instance == NULL)
263                instance = new CSkinc();
264             return instance;
265     }
```

C++ HEADER FILE: CSub1.h

```
301     #ifndef CSub1_H_INCLUDED
302     #define CSub1_H_INCLUDED
303
304     #include <f2c.h>
305
306     #include "CMatinv.h"
307     #include "CSkinc.h"
308
309     class CSub1 { //Parents: root
310
311         public:
312             //COMMON BLOCK:
313             CSkinc* pSkinc;
314             //CHILD CLASS:
315             CMatinv* pMatinv;
316
317             CSub1();
318             int sub1();
319     };
320     #endif
```

C++ CLASS FILE: CSub1.C

```
351     #include <iostream.h>
352     using namespace std;
353     #include "CSub1.h"
354
355
356     /*
```

```
#if 0  // BEGIN INSERT OF ORIGINAL FORTRAN CODE: sub1.f

      SUBROUTINE SUB1

      COMMON /SKINC/ TT, RNOL, PT, IS, JS

      TT = 100
      RNOL = 2000
      MAX = 92
      N = 88
      DETERM = 89.9

      CALL MATINV (MAX, N, DETERM)

      RETURN
      END
#endif // END INSERT OF ORIGINAL FORTRAN CODE: sub1.f
*/
/* ../F77CPP_FORTRAN/sub1.f -- translated by f2c (version 20020621).
   You must link the resulting object file with the libraries:
        -lf2c -lm   (in that order)
*/


/* F77CPP: SUBROUTINE */ int CSub1::sub1(){
    long int n, max__;
    float determ;

    pSkinc->tt = (float)100.;
    pSkinc->rnol = (float)2e3;
    max__ = 92;
    n = 88;
    determ = (float)89.9;
    pMatinv->matinv(&max__, &n, &determ);
    return 0;
} /* sub1_ */


CSub1::CSub1()
{
    pMatinv = new CMatinv();
    pSkinc = CSkinc::getinstance();
}
```

The remaining code describes the MATINV subroutine, as converted by *F77toCpp* from FORTRAN77 into C++. It is included here for reference purpose, as it was referenced by the above code:

C++ HEADER FILE: `CMatinv.h`

```
401    #ifndef CMatinv_H_INCLUDED
402    #define CMatinv_H_INCLUDED
403
404    #include <f2c.h>
405
406    #include "CSkinc.h"
407
408    class CMatinv { //Parents: SUB1
409
410        public:
411            //COMMON BLOCK:
412            CSkinc* pSkinc;
413
414            CMatinv();
415            int matinv(long int *max__, long int *n, float *determ);
416    };
417    #endif
```

C++ HEADER FILE: `CMatinv.C`

```
451    #include <iostream.h>
452    using namespace std;
453    #include "CMatinv.h"
454
455
456    /*
457    #if 0  // BEGIN INSERT OF ORIGINAL FORTRAN CODE: matinv.f
458
459
460
461          SUBROUTINE MATINV(MAX,N,DETERM)
462
463          COMMON /SKINC/ TT,RNOL,PT,IS,JS
464
465          DIMENSION IWK(10,2)
466
467          EQUIVALENCE (IROW,IS), (ICOLUM,JS)
468
469    C INITIALIZATION
470
471    C SEARCH FOR PIVOT ELEMENT
472            AMAX=0.0
473
474
475             IROW=J
476               ICOLUM=K
477               AMAX=TMAX
478
479    C       INTERCHANGE ROWS TO PUT PIVOT ELEMENT ON DIAGONAL
480
481          IWK(I,1)=IROW
```

```
482           IWK(I,2)=ICOLUM
483
484           RETURN
485           END
486     #endif // END INSERT OF ORIGINAL FORTRAN CODE: matinv.f
487     */
488     /* ../F77CPP_FORTRAN/matinv.f -- translated by f2c (version 20020621).
489        You must link the resulting object file with the libraries:
490             -lf2c -lm   (in that order)
491     */
492
493
494     /* F77CPP: SUBROUTINE */ int CMatinv::matinv(long int *max__, long int
*n, float *determ){
495          /* Local variables */
496          long int i__, j, k, iwk[20] /* was [10][2] */;
497          float amax, tmax;
498     #define irow ((long int *)&pSkinc->+ 3)
499     #define icolum ((long int *)&pSkinc->+ 4)
500
501
502     #define iwk_ref(a_1,a_2) iwk[(a_2)*10 + a_1 - 11]
503
504     /* INITIALIZATION */
505     /* SEARCH FOR PIVOT ELEMENT */
506          amax = (float)0.;
507          *irow = j;
508          *icolum = k;
509          amax = tmax;
510     /*       INTERCHANGE ROWS TO PUT PIVOT ELEMENT ON DIAGONAL */
511          iwk_ref(i__, 1) = *irow;
512          iwk_ref(i__, 2) = *icolum;
513          return 0;
514     } /* matinv_ */
515
516     #undef iwk_ref
517     #undef icolum
518     #undef irow
519
520
521
522     CMatinv::CMatinv()
523     {
524     }
```

Vita

Malini Kothapalli

Malini Kothapalli was born on January 16th 1979 in Nellore, India. She graduated from Regional Engineering College, Warangal in 1999 with a degree in Electronics & Communications Engineering. She received numerous scholarships including the scholarship for being in the top 5 in her class during undergraduate days. She then worked as a Software Engineer at Siemens Communication Systems, Bangalore and at Softalia Ltd., Hyderabad.

She is currently pursuing a Masters degree in Computer Science and Applications at Virginia Tech. During this academic phase, Malini worked as a research assistant and as a teaching assistant. Malini will begin a position as a Software Analyst with SWIFT, Manassas, VA starting June 2003.