A Case Study of Feature Location in Unstructured Legacy Fortran Code[1,2]

Norman Wilde
Michelle Buckellew
University of West Florida

Henry Page
SRS Technologies ASI Division

Vaclav Rajlich
Wayne State University

Abstract

Feature location is a major problem in supporting legacy code. Two methods, the Software Reconnaissance technique and the Dependency Graph search method, have been proposed to help software engineers locate features in a program that needs to be modified. Both methods have been developed and evaluated largely using relatively modern C code. This report describes preliminary experiences in adapting them to legacy Fortran code typical of systems developed in the 1970's.

A case study was performed to apply the two methods to the CONVERT3 program, which is part of the FASTGEN geometric modeling suite. CONVERT3 exhibits many characteristics typical of legacy Fortran programs such as poor modularity, prolific use of unstructured GOTOs, tight coupling through large common blocks, and confusing program plans introduced to provide efficiency on now-obsolete platforms.

Both methods were effective in locating features in CONVERT3. The Dependency Graph search method required considerable adaptation to be useful on this kind of program and required the systematic exploration of a large amount of code. The Software Reconnaissance technique was able to locate features with less searching but, as compared to earlier experience with C, the Fortran code proved more difficult to understand once it had been located.

1. Introduction

Program comprehension is almost certainly one of the biggest time sinks in the software industry. In the maintenance phase alone it has been estimated that programmers spend half or more of their time trying to understand the structure and behavior of the system being maintained [CORB.89]. Software engineers need to understand legacy code in order to fix bugs, add features, or adapt to new environments.

Code location is a big part of code comprehension. It is fairly rare to modify a large system as a whole. In more typical tasks, software engineers locate the code that implements a particular concept or feature, analyze it to understand it, and then modify it and as little of the surrounding code as possible.

Software engineers have told us that one of the most frequent questions they need to answer in dealing with legacy systems is "Where does this program implement feature X?" For example, a maintainer may need to know: "Where is 'call forwarding' handled in this PBX switch?" or "Where in this C compiler is all the code that handles 'typedef' statements?" Software Reconnaissance and Dependency Graph search are two methods that have been proposed for locating features in unfamiliar code.

Software Reconnaissance uses test cases to help locate features. The technique has been described more completely in [WILD.95]. Simply stated, it is based on a comparison of traces of different test cases. The target program is first instrumented so that a trace is produced of the blocks or decisions executed in each test. Then test cases are run, some "with" and others "without" the desired feature. The traces are then analyzed to look for blocks or decisions that are executed "with" the feature but not "without".

Software Reconnaissance has been tried effectively on C code from a number of companies [WILD.96, AGRA.98]. Tool development and many of these studies were supported by the Software Engineering Research Center, an NSF supported industry-university cooperative research center. The University of West Florida has developed the Recon2 tool for C which is freely available [RECON2]. There are also now commercial tools available that provide a Reconnaissance capability for C, such as χSuds™ from Telcordia [XSUDS].

[1] Address correspondence to: Norman Wilde, Department of Computer Science, University of West Florida, 11000 University Parkway, Pensacola, Florida 32514, USA; email: nwilde@uwf.edu; http://www.cs.uwf.edu/~wilde; tel. +1-850-474-2542, fax. +1-850-857-6056

[2] This work was supported by the United States Air Force Office of Scientific Research (AFOSR) under grant number F49620-99-1-0057

0-7695-1028-0/01 $10.00 © 2001 IEEE

The Dependency Graph method of feature location has been described by Chen and Rajlich, who give a case study of the NCSA Mosaic software written in C [CHEN.00]. The method involves a search in the component dependency graph, beginning at a starting node which, in the absence of other information, would be the main( ) function. In each step one component (generally a C function) is selected for a visit. The software engineer explores the source code, dependency graph, and documentation to understand the component and decide if it is relevant or unrelated to the feature. The search then proceeds to another component until all the components related to the given feature are found and understood.

As described, both of these methods have so far chiefly been used on C code. However there remains a large legacy of systems in earlier languages that are still in use and must still be maintained. Legacy Fortran code has characteristics which make it particularly difficult to understand and maintain, so it is of interest to see if the Reconnaissance and Dependency Graph methods are still useful.

This report describes a case study which applied these two methods to CONVERT3, a modest sized but fairly representative example of legacy Fortran code. The objective of the case study was to adapt each method to locate features in this kind of code and to judge how each might be used to support its maintenance. The use of two different methods in the case study also provided a useful cross check on the validity of the feature locations identified by each one.

2. Tool Development

For this study a Software Reconnaissance tool was developed for Fortran. Figure 1 shows the overall tool architecture needed for 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 [RECON2] could also be used for Fortran, so that only a new instrumentor and trace manager were needed. It is expected that these will become part of the new Recon3 tool set which is currently being developed at the University of West Florida.

The instrumentor processes Fortran 77 code. It allows the user to choose any combination of instrumentation of:

1. Subroutine entry points
2. Subroutine return points
3. Basic blocks (sequences of statements with no branches)
4. Decisions (specific paths from an IF or computed GOTO statement)

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.

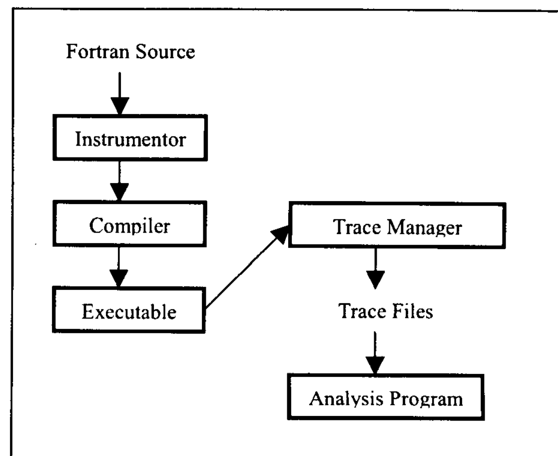

Figure 1
Software Reconnaissance Tool Architecture

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 is being developed for C, but no Fortran tool was available for this study. In the absence of such a tool, the VIFOR tool [RAJL.88] was used. VIFOR parses Fortran 77 code and creates a data base of program entities and modules and of the relationships between them. The user may then formulate queries on this data base which are displayed graphically in a one or two column format. VIFOR graphs of the calling dependencies (Figure 2) and of COMMON usage (Figure 3) were used to help guide the case study.

3. The FASTGEN System

The FASTGEN geometric modeling system is a suite of programs that allows models of solid objects such as vehicles, aircraft, etc. to be constructed from primitives such as triangles, spheres, cylinders, donuts, boxes, wedges, and rods. It is used by the United States Air Force to model the interactions between weapons and

targets by tracing rays representing explosions or projectiles.

The FASTGEN program used in the case study was CONVERT, which is modest sized, 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, including especially transformation of some primitive shapes into a set of triangles, as needed by other programs in the FASTGEN suite.

CONVERT has a long history. The original program was developed in 1978 for the Naval Weapons Center (NWC), China Lake, California by Falcon Research and Development of Denver, Colorado. 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 Armament Center (AAC), 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 for operation on personal computers (PC) and UNIX workstations.

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
TOTAL	**2335**

Table 1
Size of Main Program and Subroutines in CONVERT3
Raw line count includes comments and blanks

Version 3.0 of CONVERT consists of a single source code module (CONVERT3.F) containing 2,335 lines of code and comments written in Fortran 77. Table 1 shows the sizes of the main program and its subroutines.

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 6 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 3 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.

Another confusing aspect of CONVERT3 is the flow of control, which is optimized for an architecture which is now long obsolete. CONVERT3 was originally designed to run efficiently on a mid-70's 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 GOTO's that jump both forwards and backwards, 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.

Finally, as is common with legacy FORTRAN programs, CONVERT3 has been maintained many times by many different programmers over the years. The comments and documentation within the code (where they exist) can be obscure and perplexing. Fortunately however, there are two very good manuals for CONVERT3 users [JONE.94, AITK.93]. While these manuals help the user prepare data for CONVERT3 program and understand what it is doing, they do not directly help the maintainer decipher the code.

4. The Case Study

The case study involved independently applying the Software Reconnaissance and Dependency Graph methods to two different features of CONVERT3. Two teams 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 twenty 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, say to mirror components on a different or additional plane (i.e., 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.

5. The Mirroring Function

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

Team A used the Software Reconnaissance method which requires running one test "with" and one "without" the feature. It took around 20 minutes to set up the test data. In previous studies, we have 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 line number 400 and line number 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.

However 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. However 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 [JONE.94]. Since mirroring is not exclusive to a single primitive type (i.e., triangles, box, wedges, etc.), it was likely that the control flow analysis will lead to mirroring at a somewhat non-primitive-specific area in the code. As a sanity check, this alone eliminated quite a bit of code.

From here, Team B formalized these hypotheses:

- 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.
- 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:

- Locate the input data that tells CONVERT3 to mirror a component. [JONE.94] indicates that character positions 75 through 77 of an 80-character target geometric model (TGM) record contain the input mirror code.
- 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.

6. The Sort Function

The second part of the study examined the sort function, which, as previously described, sorts the sequence numbers of components.

Team A ran two tests of a simple geometric model, one with and one without the sort function enabled. The results were then analyzed with Software Reconnaissance. The tool marked only two areas:

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

It took Team A thirty minutes to find the sort using Software Reconnaissance, including setup. An additional sixty minutes was spent in the somewhat harder task of studying the unstructured CTOBIN subroutine to understand exactly when the sort function (in subroutine SOOT) is called.

Team B used the Dependency Graph search method to find the sort feature following the same process used for mirroring. By consulting [JONE.94], 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 2), 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:

- Locate where the input data tells CONVERT3 to sort sequence numbers. Team B believed this would be in the subroutine 'CTOBIN.'
- 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.

7. Conclusions

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 et. al called "concepts", human-oriented expressions of computational intent [BIGG.94]. 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 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.

However 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 came 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 understand 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 [BROO.83]. 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 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 our 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:

- poor modularity
- tight coupling through COMMON
- complex unstructured control
- cryptic variable names
- idiosyncratic program plans dominated by efficiency considerations that are now obsolete
- 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 great aid, whether acquired from documentation, colleagues, or through long hours studying the code.

8. Related Work

While the literature on program comprehension has now become quite extensive, there is relatively little work that specifically addresses the problem of locating where software features are implemented in code. Lakhotia has clearly identified the need for methods of feature location from his analysis of the difficulties of practical code comprehension [LAKH.93]. Many models of program comprehension emphasize the need to relate code to problem domain concepts [VONM.95]. But tools or methods for locating features and concepts have been fairly limited.

For the specific case of Fortran code, Blazy and Facon have proposed a method based on partial evaluation and constant propagation [BLAZ.93]. Some of the program inputs are specified and a specialized version of the program is generated that will provide the same outputs as the original with the specified values. The purpose of the method is to extract a complete compileable program, rather than to aid a human in comprehension.

Program slicing [WEIS.82] is a static data flow analysis method that also attempts to produce a complete program that is smaller than the original. Many variations of slicing have been proposed, but the most commonly described is backward slicing that takes a set of variables, typically program outputs, and extracts the program that would be sufficient to compute their values. Forward slicing from an input might be more useful for feature location if there is a simple input variable that controls the feature. However in many cases much of the original program will be contained in such a slice.

Biggerstaff et. al. provide a general description of the *concept assignment problem* in program comprehension [BIGG.94]. They contrast approaches based on parsing, which may be effective for programming-oriented concepts, with the kind of processing needed to identify human-oriented concepts such as the features located in this study. They describe the prototype DESIRE tool which incorporates parsing, clustering, analysis of data items and their names, and an interactive browser.

Ripple analysis [YAU.78] and impact analysis [QUEI.94] 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 provide a collection of papers covering different impact analysis approaches [ARNO.96]. 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. For example Turver and Munro [TURV.94] 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.

9. Bibliography

[AGRA.98] Agrawal, Hira; Alberi, James; Horgan, Joseph; Li, J. Jenny; London, Saul; Wong, W. Eric; Ghosh, Sudipto; Wilde, Norman, "Mining System Tests to Aid Software Maintenance", *IEEE Computer*, Vol. 31, No. 7, (July, 1998), pp. 64 - 73.

[AITK.93] Aitken, Edward D.; Jones, Susan L.; Dean, Allen W., *A Guide to FASTGEN Target Geometric Modeling: User's Manual*, ASI Systems International, Fort Walton Beach, FL: May 1993.

[ARNO.96] Arnold, Robert, and Bohner, Shawn, *Software Change Impact Analysis*, IEEE Computer Society, 1996, ISBN-0-8186-7384-2.

[BIGG.94] Biggerstaff, Ted; Mitbander, Bharat; Webster, Dallas, "Program Understanding and the Concept Assignment Problem", *Communications of the ACM*, Vol. 37, No. 5, May 1994, pp. 72 - 83.

[BLAZ.93] Blazy, Sandrine and Facon, Philippe, "Partial Evaluation as an Aid to the Comprehension of Fortran Programs", *Proc. IEEE Second Workshop on Program Comprehension*, IEEE Comp. Soc. Press, July, 1993, pp. 48-54.

[BROW.79] Brown, Gary A. *PIXPL Target Plotting Program (P7057)*, AFATL-TR-79-45, Datatec, Inc., for the Air Force Armament Laboratory, Eglin AFB, FL: April 1979.

[BROO.83] Brooks, Ruven, "Towards a Theory of the Comprehension of Computer Programs", *International Journal of Man-Machine Studies*, Vol. 18, 1983, pp 543-554.

[CHEN.00] Chen, Kunrong and Rajlich, Vaclav, "Case Study of Feature Location Using Dependence Graph", *Proc. International Workshop on Program Comprehension, 2000,* IEEE Comp. Soc. Press, June 2000, pp. 241-249.

[CORB.89] Corbi, T. A., "Program Understanding: Challenge for the 1990s", *IBM Systems Journal*, Vol. 28, No. 2, 1989, pp. 294 - 306.

[JONE.94] Jones, Susan L.; Aitken, Edward D., *Convert3.0 User's Manual*, ASI Systems International, Fort Walton Beach, FL: March 1994.

[LAKH.93] Lakhotia, Arun, "Understanding Someone Else's Code: Analysis of Experience", *Journal of Systems and Software*, Vol. 32, pp. 269 - 275, December 1993.

[QUEI.94] Queille, J.-P.; Voidrot, J.-F.; Wilde, N.; Munro, M., "The Impact Analysis Task in Software Maintenance: a Model and a Case Study", *Proc. IEEE International Conference on Software Maintenance - 1994*, IEEE Computer Society, September 1994, pp. 234 - 242.

[RAJL.88] Rajlich, Vaclav; Damaskinos, Nicholas; Linos, Panagiotis; Silva, Joao, "Visual Support for Programming-in-the-large", *Proc. International Conference on Software Maintenance*, IEEE Computer Society, October 1988, pp. 92-99.

[RECON2] "RECON - tool for C Programmers", http://www.cs.uwf.edu/~wilde/recon/.

[TURV.94] Turver, R. J. and Munro, M., "An Early Impact Analysis Technique for Software Maintenance", *Journal of Software Maintenance*, Vol. 6, No. 1, pp. 35-52, 1994.

[VONM.95] Von Mayrhauser, Annelise and Vans, A. Marie, "Program Comprehension During Software Maintenance and

Evolution", *IEEE Computer*, Vol. 28, No. 8, August, 1995, pp. 44 - 55.

[WEIS.82] Weiser, Mark, "Programmers Use Slices When Debugging", *Communications of the ACM*, Vol. 25, No. 7, July 1982, pp. 446 - 452.

[WILD.95] Wilde, Norman and Scully, Michael, "Software Reconnaissance: Mapping Program Features to Code", *Journal of Software Maintenance: Research and Practice*, Vol. 7, January 1995, pp. 49 - 62.

[WILD.96] Norman Wilde, Christopher Casey, "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension", *Proc. IEEE International Conference on Software Maintenance - 1996*, IEEE Computer Society, November 1996, pp. 312 - 318.

[XSUDS] Telcordia Software Visualization and Analysis Toolsuite, http://xsuds.argreenhouse.com/.

[YAU.78] Yau, S. S.; Collofello, J. S.; MacGregor, T., "Ripple Effect Analysis of Software Maintenance", *Proc. Compsac 78*, IEEE Computer Society, 1978, pp 60 - 65.

Appendix
VIFOR Screen Shots from the CONVERT3 Program

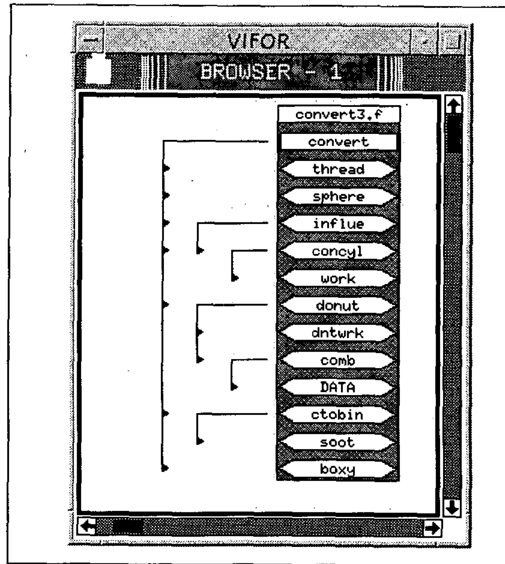

Figure 2
Calling Relationships of the CONVERT3 Program
Each icon represents a Fortran subroutine in the convert3.f module. The "hook lines" show the subroutines calls.

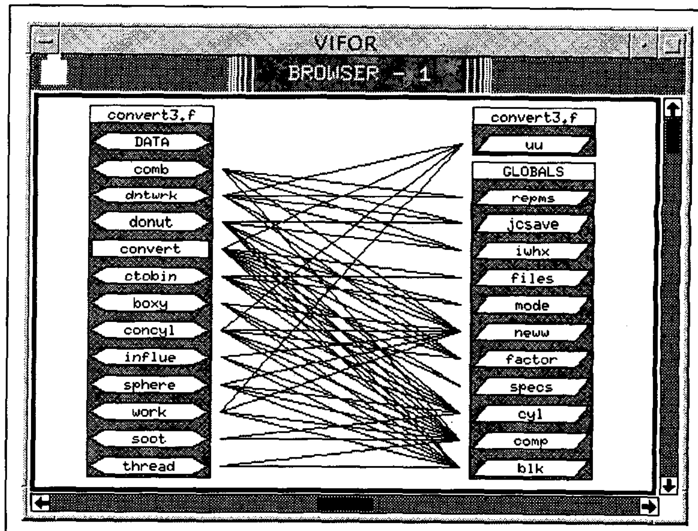

Figure 3
COMMON Block Usage by Subroutines in CONVERT3
The subroutines appear in the left column and the named common blocks in the right column. A line indicates that the block is referenced by the subroutine