REPORT DOCUMENTATION PAGE

AFRL-SR-BL-TR-01-0463

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
August 13, 2001	Final technical report	December 1998 - July 2001

4. TITLE AND SUBTITLE
Dynamic Analysis of Test and Evaluation Software

5a. CONTRACT NUMBER
F49620-99-1-0057

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Wilde, Norman

5d. PROJECT NUMBER

5e. TASK NUMBER
2304/TE

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of West Florida

Office of Research
Bldg.77, Room 131
11000 University Parkway
Pensacola, FL 32514-5751

Department of Computer Science
Bldg. 79, Room 116
11000 University Parkway
Pensacola, FL 32514-5751

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)
AFOSR/NM
801 N. Randolph St., Room 732
Arlington, VA 22203-1977

10. SPONSOR/MONITOR'S ACRONYM(S)
AFOSR

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

20011005 131

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Large legacy software systems are extremely difficult to keep up to date because they are so difficult to understand. Software Reconnaissance is a technique to aid in the understanding of such software by locating where each user "feature" is implemented. This project explored the application of Software Reconnaissance to the legacy FORTRAN 77 code that forms a large part of the Air Force Test and Evaluation software inventory. Two software tools, a FORTRAN 77 instrumentor and the TraceGraph visualization tool, were developed and are now publicly available on the internet. A case study was performed showing how the Reconnaissance could be used to aid in understanding and maintaining the CONVERT3 program which is typical of legacy FORTRAN code. A systematic methodology for reengineering such code into object-oriented C++ was then developed. The methodology prepares a domain class model and elaborates it by assigning features of the old program to the new object classes. Test cases are used both for Software Reconnaissance feature location and to validate the correctness of the new C++ version. A case study was performed showing how the methodology could be applied to CONVERT3, and providing several insights into this kind of reengineering task.

15. SUBJECT TERMS
program comprehension, dynamic analysis, software reengineering, FORTRAN, object-oriented programming

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Norman Wilde
Unclassified	Unclassified	Unclassified	UU	68	19b. TELEPHONE NUMBER *(include area code)* 850-474-2542

DYNAMIC ANALYSIS OF TEST AND EVALUATION SOFTWARE

Final Report to the

Air Force Office of Scientific Research (AFOSR)

Grant Number: F49620-99-1-0057

Norman Wilde*

Department of Computer Science

University of West Florida

Pensacola, FL 32514

August 13, 2001

* Electronic mail: nwilde@uwf.edu, URL: http://www.cs.uwf.edu/~wilde

EXECUTIVE SUMMARY

The Air Force has a large inventory of legacy FORTRAN software used to support Test and Evaluation as well as other missions. Software engineers need to be able to understand such software in order to maintain it or to reengineer it and thus keep it in step with changing mission requirements. Software Reconnaissance is a dynamic analysis methodology to aid in understanding old software by locating where specific user features are implemented. To use Reconnaissance, software is instrumented so that execution can be traced, and then test cases are executed with and without the desired feature. The resulting traces are analyzed to locate the feature in support of different software engineering tasks. Software Reconnaissance had previously been demonstrated successfully on fairly recent C programs. This project explores its application to legacy FORTRAN code typical of the Air Force Test and Evaluation inventory.

The project developed the following software engineering tools which have now been made available on our web site (http://www.cs.uwf.edu/~wilde/recon3/):

1. A FORTRAN 77 instrumentor, to facilitate tracing legacy FORTRAN code
2. The TraceGraph visualization tool, which facilitates using Reconnaissance interactively

A case study was carried out to illustrate the use of Software Reconnaissance on a typical example of legacy FORTRAN code from the Air Force inventory. The study showed that the method was effective in locating user features within a relatively small proportion of the code. The main difference with the earlier experiences using C was that the located code proved more difficult to understand, due to its lack of structure and to the presence of obsolete program plans.

The project then developed a methodology based on Software Reconnaissance for reengineering legacy FORTRAN code into object-oriented C++. A domain class model is developed and elaborated by assigning the different features of the FORTRAN program to the object classes. Software Reconnaissance test cases are used both for feature location and to validate the correctness of the resulting C++ program. A case study showed that the method seems to be a workable approach to reengineering legacy FORTRAN, though the tangled nature of this old code still requires much effort to understand.

In addition to the annual progress reports, the following intermediate technical reports have been submitted to AFOSR:

1. Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, "Feature Location in Legacy Fortran Code", submitted June 29, 2000.
2. Vaclav Rajlich, Kunrong Chen, Norman Wilde, Michelle Buckellew, Henry Page, "Software Evolution, Software Servicing, and Software Cultures", submitted November 13, 2000.
3. Norman Wilde, Michelle Buckellew, Vaclav Rajlich, "A Dynamic Analysis Methodology for Reengineering Fortran to C++", submitted May 25, 2001.

The following publications have resulted from this research (others are pending):

1. Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajlich, "A Case Study of Feature Location in Unstructured Legacy Fortran Code", *Proceedings 5th European Conference on Software Maintenance and Reengineering - CSMR 2001*, IEEE Computer Society Press, Los Alamitos, CA, March 2001, pp. 68-76.
2. Vaclav Rajlich, Norman Wilde, Michelle Buckellew, Henry Page, "Software Cultures and Evolution", to appear in *IEEE Computer.*

Participants in this research have included Michelle Buckellew, Vaclav Rajlich, Henry Page, LaTreva Pounds, Kazimiras Lukoit, Scott Stowell and Tim Hennessey.

TABLE OF CONTENTS

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I - INTRODUCTION

The Air Force Development Test Center has a large inventory of software vital for the test and evaluation (T&E) mission that needs to be continually enhanced and validated. Much of this software is old, coded mostly in FORTRAN, often poorly documented, and difficult to understand, maintain and support. For such programs to continue in service, methods and techniques are needed to facilitate understanding and reengineering into more modern paradigms such as those provided by object orientation.

To reengineer software, a programmer must understand not just "what" a code fragment does, but also "why". Old systems typically provide many different "features" to their end users. An optimization package, for example, may incorporate several different hill climbing strategies, handle several different classes of constraints, and provide many different input and output options. It can be very difficult for a programmer to understand which of these use cases a particular code fragment supports.

The University of West Florida has developed a method for dynamic analysis of software called Software Reconnaissance, which can help programmers understand the purpose of code. It provides a novel view of software that maps each program statement to the user features or use cases it supports. The mapping is constructed by running test cases that exhibit different combinations of features and tracing the program statements that are executed in each case. The traces are then analyzed using set theoretic methods to develop the mappings between use cases and code.

Software Reconnaissance was developed to work on C code and has been tested on systems at several companies including Bellcore, GTE and Northrop Grumman. The

Software Engineering Research Center (SERC), an NSF-supported industry university cooperative research center, has provided funding for these trials. Results have been favorable; programmers indicate that Software Reconnaissance provides very good starting points for locating code in large systems and can often provide insights that would be very hard to obtain by other means.

The purpose of this project was to see if Software Reconnaissance can be extended to the unstructured FORTRAN programs that are typical of the T & E code inventory. The project consisted of three parts:

1. Development of Software Reconnaissance tools for FORTRAN
2. Application of these tools to locate features in a sample of T&E FORTRAN software
3. Development and trial of a methodology based on Software Reconnaissance for reengineering FORTRAN into object-oriented C++

CHAPTER II - REVIEW OF THE LITERATURE

The literature on program comprehension is extensive. Koenemann and Robertson (1991) reported on an experiment they performed in which twelve expert programmers analyzed Pascal code and "thought aloud" to give the researchers an insight into their thought processes. The researchers concluded that programmers do not use a systematic strategy of comprehension, but generally proceed in a top-down fashion, ignoring documentation. They assert that good tools would be effective in aiding comprehension. Von Mayrhauser and Vans (1995) sought to analyze the cognitive processes of programmers as they went about the task of program comprehension. Their evaluation of cognition models concluded that most of these models do not apply to specialized maintenance tasks, and that more work is needed in this area. They combined several cognition models to create an "Integrated Metamodel." Robson, Bennett, Cornelius, and Munro (1991) examined approaches to program comprehension and concluded that inverse engineering techniques could lead to greater program understanding.

There is a lack of sufficient research on feature location, but several important papers have been written about this subject in the past decade. Biggerstaff, Mitbander, and Webster (1994) dealt with the process of recognizing concepts within a computer program to aid understanding. They suggested looking at descriptive data names, patterns of relationships between functions and data, and the use of tools to help in concept location. They determined that while a totally automated approach to feature location is not possible, a certain amount of automation is helpful. Lakhotia (1993) analyzed two programs, the GNU C compiler and the Wisconsin Program Integration Systems, and

discovered that one does not have to understand the entire program to make correct modifications. Chen and Rajlich (2000) outlined the problem of feature location. They advocated a Dependence Graph approach to feature location that involved a computer-assisted search process aiding the programmer in deciding whether a component is related to the feature being sought. Wilde and Scully (1995) introduced the Software Reconnaissance technique as a tool-based method of feature location. This technique is based on a comparison of traces of different test cases. The test cases are run "with" and "without" the feature being sought, then analyzed to determine which blocks or decisions are executed "with" the feature but not "without." An early test of the Software Reconnaissance technique found it to be effective in finding code at or near the feature being targeted (Wilde & Casey, 1996). This test also found it is important to use only a few simple test cases for effective feature location.

Although it is generally well known that legacy systems tend to become progressively more difficult to update as they age (Belady & Lehman, 1976; Kaliski & Kaliski, 1991), there has not been a large amount of research concerning the maintenance of legacy FORTRAN code. A tool for modeling large FORTRAN programs was described by Rajlich, Damaskinos, Linos, and Silva (1988). The VIFOR tool displays a graphical representation of the FORTRAN code in order to enhance understanding and to aid in building and modifying the program. Blazy and Facon (1993) described a technique and a tool supporting partial evaluation of FORTRAN programs. The tool produces a complete reduced program to aid in program comprehension. Blazy and Facon (1994) also presented a tool that facilitates the comprehension of large, complex general-purpose FORTRAN programs by specialization. Rugaber, Stirewalt, and Wills

(1995) described their experiences in detecting delocalized, overlapping fragments of code in a series of FORTRAN programs using analysis tools. They stated that such interleaving code compromises program comprehension, and that detecting this code and examining the individual fragments can improve understanding. An experience in restoring a legacy program, written partly in FORTRAN, was recounted by Rugaber and White (1998). The restoration team attempted to convert the FORTRAN code into C using a freeware tool set, but this resulted in unmanageable "spaghetti" code. Instead, they decided to increase the use of makefiles that would allow the languages to coexist.

Ripple analysis (Yau, Collofello, & MacGregor, 1978) and impact analysis (Queille, Voidrot, Wilde, & Munro, 1994) are names given to a collection of techniques used to identify how changes in one component of a system may affect other components. Arnold and Bohner (1996) provide a collection of papers covering different impact analysis approaches. The majority of this work looks at the impacts of a change in one code component on another code component. However, the most general kind of impact analysis also follows requirements traceability links between documentation and code. Thus, generalized impact analysis may be useful to locate features mentioned in the requirements document. Turver and Munro (1994), for example, have described a technique for modeling documentation entities and their connections to code in a Ripple Propagation Graph and for identifying the impact set from a change request. Obviously, this method will only work if requirements traceability information for the program has been carefully maintained, and this is fairly rare in practice.

Chikofsky and Cross (1990) attempted to clear up the confusion surrounding the terms "reengineering" and "reverse engineering." They define reengineering as "the

examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form." In contrast, reverse engineering is "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction." They emphasize that "reengineering generally includes some form of reverse engineering (to achieve a more abstract description) followed by some form of forward engineering or restructuring." Rugaber (1992) discussed why reverse engineering is difficult and outlined a methodology called Synchronized Refinement to help solve these problems. Synchronized Refinement analyzes a program and describes its behavior in the vocabulary of the application domain and its structure in terms of design decisions. A test of Synchronized Refinement showed that while it improved program comprehension and reverse engineering, it is extremely labor-intensive (Ornburn & Rugaber, 1992). DeBaud and Rugaber (1995) discussed a method for reengineering that uses an executable domain model to enhance understanding and an object-oriented framework to guide the reengineering. Their method seemed to speed the process, but the procedure of domain analysis and framework construction was time-consuming. Kozaczynski and Wilde (1992) described the problems inherent in reengineering transaction systems. They highlighted the difficulties caused by the necessity of an incremental approach and the problems caused by attempting to reengineer at the same time that the operational system continues to evolve. Wilde, Casey, Vandeville, Trio, and Hotz (1998) experimented with using Software Reconnaissance to aid in design recovery for a large multi-process system, JointSTARS. They found that using Software Reconnaissance on real-time systems was effective in recovering "design threads" from a

trace of inter-process messages. Linos et al. (1994) created a visual tool called CARE (Computer-Aided Re-Engineering) to facilitate the comprehension and reengineering of C programs. Their study on the use of CARE demonstrated that it reduced the average time needed to complete the maintenance task and helped their students make better-quality changes. Sneed and Majnar (1998) presented an attractive approach to reengineering which required "wrapping" the existing code, thus creating a large object component that a new object-oriented system can use. Wrapping has the advantage of requiring relatively little study of the legacy code, which is largely treated as a black box. However, there is still a considerable testing burden to make sure everything is working correctly and some of the benefits of recovering domain knowledge are lost since it is hidden inside the wrapped component. Given the wide range of situations that may be encountered even within a single system, Canfora, Cimitile, De Lucia, and Di Lucca (2001) suggested that tools and methods must depend on the specific system and advocated an eclectic approach in which a software engineer combines and tailors different methods for the problem at hand.

One study specifically addressed reengineering Department of Defense legacy code. Bergey, Smith, and Weiderman (1999) presented a series of guidelines to follow when creating a reengineering strategy. They argued that reengineering should be managed more rigorously than the development of new systems. Their guidelines provide a starting point for establishing the discipline necessary to reengineer legacy code. The guidelines are based in part on an earlier study that identified some of the key reasons reengineering of legacy systems often fails (Bergey, Smith, Tilley, Weiderman, & Woods, 1999).

Lately, interest has increased in the identification of objects in legacy procedural code as an aid to comprehension or reengineering (Canfora, Cimitile, & Munro, 1996; Cimitile, De Lucia, Di Lucca, & Fasolino, 1999; Di Lucca, Fasolino, & De Carlini, 2000; Liu & Wilde, 1990; Livadas & Johnson, 1994). Most of the methods described in these papers involve tools that cluster code components in some way to search for patterns of interaction that may indicate a candidate object. Lakhotia (1997) has provided a framework for describing and comparing such techniques. If a tool is used, a fairly large degree of human interaction with the tool is usually needed to help make sure that the identified objects are really meaningful. Jerding and Rugaber (1997) used a visualization tool called ISVis to help determine the major components of a system and the ways these components interact to accomplish the program's goals. In a case study, they found it to be successful in aiding architectural understanding. Subramaniam and Byrne (1996) identified a nine-step method for examining FORTRAN code to extract an object model. Achee and Carver (1994) developed an algorithm to identify objects in imperative code, specifically FORTRAN. They chose a bottom-up approach. Their algorithm evaluates the subroutines of the program to determine a set of objects, and examines the relationships among the parameters to construct the attribute sets of the objects. The main problem with this method is that it does not take into account the lack of modularity typical of legacy programs. Cimitile, Tortorella, and Munro (1994) experimented with identifying abstract data types in existing code. Because these abstract data types are the basis of new, reusable modules of the program, they could be translated into objects. The algorithm presented by these researchers seemed to aid in improving program

comprehension but was found to be ineffectual in identifying abstract data types in legacy code that had been heavily modified.

It is obvious from this previous research that a better method for reengineering legacy code into object-oriented code is necessary. Dynamic analysis tools should be evaluated to see if they assist in the tasks of feature location and program comprehension, which are so essential to reengineering. These tools should also be used for regression testing to provide some confidence that the new program effectively mimics the old.

CHAPTER III - TOOLS USED IN THIS WORK

Figure 1 shows the overall tool architecture needed for Software Reconnaissance. The user's target source program is first run through an *instrumentor* that inserts subroutine calls to record trace events. It is then compiled and executed. As it runs, the trace events are captured by the *trace manager,* which writes out event records to trace files. Finally, an *analysis program* compares the traces with and without the feature to do the actual code localization.

Fortunately, the analysis program used with the existing Recon2 tool for C (Wilde, 1996) could also be used for FORTRAN, so that only a new instrumentor and trace manager were needed. These were written and tested in the first phase of this project. The instrumentor and trace manager have now been made available on the web site of the Recon3 tool set at The University of West Florida. (http://www.cs.uwf.edu/~wilde/recon3/).

The instrumentor processes FORTRAN 77 code, since that is the dialect most used in the Air Force's Air Armament Center software. It allows the user to choose any combination of the following: instrumentation of subroutine entry points, subroutine return points, basic blocks (sequences of statements with no branches), and decisions (specific paths from an IF or computed GOTO statement).

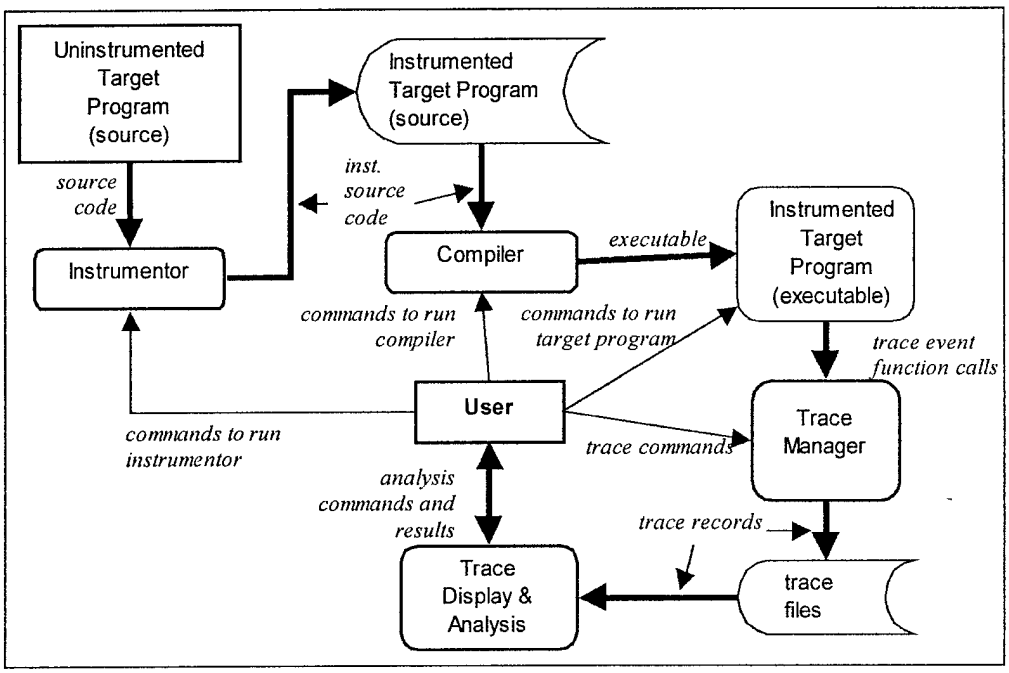


Figure 1. Software Reconnaissance tool architecture.

The trace manager component was implemented as a simple collection of FORTRAN 77 subroutines that write each trace record directly to a trace file. These subroutines are linked together with the user's instrumented target program. One trace file is produced for each execution of the program, and the trace files from runs "with" and "without" the feature are then fed to the analysis program to locate the feature.

Simultaneously with the development of the FORTRAN 77 tools, a new analysis tool called TraceGraph was written to make Software Reconnaissance easier to use. Up to now, a programmer using the technique has needed a cycle such as the one shown in Figure 2. The programmer has to switch back and forth between running tests, keeping track of trace files, setting up the analysis program so that it will know which traces exhibit each feature, and running the analysis program. While this process may be

marginally adequate for batch programs, it is particularly awkward for long running or interactive programs such as a word processor or a web server. It takes considerable time to start up and terminate each test case, so the test-analyze cycle is quite time consuming.

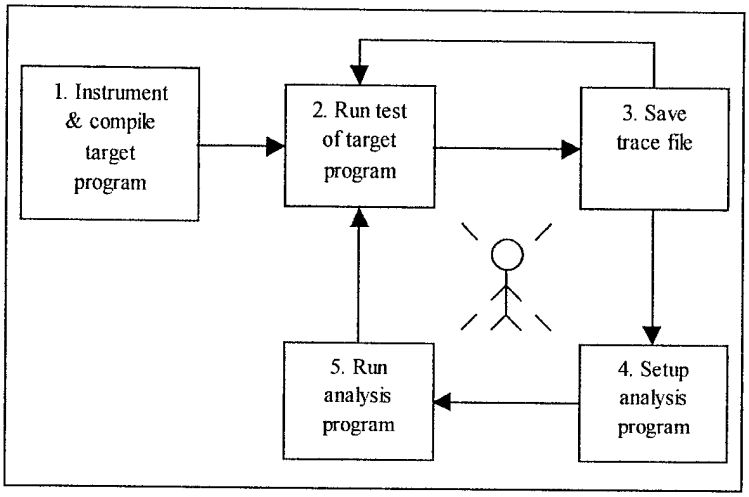


Figure 2. Locating features using Software Reconnaissance.

Accordingly we have developed a visually oriented feature location tool called TraceGraph that combines two concepts:

1. Immediate feedback from a running program

TraceGraph monitors the program as it executes; trace files are written and analyzed continuously. The maintainer of, for example, the word processor would only need to start it once. To locate the code for, say, the spell check feature, he would simply do the spell check operation in one window and immediately check the TraceGraph window to see what was executed. (Figure 3)

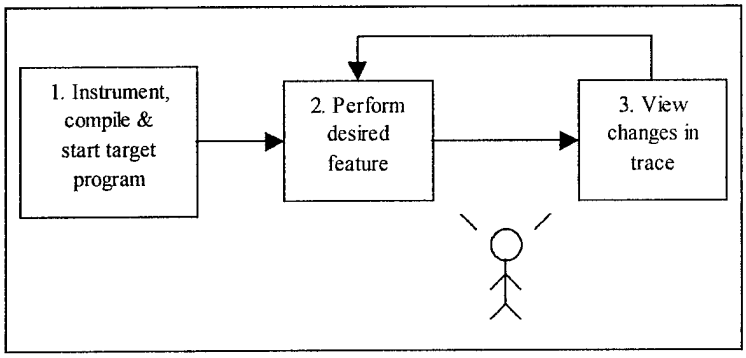


Figure 3. Locating Features Using TraceGraph

2. Graphic display of results

TraceGraph uses a display somewhat like an oscilloscope trace which slowly extends to the right as the program executes (see Figure 4). Each vertical column corresponds to a time period, say 5 seconds of execution, or to a different trace file. Each horizontal row corresponds to one software component. Each small rectangle is grey if the component was executed in that period or file, or blank if it was not. Red rectangles are used for emphasis the first time the component is executed. Figure 4 shows the display of traces from a FORTRAN 77 program where the programmer has traced tests for each feature. The red rectangles let him pick out quickly the code for a specific feature.

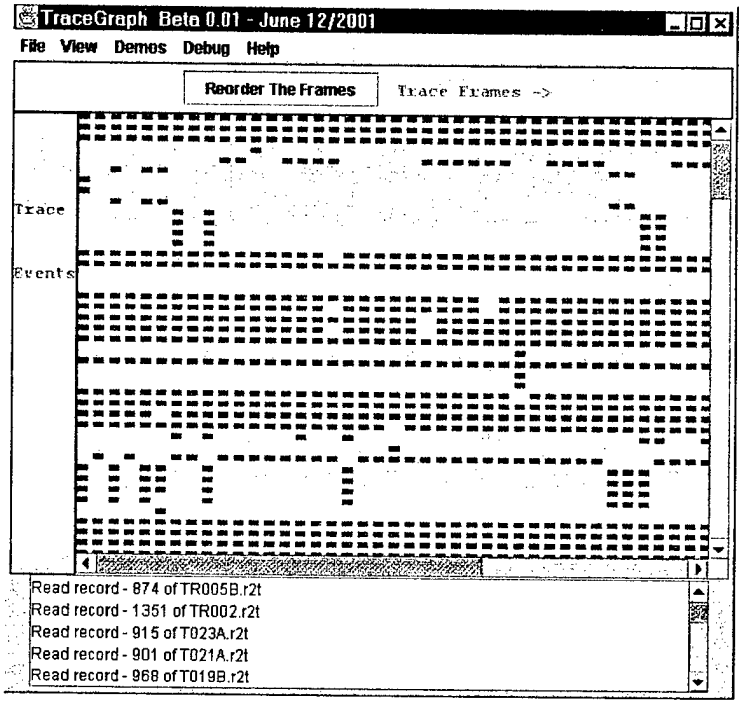


Figure 4. A TraceGraph display from a FORTRAN 77 program

Just as an engineer uses an oscilloscope to see how a circuit responds to different inputs, a programmer will be able to use TraceGraph to view how the program responds to different actions. TraceGraph also takes advantage of the human's ability to distinguish texture in images. It is easy for the eye to pick out a change in the execution of a component, even if it is represented only by a color change in a few pixels.

The TraceGraph is now available on the Recon3 web site (http://www.cs.uwf.edu/~wilde/recon3/).

As will be described in the following chapter, in our first case study the Software Reconnaissance method of feature location was compared against the Dependency Graph method proposed by Chen and Rajlich (2000). The Dependency Graph method is intended to be a computer-assisted search process, with different and alternating roles for

the computer and for the software engineer. An interactive tool for C is being developed at Wayne State University, but no FORTRAN tool was available for this study. In the absence of such a tool, the VIFOR tool (Rajlich, Damaskinos, Linos, and Silva, 1988) was used. VIFOR parses FORTRAN 77 code and creates a database of program entities and modules and of the relationships between them. The user may then formulate queries on this database which are displayed graphically in a one or two column format. VIFOR graphs of the calling dependencies (Figure 5) and of COMMON usage (Figure 6) were used to help guide the case study.

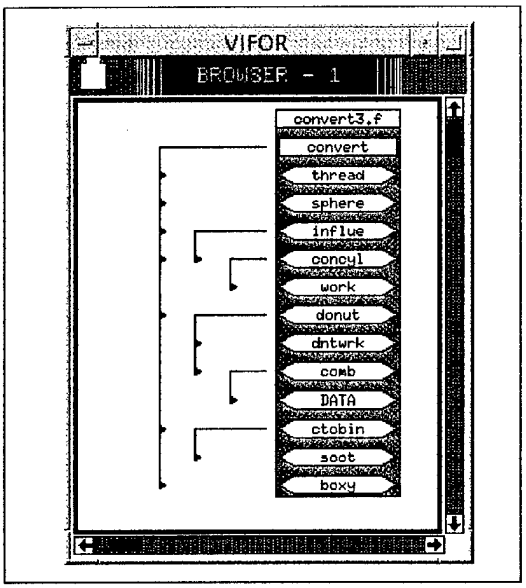


Figure 5. VIFOR screen shot of the calling relationships of the CONVERT3 program. Each icon represents a FORTRAN subroutine in the convert3.f module. The "hook lines" show the subroutines' calls.

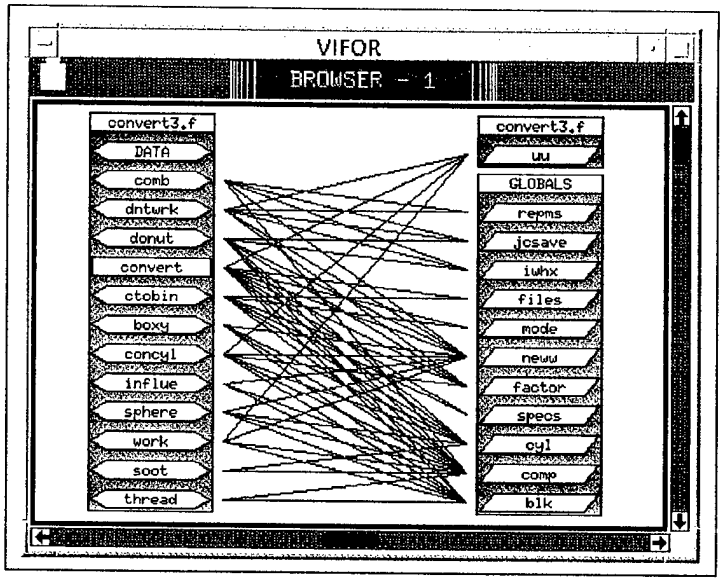


Figure 6. VIFOR screen shot of COMMON usage by subroutines in CONVERT3.
The subroutines appear in the left column and the named common blocks in the right column. A line indicates that the block is referenced by the subroutine.

CHAPTER IV - FEATURE LOCATION IN LEGACY FORTRAN CODE

The FASTGEN System

The FASTGEN geometric modeling system is a suite of programs that allows models of solid objects such as vehicles, aircraft, and so forth to be constructed from primitives such as triangles, spheres, cylinders, donuts, boxes, wedges, and rods. It is used by the Air Force to model the interactions between weapons and targets by tracing rays representing explosions or projectiles. The data used to describe individual components in a FASTGEN geometric model consist of three-dimensional coordinates, a component identification code name, a space code which identifies the area or compartment where the component resides, a material thickness, and a geometric code that defines the geometric primitive (triangle, sphere, and so forth) used in modeling the surfaces of the component.

The FASTGEN program used in the case study was CONVERT, which is a small program (2,335 lines of code), but seems to be typical of the rest of the tool suite. CONVERT is a preprocessor to expand simplified geometric model input and to transform models into the formats required by other tools that perform ray tracing or model visualization. CONVERT provides a large number of options for processing model data, especially including transformation of some primitive shapes into a set of triangles.

Aitken, Jones, and Dean (1993) pointed out that FASTGEN can approximate any type of surface by using triangles as primitives.

> This method allows any surface, flat or curved, exterior or interior, to be approximated by describing it as a series of one or more consecutively adjacent triangles whose points (vertices) are located in three-dimensional space. Flat surfaces can be described with large triangles and a few smaller ones if the surface is irregular. Curved surfaces can be

> described using several small triangles, with the size of the triangle decreasing if increased accuracy is desired. . . . Any three consecutively sequenced points define a triangle. (Aitken, Jones, & Dean, 1993, p. 2)

FASTGEN ray tracing tools accept component descriptions defined using triangle, sphere, cylinder, donut, and rod primitives. However, FASTGEN plot programs such as PIXPL (Brown, 1979) plot only those components described using triangle and rod primitives. CONVERT thus provides an option to transform components described with sphere, cylinder, and donut primitives to approximated components using triangle primitives. CONVERT always transforms components described with box and wedge primitives to equivalent components using triangle primitives.

CONVERT has a long history. Falcon Research and Development of Denver, Colorado developed the original program in 1978 for the Naval Weapons Center (NWC) at China Lake, California. The program has been maintained and updated many times in efforts to keep pace with the introduction of different hardware platforms. In the 1980s, modifications were performed by the Vulnerability Assessment Branch (DLYV) of the Air Force Armament Laboratory (AFATL), now Air Force Research Laboratory (AFRL), Eglin Air Force Base, Florida for compatibility with a Control Data Corporation (CDC) 6600 computer system. Later, it was adapted for use on the CDC Cyber 176, CRAY Y-MP 8/2128 and several Digital Corporation VAX-series computer systems. In 1994, CONVERT was modified and designated CONVERT3.0 (also known as CONVERT3) for operation on personal computers (PC) and UNIX workstations.

CONVERT3 exhibits many of the characteristics common in legacy FORTRAN code. The subroutines tend to be quite large (See Table 1) and not necessarily cohesive. Variable and subroutine names are limited by the language to six characters and are thus

usually cryptic. Much of the data is held in a series of large named common blocks, which serve to couple the subroutines closely. Figure 6 is a VIFOR screen shot showing the use of common blocks (global data) by the subroutines in CONVERT3. As can be seen, most subroutines use almost all of the COMMON blocks. Tracing data flow through programs with this sort of structure is quite difficult.

Table 1. Size of Main Program and Subroutines in CONVERT3

Subroutine	Number of lines
CONVERT (main program)	675
CTOBIN	219
SOOT	60
INFLUE	135
BOXY	116
WORK	96
SPHERE	67
THREAD	67
CONCYL	240
DONUT	234
DNTWRK	77
COMB	159
DATA	190

Note. Raw line count includes comments and blanks.

Another confusing aspect of CONVERT3 is the flow of control, which is optimized for an architecture that is now long obsolete. CONVERT3 was originally designed to run efficiently on a mid-70s mainframe. In this kind of machine, it was very important to batch together I/O operations and computations. The operating system would tend to swap out any job that was doing I/O, and thus interrupt computations. Execution was much more efficient if a large number of records could be read, then all processed together before doing any new I/O.

For this reason, CONVERT3 reads and processes in batches of 200 records. The processing loops are implemented using unstructured GOTOs that jump both forward and backward, often a hundred lines or more. The resulting structure is complex and seems to be totally arbitrary unless the programmer is aware of the kinds of optimizations used in early code.

The Case Study

The case study involved independently applying the Software Reconnaissance and Dependency Graph methods to two different features of CONVERT3. Two teams consisting of researchers from The University of West Florida Department of Computer Science participated in the study, with each team being assigned a search method and working independently. The results from the two teams were then compared.

Team A consisted of an experienced academic programmer and a graduate student, neither of whom had previous domain knowledge of the CONVERT3 program. This team used Software Reconnaissance to locate starting points in the code and then analyzed the code from these starting points. The Software Reconnaissance output was

supplemented in some cases by looking at the raw traces produced during each run to better understand the flow of control.

Team B was an experienced programmer who had worked with the CONVERT3 program almost 20 years earlier in the early 1980s. Team B was assigned the Dependency Graph search method for feature location but found that it needed some adaptation (described below) for legacy FORTRAN code.

The goals of the case study were to establish:

1. Any adaptations that might be needed in each method for use with legacy FORTRAN code.
2. The possible benefits and drawbacks of each method as applied to this domain.
3. Any inconsistencies between the results of the two methods that might give insight into their applicability.

Obviously, since the two teams not only were of different sizes but also had different levels of experience with CONVERT3, it was not relevant to directly compare time and effort between the teams.

For the study, two features of CONVERT3 were chosen that might plausibly need to be understood as part of future modifications. The program has a large number of switches and options representing different features that could have been selected. The two features finally chosen were a mirroring function and a sort function.

CONVERT3 allows mirroring to simplify data entry of symmetric objects. The user can input one component of the object and specify that it will have a "mirror" component generated automatically by reversing the y-axis coordinate. Maintainers might wish to search for the mirroring function in order to modify it, for example to mirror components

on a different or additional plane (that is, adding the z-axis as well as the x-axis and y-axis).

The sort function is related to another simplification of data entry. Often, one point may belong to several of the triangles making up a surface. CONVERT3 allows the user to enter such a point only once, but to assign it several sequence numbers indicating its participation in the different triangles. In the output file, the point is echoed several times to describe each triangle completely. The sort function guarantees that all of these points are in the right order, which is necessary for some of the other FASTGEN programs. A maintainer might need to locate this code to fix a bug or to change the sorting algorithm.

The mirroring function proved to be the more complicated of the two features to find and understand.

Team A used the Software Reconnaissance method that requires running one test "with" and one "without" the feature. It took around 20 minutes to set up the test data. In previous studies, it has been found that Software Reconnaissance works best with very simple test cases, using data "with" and "without" the feature that is as similar as possible. A very simple geometric model was created and run once with the mirroring flag set and once with it turned off. The results were then analyzed with Software Reconnaissance. The tool marked 5 areas as potentially related to mirroring, all between lines 400 and 500 of the program.

The next stage was to try to understand the marked code and its relationship to the rest of the program. This proved to be difficult, because it turned out that the program makes several passes through this area of code to handle mirroring. When the mirroring flag is set, CONVERT3 writes the component to a scratch tape during initial processing.

Then it rewinds the scratch tape, reads in the component to the same data area used previously, and jumps back to make a second pass through the same code to process the mirrored components. Presumably, the original programmer was attempting to conserve memory and reuse code. A more modern approach would have been to call a subroutine twice.

This program plan made comprehension difficult, since some of the code marked by Software Reconnaissance had been executed first time through while the rest was executed second time through. A direct reading of the marked code did not make sense. The team resorted to reading the raw trace file to learn the actual order in which statements were executed, and this eventually revealed the scratch tape program plan just described. Even with this assistance, the control variables governing the looping structure were not completely understood.

Team B adapted the Dependency Graph search method to find the mirroring function. First the user documentation was reviewed to try to refresh understanding of the feature (Jones & Aitken, 1994). Since mirroring is not exclusive to a single primitive type (that is, triangles, box, wedges, and so forth), it was likely that the control flow analysis would lead to mirroring at a somewhat non-primitive-specific area in the code. This alone eliminated quite a bit of code.

From here, Team B formalized these hypotheses:

1. since mirrored components are developed from other components in the target geometric model, some descriptive name or comment would lead Team B to the target geometric model internal data structure and variables; and

2. despite the cryptic nature of the program variable names, the use of "MIRROR" or "MIR" would perhaps be an indication of whether or not a component is to be mirrored.

Then, Team B broke the task into these sub-goals:

1. locate the input data that tells CONVERT3 to mirror a component. The user documentation indicated that character positions 75 through 77 of an 80-character target geometric model (TGM) record contain the input mirror code (Jones & Aitken, 1994); and
2. understand enough of the data structure from where the input data is located to follow the control flow to the mirroring functionality.

Team B then read the code forward linearly from the place where the TGM record was read, looking for places where the mirror code was used. Aided by Team B's domain knowledge of CONVERT3, it was then able to locate and understand the mirroring function without much difficulty.

The second part of the study examined the sort function, which sorts the sequence numbers of components. The individual components of a FASTGEN target geometric model are made up of primitive shapes: triangles, boxes, wedges, cylinders, donuts, spheres, and rods. These shapes are transformed by the CONVERT3 program into triangle primitives. The user creates a separate record for each component. Each point in the component is assigned up to eight sequence numbers that help identify how the points are connected to each other.

A convenient mathematical form for describing a component surface is based on the fact that any surface, flat or curved, can be approximated by one or more flat

> triangular surfaces. Triangles are formed by connecting three non-collinear points, thus forming a plane. If more than one triangle is required to describe a surface, then the triangles must be sequenced such that any three successive points define a triangle (not already described) or a straight line on the surface of the component. The proper selection of sequence [numbers] is critical. (Jones & Aitken, 1994, p. 21)

CONVERT3 takes a record with several sequence numbers and outputs the record several times, once with each sequence number. With the sort option activated, these records are sorted in numerical order by sequence number.

Team A adapted a sample geometric model that included sorting based from a figure in the user documentation (Jones & Aitken, 1994) to use with the Software Reconnaissance technique. To provide the two tests, the model was run with and without the sort function using the instrumented version of CONVERT3. The results were then analyzed with Software Reconnaissance. The tool marked only two areas:

1. 5 lines at line 813 in the CTOBIN subroutine where SOOT is called and checks for duplicate sequence numbers; and
2. the SOOT subroutine – comments indicate it is a shell sort.

At first, Team A had difficulty in understanding exactly what the sort function is supposed to do. The descriptions in the user documentation (Jones & Aitken, 1994) and the FASTGEN user manual (Aitken, Jones, & Dean, 1993) were somewhat confusing. The user documentation states that CONVERT3 is able to "sort records in proper sequential order," but it was unclear whether this meant that CONVERT3 would "assign sequence numbers" or "use modeler assigned sequence numbers to sort the records of a

component" (Jones & Aitken, 1994). Adding to the confusion, the FASTGEN user manual says that CONVERT3 can "create properly sequenced approximations of a component when input for the component contains multiple sequence numbers and a triangle vertex assigned on a single record" (Aitken, Jones, & Dean, 1993). This could mean that CONVERT3 creates new sequence numbers; however, this feature is listed separately from the sort. Without domain knowledge, Team A found that their main problem was determining exactly what the sort function sorts!

It took Team A 30 minutes to find the sort using Software Reconnaissance, including setup. An additional 60 minutes were spent studying the CTOBIN subroutine to understand when the sort function (in subroutine SOOT) is called. Team A finally decided that CTOBIN reads the input and performs replication of points having multiple sequence numbers. Just before it starts processing a new component, it jumps to the sort function in the SOOT subroutine to sort the old one. Afterwards, it goes back to process the first record of the new component, continuing in this manner.

Team B used the Dependency Graph search method to find the sort feature following the same process used for mirroring. By consulting the user documentation (Jones & Aitken, 1994), Team B determined why sorting was needed. Team B also noted the use of the variable name ISORT as a flag to toggle the sorting of sequence numbers. From the VIFOR calling hierarchy (Figure 5), it was apparent that the subroutine CTOBIN calls a subroutine named SOOT. Thanks to its domain knowledge, Team B realized that this is a sort with the name changed slightly so as not to duplicate a FORTRAN intrinsic sort subroutine.

Then, Team B broke the task into these sub-goals:

1. locate where the input data tells CONVERT3 to sort sequence numbers (Team B believed this would be in the subroutine CTOBIN); and
2. understand enough of the data structure from where the input data is located to follow the control flow from the input data to the sort subroutine.

From here, Team B was able to locate and understand the sort feature with little trouble.

Conclusions from the Case Study

Software Reconnaissance and Dependency Graph search are two methods for locating code that needs to be modified. It should first be pointed out that Software Reconnaissance can only locate "features," that is, program functionalities that the user can control by varying the test data. For example, mirroring or sorting in CONVERT3 are turned on or off by appropriate user inputs. The Dependency Graph method is, in principle, somewhat more flexible since it involves a human-guided search of the program. It can thus be used to locate what Biggerstaff, Mitbander, and Webster (1994) called "concepts," human-oriented expressions of computational intent. While all features are concepts, not all concepts are features and thus appropriate for Software Reconnaissance. For example, a maintainer of CONVERT3 might want to change the size of the output record buffer. It would be difficult to use Software Reconnaissance to locate "writing an output record" since all test cases write output; this "concept" cannot be turned on and off and thus is not a "feature" suitable for Software Reconnaissance. However, a programmer could apply the Dependency Graph method to locate it by starting from WRITE statements and tracing data flow backwards.

Like most programs, CONVERT3 does have many features such as the mirroring and sorting used in this case study. For both of these features, the two methods found the same code and led to essentially the same understanding of the program. The Dependency Graph method perhaps had a slight advantage in the mirroring feature, since it was able to provide Team B with an understanding of the control variables that governed the complex looping while Team A, using Software Reconnaissance, was still somewhat confused. Either team would have been able to make a simple modification, such as changing the axis of mirroring. Either team would have had considerable difficulty with a more complex modification, such as introducing simultaneous mirroring on both axes, since that would require major modifications to the obscure flow of control.

The Dependency Graph search method for feature location proved to be difficult to apply as originally described due to the lack of modularity in the code and the difficulty in interpreting the code as it was encountered. This method views the code as a graph of components that are visited systematically, with each one being completely understood before moving to another. Unfortunately, there are no clear components to understand in the CONVERT3 program. Most subroutines are large and do not follow modern conventions of cohesion and coupling so they do not constitute a meaningful "chunk" to be understood. Calling dependencies were thus not very useful. Hand tracing of data flows within subroutines proved a more useful, though tedious, approach.

Team B thus had to adapt the Dependency Graph method by exploiting the user documentation and previously acquired domain knowledge. The data items relevant for the feature were identified from the documentation. Though cryptic, names such as MIR and ISORT did sometimes serve as "beacons" to identify the purpose of data (Brooks,

1983). Then the input statements for these data items were identified, and Team B traced data flow forward, systematically exploring most of the code but skipping some areas that were clearly irrelevant. This method was successful, but seems to require the exploration of a large fraction of the code. However, the effort expended on the first feature (mirroring) did pay dividends later. In studying the sort feature, Team B found that the mirroring study had already provided an understanding of a large fraction of the relevant subroutine (CTOBIN).

The Software Reconnaissance method succeeded in locating the features in a relatively small area of the code. In previous studies of C code, it has generally (but not always) been fairly easy to understand a feature once it was located. However, with CONVERT3, understanding of located features was considerably more difficult due to:

1. poor modularity;
2. tight coupling through COMMON;
3. complex unstructured control;
4. cryptic variable names;
5. idiosyncratic program plans dominated by efficiency considerations that are now obsolete; and
6. lack of effective comments.

The use of the trace file to aid comprehension was partially effective. It did reveal the "twice through" plan in the mirroring case but Team A was unable to understand how the looping was controlled because the trace does not show the data values at each point. Software Reconnaissance is effective in feature location, but that is only part of the job of feature comprehension.

Because the Dependency Graph search method forces the user into a better understanding of how the code functions, it might be the more suitable method for maintainers who have little domain knowledge but a need for better acquaintance with the code. The Software Reconnaissance method might be more appropriate for use by maintainers who already are partially familiar with the code.

In general, both techniques were effective in locating the features, but there may be a trade-off: Software Reconnaissance locates features with less search through the code, while the Dependency Graph method requires more study, but results in more complete code comprehension. For large, infrequently changed programs, Software Reconnaissance may be the better alternative. For smaller, more frequently modified programs, where an investment in comprehension may have benefits later, the Dependency Graph method might be a better strategy. For both techniques, domain knowledge is a valuable aid, whether acquired from documentation, colleagues, or through long hours studying the code.

CHAPTER V - OBJECT-ORIENTED REENGINEERING

The object-oriented language paradigm was an obvious choice for the reengineering of the CONVERT3 program, since the program deals with geometric models. The geometric shapes used in CONVERT3 are natural candidates for objects.

It was hypothesized that reengineering the CONVERT3 program into object-oriented code would result in an effective methodology for reengineering using dynamic analysis techniques. The first goal for this methodology was that it be practical: it must not require excessive time, not require tools that are not widely available, and not require excessive time from a domain expert. The methodology should also be verifiable: all old features should have been identified and handled at the end of the process, and a sufficient set of tests should exhibit the same results with the old and new program. The third goal was that the methodology be incremental, with intermediate milestones possible. Finally, the new system should use established documentation techniques, such as the Unified Modeling Language (UML) described by Rumbaugh, Jacobson, and Booch (1999).

As shown in Figure 7, the proposed methodology involved several phases. First, any available documentation and domain experts are consulted to prepare a feature list for the legacy program and an initial UML domain class model. Pairs of test cases are written for each identified feature, one case with the feature and another similar case without it. These test cases are used both for feature location and as a regression test set to validate the reengineered program.

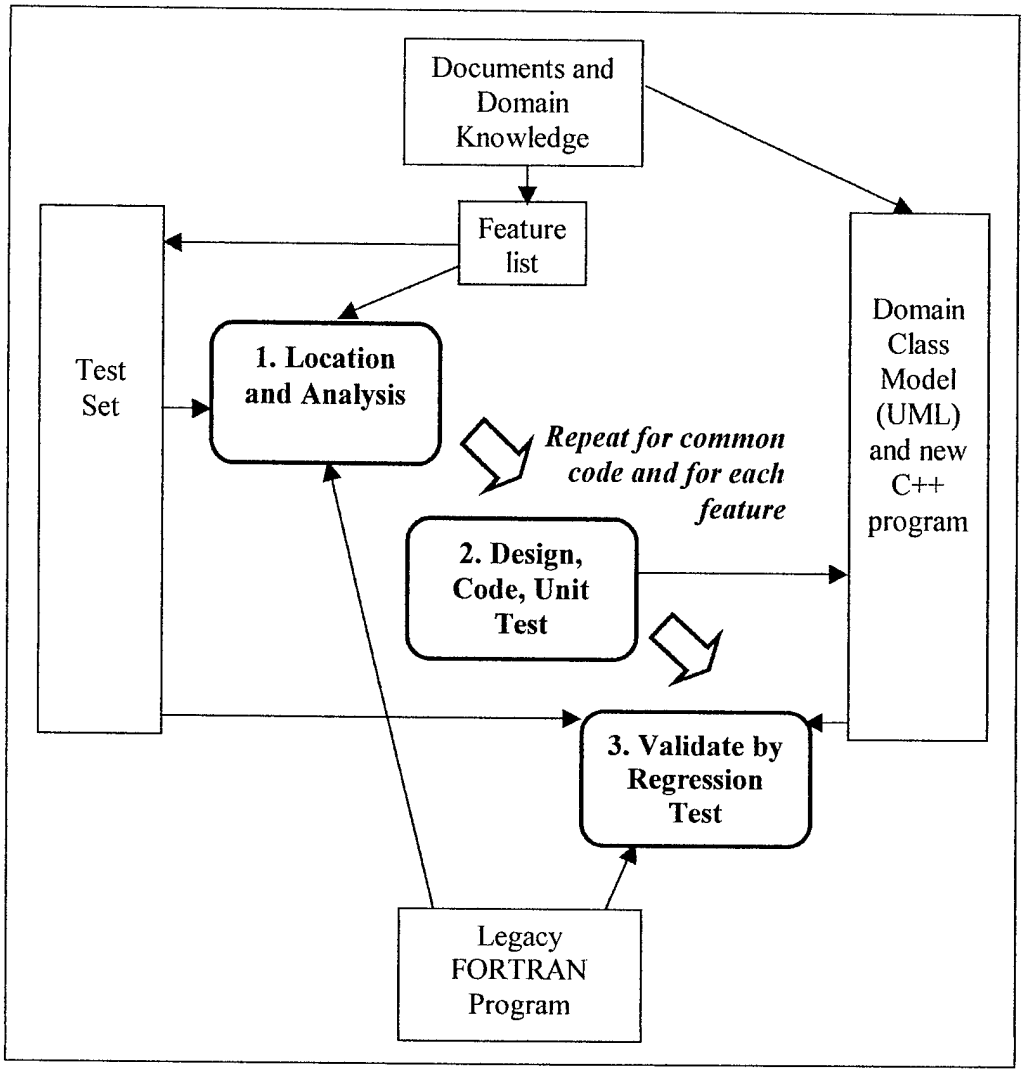


Figure 7. The reengineering methodology.

An initial Software Reconnaissance analysis is then performed using this test set. One result is the identification of the so-called "common" code, which is executed in every test case (Wilde & Scully, 1995). Common code is typically scaffolding for the application, handling initializations, opening files, utility processing, and so on. This code must be reengineered first to create a minimal version of the new program that can actually run. The three steps involved are shown in Figure 7 and outlined below.

1. Location and Analysis: The software engineer locates the code using Software Reconnaissance and the test set and analyzes it to find out exactly what it does. Some functionality may be discarded as obsolete. Study of the code may also reveal previously unidentified features that need to be reengineered. These are added to the feature list and test cases are written for them.
2. Design, Code, Unit Test: The software engineer decides which object classes will be involved in implementing this feature in the C++ program. Methods are designed, coded, and unit tested for the identified classes. In some cases the UML class design may be modified to reflect new knowledge acquired in the analysis. If object interactions are complex, the software engineer may choose to draw a UML sequence diagram for documentation.
3. Validate by Regression Test: If sufficient new code has accumulated, the software engineer performs regression testing by running the test set against both the old and new programs. Any errors of design or coding are caught as early as possible.

Once the common code has been reengineered, the rest of the process proceeds incrementally, feature by feature. A feature is chosen from the feature list and the three steps given above are again followed:

1. Location and Analysis.
2. Design, Code, Unit Test.
3. Validate by Regression Test.

The reengineering process ends when all desired features have been reengineered. If any code is still uncovered by the test set, it is examined to be sure that no important features have been missed. The finished products are a UML design and a cross-reference

between the UML design and the original code, a regression test set that covers all of the legacy program except features deemed unnecessary, and an object-oriented program that produces exactly the same output as the old program on the test set. It was theorized that this plan would cause reengineering to be faster and easier due to more effective feature location.

A number of research and practical issues emerged during the preparation of the methodology. Among the research issues raised were:

1. how much of the code would be common, how common code should be handled, and how to understand common code and allocate it to objects;
2. how much of the code is obsolete, artifact, or dead; and
3. how much help would be needed from the domain expert.

The practical issues raised included:

1. whether there are any things in FORTRAN 77 that cannot be converted to C++, such as the binary file or native function calls;
2. whether the Recon2 tool is capable of doing the analysis of common code as needed; and
3. whether tessalations and other aspects of geometric modeling should be studied, and whether this is essential for understanding CONVERT3.

The answers to many of these research and practical issues would emerge during the reengineering process itself.

The Reengineering Process

Following the reengineering methodology described in the previous section, the process began with reviewing the CONVERT3 user's manual (Jones & Aitken, 1994) and the FASTGEN user manual (Aitken, Jones, & Dean, 1993) to create an initial domain class model. Prospective classes were underlined in the text as it was read. This list of classes was used to make an initial UML class diagram, which would be revised as the project proceeded. The initial class model is shown in Figure 8. From this same source an initial list of 47 CONVERT3 features was prepared. The final list of features is shown in Appendix A.

To apply Software Reconnaissance for feature location, and to provide regression tests to validate the C++ version, two test cases were identified for each feature, one "with" the feature and a similar case "without" the feature. A list of these test cases is given in Appendix B. C shell scripts were written that allowed all the test cases to be executed at once.

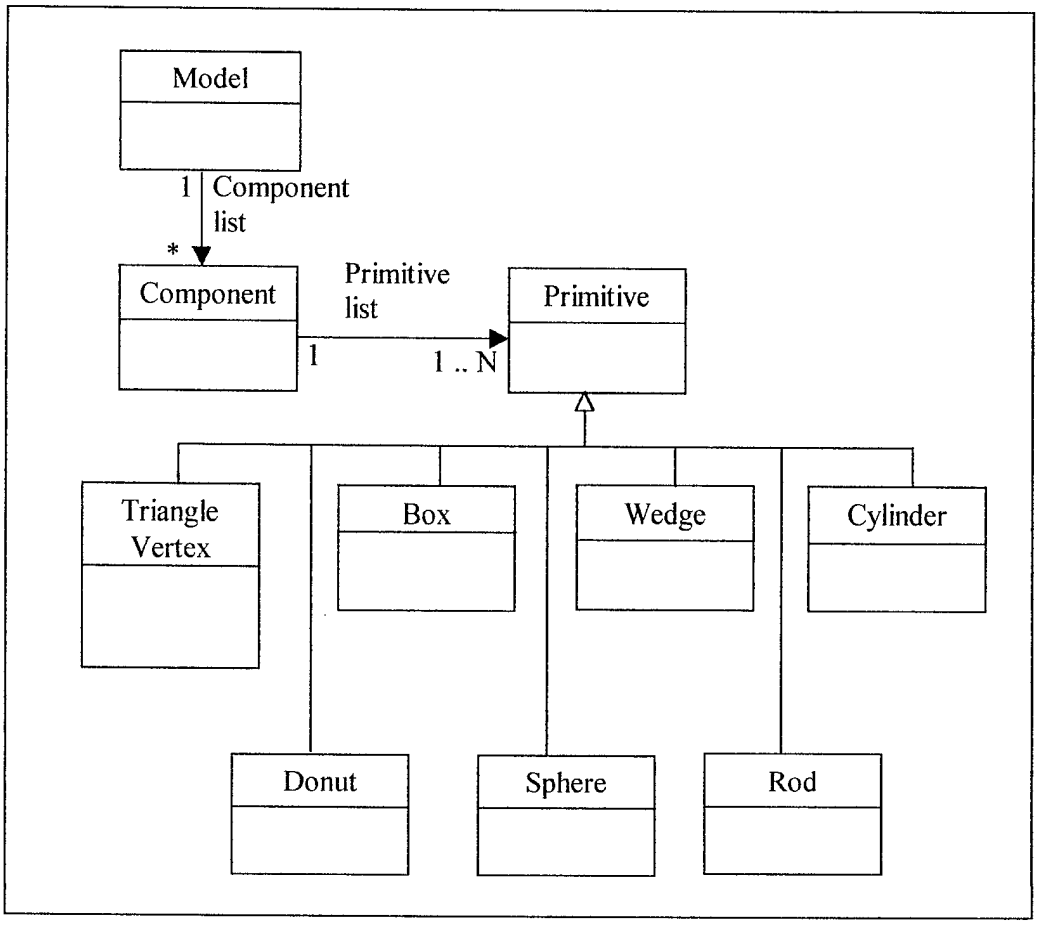


Figure 8. Initial UML domain class model for CONVERT3.

CONVERT3 was instrumented using the Software Reconnaissance FORTRAN instrumentor to trace the following code parts: subroutine entry points, subroutine return points, and basic blocks.

Running the initial test cases produced the statistics on CONVERT3 shown in Table 2. As shown, the initial tests provided a coverage of 63.4% of CONVERT3, fairly typical for a functional test set. Reengineering started with the 13.4% that is "common" code, executed on every invocation of CONVERT3. Examination of this code showed that much of it was related to reading 80 column records for a geometric model, and these

functions were assigned to loadRawRecords( ) methods in the Model and Component classes. The remaining common code performed initializations, such as opening files and setting parameters. This code was related to the particular CONVERT3 application, rather than to the domain of geometric modeling, so a Convert class was created to provide application-specific processing in static methods.

Table 2 - Code Parts (entry points, return points, and basic blocks) in CONVERT3

Type of part	Number	Percentage
Parts executed in one or more test case	265	63.4
Unexecuted parts	153	36.6
Total number of code parts	418	100.0
"Common" parts executed in all test cases	56	13.4

Reengineering of the common code, one of the research issues raised, proved hard because of extreme difficulty in understanding CONVERT3's control flow. Eventually it proved necessary to hand flowchart the two large subroutines that contained most of the common code. The identification of the common code by Software Reconnaissance helped in program comprehension. This was because as each basic block in these subroutines was examined, it was known beforehand if the block was common or only executed for some features. However, even with the flowcharts and the Reconnaissance information, the analysis was still difficult and time consuming. Eventually, though, a shell of the new C++ program was completed that emulated the basic processing of the FORTRAN version.

One unanticipated reengineering problem was the difficulty in mapping FORTRAN's input/output model into C++. For example, FORTRAN expects fixed width fields in fixed length records, with the length established in the FORMAT statement. If the actual input record is shorter, FORTRAN assumes that the remaining data are blanks, and assigns a value of zero to any numerical input fields. This behavior is hard to emulate using C++ streams, and special FortranInput and PrimitiveRecord classes were created to perform the mapping. Similarly, FORTRAN's STOP statement, which writes a message to the console, does not map easily into C++. A Utility class was created to hold a fatal error handling method that emulates it.

As code examination continued, certain additional features of CONVERT3 were identified. Some of these were error checks that had not been mentioned in the user manual, while others were features that had been missed during reading of the manual, but were found in reading the code. Tests for these features were added to the regression test set.

Each feature in the feature list was then analyzed by first applying Software Reconnaissance to locate the "marker" code for the feature, analyzing that code, and then implementing a C++ version by adding methods to the domain class model. In some cases the analysis revealed similarities between classes that allowed generalizations. For example, the processing for Wedge and for Box classes proved to be almost identical, though handled in different subroutines in the original FORTRAN. Both classes were thus made into derived classes of an abstract Prism class.

As soon as the common code and some basic input/output features had been implemented, the test set could be used to verify the reengineered version by regression

testing against the original version. Problems could thus be identified and resolved early. For example, one difficulty arose in the assignment of sequence numbers. All CONVERT3 input and output records have sequence numbers that indicate the order in which primitives, especially triangle vertices, should be interpreted in the geometric model. When CONVERT3 transforms, for example, a sphere into a set of triangles, it generates the sequence numbers using a very complicated algorithm that writes and reads a scratch tape, presumably to avoid overrunning a fixed-size array. This creates a dilemma in implementing the C++ version, which has no such array restrictions; complex and meaningless code would be needed to exactly replicate the sequence numbers. Instead it was decided to simply generate numbers in sequence, but then a filter program had to be added for regression testing to remove the numbers before comparing the output of the C++ and FORTRAN versions.

At the point where reengineering stopped, the feature list had grown to 53 features of which 35 had been reengineered, one had been discarded as obsolete, and 17 remained pending (See Appendix A for a complete list of features). The features related to cylinder, donut, and rod primitives were left pending due to lack of time.

Of the 60 regression tests, 36 ran correctly, with the remainder being related to the unimplemented features. Figure 9 shows the "as-built" UML class diagram, with the still unimplemented classes indicated by dotted boxes. Final products of the reengineering included this UML class diagram, a UML sequence diagram for one particularly complex feature, commented C++ source code for the classes making up both the domain model and the new CONVERT3 application, and the regression test set. The reengineered

product would be relatively easy to maintain or enhance, even by software engineers who had not participated in its creation.

Conclusions from the Reengineering Process

Expecting to be able to reach a complete understanding of the original code and reengineer it exactly into its new form is a bit naïve. At the start of this study, it was hoped that in the long run, more sophisticated tools could be written to support exact reengineering. After the study, it is evident that any such approach would prove unproductive for programs like CONVERT3, since much of the CONVERT3 code should probably *not* be reengineered into the new version. Such code includes the previously mentioned obsolete program plans, such as the scratch tapes, reading and writing in 200 record blocks, and the complex assignment of sequence numbers. To produce a good program, the software engineer needs to reengineer away many such details.

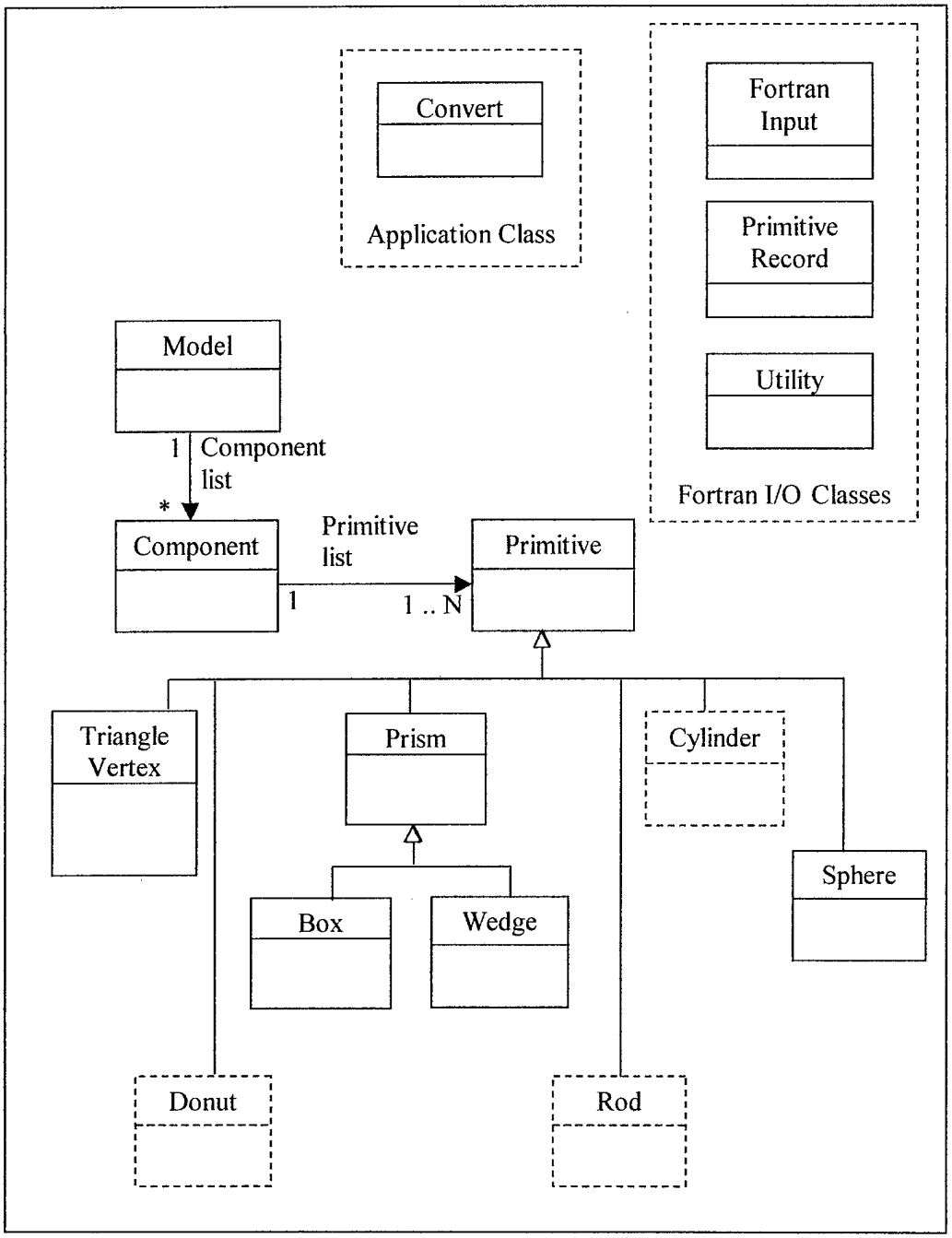


Figure 9. "As-built" UML class diagram showing domain and application-specific classes.

In fact, it is even an interesting question if complete understanding is really desirable in order to reengineer programs like CONVERT3. There were several examples of code that, though unintelligible, were certainly unnecessary. For example, there is code that

seems to be executed only when the FORTRAN program's arrays are almost full. Since the new version uses vector data structures that cannot overflow, a complete understanding of the FORTRAN processing is almost certainly a waste of time. Similarly, there is at least one large complex subroutine that was never executed in any of the tests, and seems to be used only when input flags are set to values that would be invalid according to the user manual. Apparently this code is for an obsolete feature. Since program comprehension is so time consuming for programs like CONVERT3, an efficient reengineering strategy should probably just disregard such subroutines. The dynamic analysis methodology helps identify them because they are not executed in the test set.

Though tools for exact reengineering may be impossible for programs like this, certainly some tool support of a fairly simple kind would have been useful, especially if a larger program than CONVERT3 were to be reengineered. If a larger project were started, a simple database could be prepared to track features, test cases, and the mapping between features and basic blocks. The lists kept in text files during this study became quite complicated as the project continued.

More ambitious would be a tool to help extract the fragments of FORTRAN for each feature to aid in their conversion to C++. The problem is that the whole mindset of the programmer has changed in the transition to object-orientation. The programmer of CONVERT3 clearly thought of the program as reading and writing groups of records; the outlook is entirely procedural and very low level. The new CONVERT3 instead constructs a structured geometric model of a solid object, stores it, transforms it, and then writes it out again. The large cognitive gap between these views means that the

reengineering of a feature is much more complex than simply extracting a few lines of code from one program and attaching them to another. However, some sort of specialized editor could well be useful to keep track of what has been done and to provide traceability between the old and new code.

One important aspect of any reengineering process is the need to proceed incrementally to keep project risks at an acceptable level (Olsem, 1998). The FASTGEN system, of which CONVERT3 is a part, is relatively easy to approach incrementally since the different programs communicate via files and may execute independently of each other. Thus each program may be reengineered separately.

Of the research and practical issues that were raised at the start of the reengineering process, a number were solved. The research issues raised included how to handle the common code; how much of the code is obsolete, artifact, or dead; and how much help would be needed from the domain expert. Although it comprised a relatively small percentage of the overall code, the common code proved the most difficult to reengineer due to the complicated control flow. Roughly one-third of the code was not executed in any test case, and was probably obsolete or dead. The domain expert was never consulted; most of the information needed was taken from reading the manuals and studying the code.

The practical issues raised included whether there are any things in FORTRAN 77 that cannot be converted to C++, such as the binary file or native function calls; whether the Recon2 tool is capable of doing the analysis of common code as needed; and whether tessalations and other aspects of geometric modeling should be studied, and whether this is essential for understanding CONVERT3. These issues were more easily solved.

FORTRAN proved difficult to convert to C++, especially the FORTRAN input/output model. The code was created to mimic the peculiarities of FORTRAN, but not without extra time and effort. The Recon2 tool was found to be capable of analyzing the common code. Finally, it was found that very little knowledge of computer graphics and geometric modeling was needed to understand the CONVERT3 program.

Even for the CONVERT3 program considered by itself, the methodology is incremental by feature. The version completed in the case study could be used on those geometric models that do not include the rod, cylinder, and donut primitives that are still pending. If features are implemented beginning with the most commonly used the reengineered product may thus enter partial service well before the entire task is completed.

CHAPTER VI - RESULTS AND CONCLUSIONS

We believe that the case study presented in Chapter IV shows that Software Reconnaissance is likely to be a useful technique for understanding and maintaining legacy FORTRAN programs. The method was successful in locating features that may need to be maintained, and probably requires considerably less study of code than the Dependency Graph technique. Just as had previously been found in studies of C code, Reconnaissance allows a software engineer to quickly focus on a part of the program that may need to be maintained.

The difficulties, and the contrast to our studies of C, came in understanding the code for each feature once it had been located. Programs like CONVERT, with their extraordinarily convoluted control flow, do not yield their secrets easily. Because of the GOTO's that jumped a hundred or more lines at a time and the complicated program plans (e.g. the use of scratch tapes in mirroring) the work necessary after the feature had been located was considerably more than in our experience with more modern programs.

The case study in Chapter V showed that the reengineering methodology may be a workable approach to creating an object-oriented version of badly structured legacy code. Its advantage is that it provides a systematic way of approaching the task. The test cases are used both as an aid to analysis and as a verification tool. Their coverage of the original code provides reasonable confidence that the reengineered product includes all needed functions of the original. This confidence is not, of course, the same as absolute certainty. It is well known that a program may still contain bugs, despite passing all the tests of a high-coverage test set. Similarly, the test set generated by this methodology

may not reveal differences between the old and new version that could be significant to its eventual users. The methodology is a compromise between the desire for perfect certainty and the reality of scarce software engineering time.

The code produced by the methodology is fairly "clean" and should prove a good basis for further evolution. Obsolete program plans, such as the use of scratch tapes, have been eliminated. Obsolete gold plating, such as the indexing algorithm for the sequence numbers, has also been eliminated. New program plans, such as the emulation of FORTRAN input, are provided where essential to preserve compatibility. In addition, a UML design and a test set are available as support for future development. Most importantly, the method does not only reengineer the specific product, but also produces a domain class model to serve as a reusable component in other applications.

However, the final and greatest lesson learned is that reengineering of code as complicated as CONVERT3 will always require a lot of human effort, especially if the software engineers do not have previous experience with the program. There seems to be no substitute for painstaking study to understand this kind of legacy code.

ACKNOWLEDGEMENTS

We would like to thank the many people, both at the University of West Florida and elsewhere, who have contributed to this work. Dr. Vaclav Rajlich of Wayne State University contributed both the VIFOR tool and the Dependency Graph method of feature location. His assistance in designing the case study of Chapter IV was invaluable. Ms. Michelle Buckellew was the main research assistant in this work at the University of West Florida and contributed endless hours to both case studies as well as to the documentation of their results. Henry Page and LaTreva Pounds, former UWF students currently working in the Northwest Florida region, contributed their experience with FASTGEN and CONVERT to our case studies; this work probably would not have possible without them. Finally, Kazimiras Lukoit, Scott Stowell and Tim Hennessey developed and tested the first prototype of the TraceGraph tool, which we expect to be the main analysis tool for performing Software Reconnaissance in the future.

REFERENCES

Achee, B.L., & Carver, D. (1994). A greedy approach to object identification in imperative code. Proceedings of the IEEE Third Workshop on Program Comprehension, Washington, DC, 3, 4-11.

Agrawal, H., Alberi, J., Horgan, J., Li, J. J., London, S., Wong, W. E., Ghosh, S., & Wilde, N. (1998). Mining system tests to aid software maintenance. IEEE Computer, 31 (7), 64-73.

Aitken, E. D., Jones, S. L., & Dean, A. W. (1993). A guide to FASTGEN target geometric modeling: User's manual. Fort Walton Beach, FL: ASI Systems International.

Arnold, R., & Bohner, S. (1996). Software change impact analysis. Piscataway, NJ: IEEE Computer Society.

Belady, L., & Lehman, M. (1976). A model of large program development. IBM Systems Journal, 15 (3), 225-252.

Bergey, J., Smith, D., Tilley, S., Weiderman, N., & Woods, S. (1999). Why reengineering projects fail (Tech. Rep. No. CMU/SEI-99-TR-010). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute.

Bergey, J., Smith, D., & Weiderman, N. (1999). DoD legacy system migration guidelines (Tech. Rep. No. CMU/SEI-99-TN-013). Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute.

Biggerstaff, T. J., Mitbander, B. G., & Webster, D. E. (1994). Program understanding and the concept assignment problem. Communications of the ACM, 37 (5), 72-83.

Blazy, S., & Facon, P. (1993). Partial evaluation as an aid to the comprehension of fortran programs. Proceedings of the IEEE Second Workshop on Program Comprehension, Capri, Italy, 93, 48-54.

Blazy, S., & Facon, P. (1994). SFAC, a tool for program comprehension by specialization. Proceedings of the IEEE Third Workshop on Program Comprehension, Washington, DC, 3, 162-167.

Brooks, R. (1983). Towards a theory of the comprehension of computer programs. International Journal of Man-Machine Studies, 18, 543-554.

Brown, G. A. (1979). PIXPL target plotting program (P7057) (Air Force Armament Laboratory Tech. Rept. No. AFATL-TR-79-45). Eglin AFB, FL: Datatec, Inc.

Canfora, G., Cimitile, A., De Lucia, A., & Di Lucca, G. A. (2001). Decomposing legacy systems into objects: an eclectic approach. Submitted to Information and Software Technology.

Canfora, G., Cimitile, A., & Munro, M. (1996). An improved algorithm for identifying objects in code. Software - Practice and Experience, 26 (1), 25-48.

Chen, K., & Rajlich, V. (2000). Case study of feature location using dependence graph. Proceedings of the IEEE Eighth Workshop on Program Comprehension, Limerick, Ireland, 8, 241-247.

Chikofsky, E. J., & Cross, J. H., II. (1990). Reverse engineering and design recovery: A taxonomy. IEEE Software, 7 (1), 13-17.

Cimitile, A., De Lucia, A., Di Lucca, G. A., & Fasolino, A. R. (1999). Identifying objects in legacy systems using design metrics. Journal of Systems and Software, 44, 199-211.

Cimitile, A., Tortorella, M., & Munro, M. (1994). Program comprehension through the identification of abstract data types. Proceedings of the IEEE Third Workshop on Program Comprehension, Washington, DC, 3, 12-19.

Corbi, T. A. (1989). Program understanding: Challenge for the 1990s. IBM Systems Journal, 28 (2), 294-306.

DeBaud, J. M., & Rugaber, S. (1995). A software re-engineering method using domain models. Proceedings of the IEEE International Conference on Software Maintenance, Seattle, WA, 95, 204-213.

Di Lucca, G. A., Fasolino, A. R., & De Carlini, U. (2000). Recovering class diagrams from data-intensive legacy systems. Proceedings of the IEEE International Conference on Software Engineering, San Jose, CA, 00, 52-63.

Jerding, D., & Rugaber, S. (1997). Using visualization for architectural localization and extraction. Proceedings of the IEEE Fourth Working Conference on Reverse Engineering, Amsterdam, the Netherlands, 4, 56-65.

Jones, S. L., & Aitken, E. D. (1994). Convert3.0 user's manual. Fort Walton Beach, FL: ASI Systems International.

Kaliski, M. E., & Kaliski, B. S. (1991). The software sleuth. St. Paul, MN: West Publishing Company.

Koenemann, J., & Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. Proceedings of the ACM Conference on Computer Human Interaction, New Orleans, LA, 91, 125-130.

Kozaczynski, W., & Wilde, N. (1992). On the re-engineering of transaction systems. Journal of Software Maintenance: Research and Practice, 4 (3), 143-162.

Lakhotia, A. (1993). Understanding someone else's code: Analysis of experiences. Journal of Systems and Software, 23, 269-275.

Lakhotia, A. (1997). A unified framework for expressing software subsystem classification techniques. Journal of Systems and Software, 36, 211-231.

Linos, P., Aubet, P., Dumas, L., Helleboid, Y., LeJeune, P., & Tulula, P. (1994). Visualizing program dependencies: An experimental study. Software – Practice and Experience, 24 (4), 387-403.

Liu, S. S., & Wilde, N. (1990). Identifying objects in a conventional procedural language: An example of data design recovery. Proceedings of the IEEE Conference on Software Maintenance, Washington, DC, 90, 266-271.

Livadas, P. E., & Johnson, T. (1994). A new approach to finding objects in programs. Journal of Software Maintenance: Research and Practice, 6 (5), 249-260.

Olsem, M. R. (1998). An incremental approach to software systems reengineering. Journal of Software Maintenance: Research and Practice, 10 (3), 181-201.

Ornburn, S. B., & Rugaber, S. (1992). Reverse engineering: Resolving conflicts between expected and actual software designs. Proceedings of the IEEE Conference on Software Maintenance, Orlando, FL, 92, 32-40.

Queille, J. P., Voidrot, J. F., Wilde, N., & Munro, M. (1994). The impact analysis task in software maintenance: A model and a case study. Proceedings of the IEEE International Conference on Software Maintenance, Washington, DC, 3, 234-242.

Rajlich, V., Damaskinos, N., Linos, P., & Silva, J. (1988). Visual support for programming-in-the-large. Proceedings of the IEEE International Conference on Software Maintenance, Phoenix, AZ, 88, 92-99.

Robson, D. J., Bennett, K. H., Cornelius, B. J., & Munro, M. (1991). Approaches to program comprehension. Journal of Systems and Software, 14 (2), 79-84.

Rugaber, S. (1992). Program comprehension for reverse engineering. Proceedings of the AAAI Workshop on AI and Automated Program Understanding, San Jose, CA, 106-110.

Rugaber, S., Stirewalt, K., & Wills, L. (1995). Detecting interleaving. Proceedings of the IEEE International Conference on Software Maintenance, Nice, France, 95, 265-274.

Rugaber, S., & White, J. (1998). Restoring a legacy: Lessons learned. IEEE Software, 15 (4), 28-33.

Rumbaugh, J., Jacobson, I., & Booch, G. (1999). The unified modeling language. Reading, MA: Addison Wesley.

Sneed, H. M., & Majnar, R. (1998). A case study in software wrapping. Proceedings of the IEEE International Conference on Software Maintenance, Bethesda, MD, 98, 86-93.

Subramaniam, G., & Byrne, E. (1996). Deriving an object model from legacy fortran code. Proceedings of the IEEE International Conference on Software Maintenance, Monterey, CA, 96, 3-12.

Turver, R. J., & Munro, M. (1994). An early impact analysis technique for software maintenance. Journal of Software Maintenance: Research and Practice, 6 (1), 35-52.

Von Mayrhauser, A., & Vans, A. M. (1995). Program comprehension during software maintenance and evolution. IEEE Computer, 28 (8), 44-55.

Wilde, N. (1996). RECON - tool for C programmers [On-line]. Available: http://www.cs.uwf.edu/~wilde/recon/

Wilde, N., & Casey, C. (1996). Early field experience with the software reconnaissance technique for program comprehension. Proceedings of the IEEE International Conference on Software Maintenance, Monterey, CA, 96, 312-318.

Wilde, N., Casey, C., Vandeville, J., Trio, G., & Hotz, D. (1998). Reverse engineering of software threads: A design recovery technique for large multi-process systems. The Journal of Systems and Software, 43, 11-17.

Wilde, N., & Scully, M. (1995). Software reconnaissance: Mapping program features to code. Journal of Software Maintenance: Research and Practice, 7, 49-62.

Yau, S. S., Collofello, J. S., & MacGregor, T. (1978). Ripple effect analysis of software maintenance. Proceedings of Compsac 78, New York, NY, 78, 60-65.

APPENDIX A - FINAL LIST OF CONVERT3 FEATURES

This table shows which features were reengineered into the object-oriented version of CONVERT3 and which features were still pending at the completion of the reengineering process described in Chapter IV. The first table contains features identified at the beginning of the reengineering process; the second table contains features discovered during the reengineering process itself.

Features	**Action Taken**
Transform sphere to triangles	reengineered
Transform cylinder to triangles	pending
Transform donut to triangles	pending
Create mirror image component description	reengineered
Create properly sequenced approximations	reengineered
Sort records for components	reengineered
Process box	reengineered
Process wedge	reengineered
Interactive mode	reengineered
Batch mode	pending
Process cylinder without transforming to triangles	pending
Process triangle	reengineered
Process donut without transforming to triangles	pending
Process sphere without transforming to triangles	reengineered
Process rod	pending
Process components with triangle primitives >2.99" thick ($NARM)	pending
Process components of normal thickness	reengineered
Convert all components to triangle primitives	pending
Print ASCII output file	reengineered
Do not print ASCII output file	reengineered
Print only 200th record in the block	discarded feature

	(obsolete)
Print each entire block of 200 records	reengineered
Write binary file	reengineered
Do not write binary file	reengineered
Drop components from converted target deck	reengineered
Do not drop components from converted target deck	reengineered
Break long cylinders/cones	pending
Break long rods	pending
Use non-default break ratio	pending
Process intentional interference records ($INTERFE)	reengineered
Process non-interfering records	reengineered
Process component codes ($CODE)	reengineered
Process component without component code	reengineered
Process cylinder in plate mode	pending
Process cylinder in volume mode	pending
Process donut in plate mode	pending
Process donut in volume mode	pending
Process box in plate mode	reengineered
Process box in volume mode	reengineered
Process wedge in plate mode	reengineered
Process wedge in volume mode	reengineered
Process sphere in plate mode	reengineered
Process sphere in volume mode	reengineered
Process triangle in plate mode	reengineered
Process triangle in volume mode	reengineered
Process component with target ID number ($VEHICLE)	reengineered
Process component without target ID number	reengineered

Features Found During Reengineering	**Action Taken**
Error check for input file	pending
Error check for target deck file	pending
Drop more than 14 components from converted	reengineered

target deck	
Error check for duplicate sequence numbers	reengineered
Create mirror image triangle component with –1 flag	reengineered
Process triangle without mirroring	reengineered

APPENDIX B - LIST OF TEST CASES USED IN REENGINEERING CONVERT3

This is the final list and descriptions of each test case used in reengineering the CONVERT3 program.

Test Case	Description
T001A	A sphere component to be transformed into triangles
T001B	A sphere component not transformed into triangles
T002A	A cylinder component to be transformed into triangles
T002B	A cylinder component not transformed into triangles
T002C	A cylinder component with L/R ratio greater than 10
T002D	A cylinder component with L/R ratio greater than FACT
T003A	A donut component to be transformed into triangles
T003B	A donut component not transformed into triangles
T003C	A donut component with all components transformed
T004A	A component to be mirrored
T004B	A component not to be mirrored
T005A	A component to have sequence numbers sorted
T005B	A component to have sequence numbers unsorted
T006	A box component to be processed
T007	A wedge component to be processed
T008A	A component to be processed in interactive mode
T008B	A component to be processed in batch mode
T009	A triangle component to be processed
T010A	A rod mode component to be processed
T010B	A rod mode component with L/R ratio greater than 10
T011A	A component with thickness greater than 2.99"
T011B	A component with normal thickness
T012A	A component to be processed without an output file
T012B	A component to be processed with an output file
T013A	A component to be processed with only the 200th record in each block to be written
T013B	A component to be processed with each block of 200 records to be written
T014A	A component to be processed without a binary out file

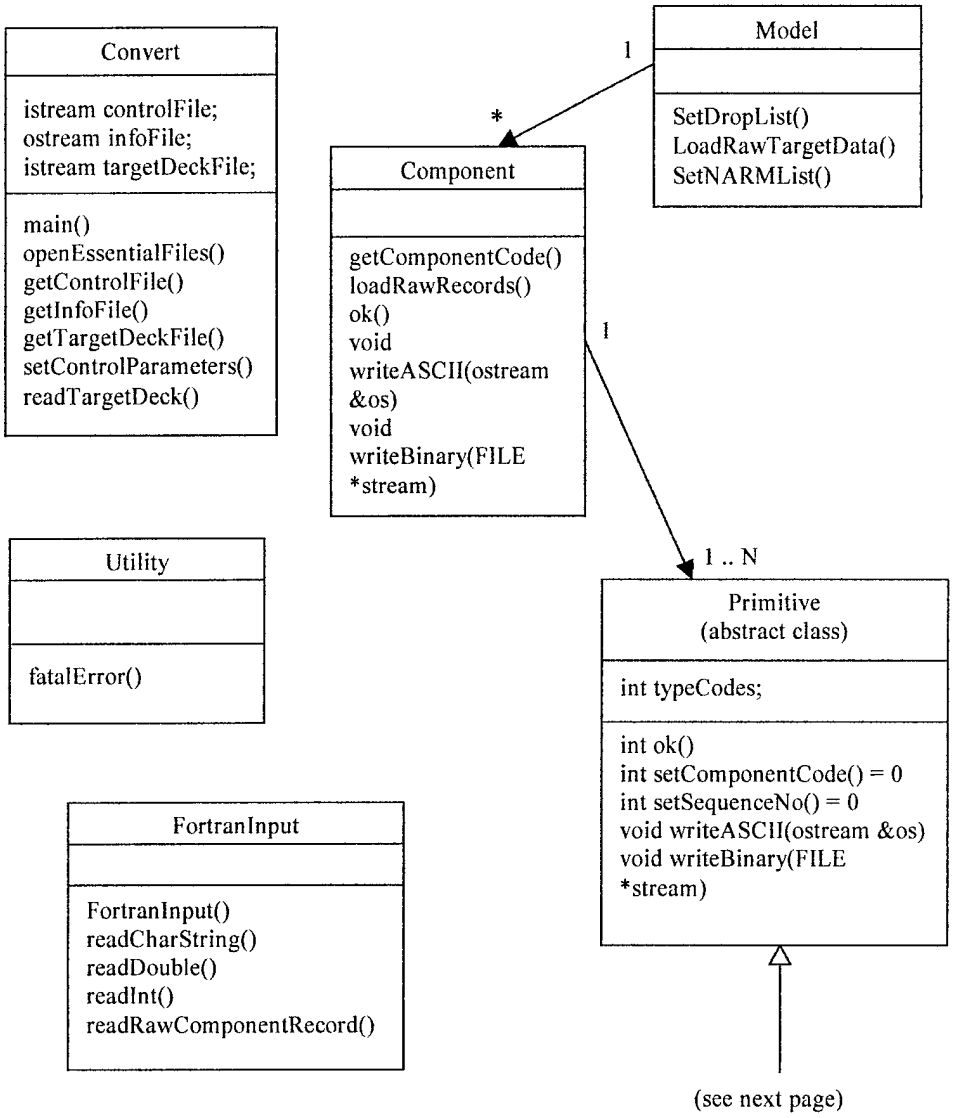
T014B	A component to be processed with a binary out file
T015A	Two cylinder components, one to be dropped
T015B	Two cylinder components, none to be dropped
T016A	Two box components, to be processed as intentional interference records
T016B	Two box components to be processed normally
T017A	A component to be processed with component code
T017B	A component to be processed without component code
T018A	A cylinder component in plate mode
T018B	A cylinder component in volume mode
T019A	A donut component in plate mode
T019B	A donut component in volume mode
T020A	A box component in plate mode
T020B	A box component in volume mode
T021A	A wedge component in plate mode
T021B	A wedge component in volume mode
T022A	A sphere component in plate mode
T022B	A sphere component in volume mode
T023A	A triangle component in plate mode
T023B	A triangle component in volume mode
T024A	A box component with target ID number
T024B	A box component with no target ID number
TR001A	A box component with an error in IV
TR001B	A box component with an error in IDONT
TR001C	A box component with an error in ISORT
TR001D	A box component with an error in IASCI
TR001E	A box component with an error in KPRNT
TR001F	A box component with an error in IBIN
TR001G	A box component with an error in FACT
TR001H	A box component with an error in PTSS
TR001I	A box component with an error in PSTT
TR002	A box component with an error in target deck
TR003	More than 14 components to be dropped
TR004	Two triangle components with duplicate sequence numbers
TR005A	A triangle component with -1 mirroring
TR005B	A triangle component without mirroring

APPENDIX C - DESCRIPTION OF OBJECT-ORIENTED CONVERT3 CODE

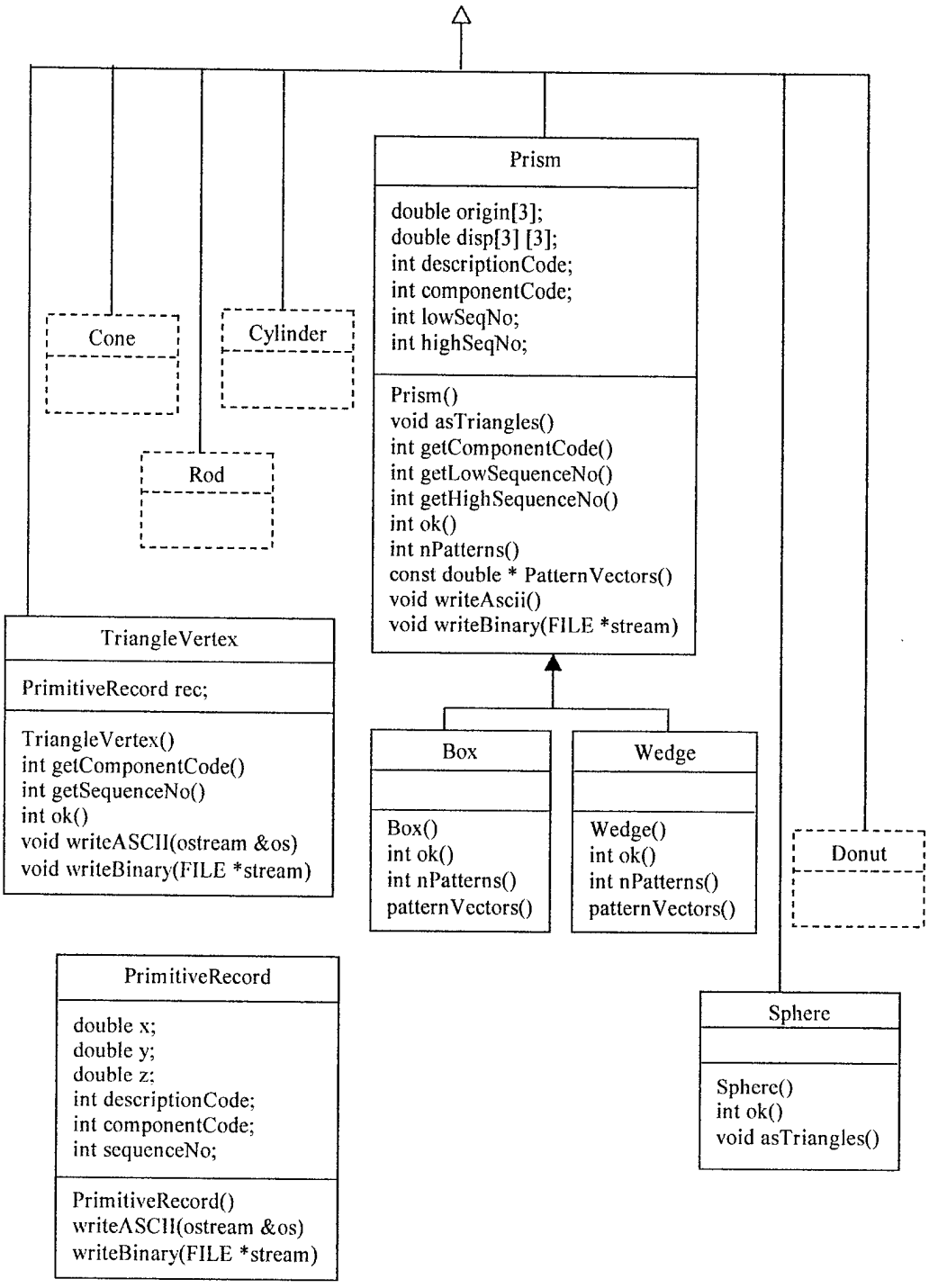
The object-oriented version of CONVERT3 was written in C++ and contains 5,367 lines of code. The entire FORTRAN version of CONVERT3 was not reengineered, but most of the major features of the old program have been recreated in the new one. (See Appendix A for a complete list of reengineered features.) The files that make up the object-oriented version of CONVERT3 are as follows: Box.cc, Component.cc, Convert.cc, FortranInput.cc, Model.cc, Primitive.cc, Prism.cc, Sphere.cc, TriangleVertex.cc, Utility.cc, and Wedge.cc. Each executable .cc file has a corresponding header file with the same name. The main( ) subroutine is found in the Convert.cc file.

The Component.cc file contains the implementation for a component object, each of which represents one component of a target geometric model. The Convert.cc file contains the main( ) subroutine and several related subroutines. FortranInput.cc partially implements the FORTRAN input routines. Model.cc is an implementation of the Model object, which represents a complete target geometric model. Primitive.cc contains implementation for the classes Primitive and PrimitiveRecord. The files named after geometric shapes, Box.cc, Prism.cc, Sphere.cc, TriangleVertex.cc, and Wedge.cc, implement the classes that deal with their namesake geometric figures. Utility.cc is used to generate error messages.

Figure 9 shows the "as-built" UML class diagram, with the still unimplemented classes indicated by dotted boxes. A more detailed class diagram follows, also with the unimplemented classes, Donut, Rod, Cylinder, and Cone, indicated by dotted boxes.



(see next page)


Prism
double origin[3];
double disp[3] [3];
int descriptionCode;
int componentCode;
int lowSeqNo;
int highSeqNo;
Prism()
void asTriangles()
int getComponentCode()
int getLowSequenceNo()
int getHighSequenceNo()
int ok()
int nPatterns()
const double * PatternVectors()
void writeAscii()
void writeBinary(FILE *stream)
Cone
Cylinder
Rod
TriangleVertex
PrimitiveRecord rec;
TriangleVertex()
int getComponentCode()
int getSequenceNo()
int ok()
void writeASCII(ostream &os)
void writeBinary(FILE *stream)
Box
Box()
int ok()
int nPatterns()
patternVectors()
Wedge
Wedge()
int ok()
int nPatterns()
patternVectors()
Donut
PrimitiveRecord
double x;
double y;
double z;
int descriptionCode;
int componentCode;
int sequenceNo;
PrimitiveRecord()
writeASCII(ostream &os)
writeBinary(FILE *stream)
Sphere
Sphere()
int ok()
void asTriangles()