ELSEVIER

The Journal of Systems and Software 65 (2003) 105–114

The Journal of Systems and Software

www.elsevier.com/locate/jss

A comparison of methods for locating features in legacy software ☆

Norman Wilde [a,*], Michelle Buckellew [b], Henry Page [c], Vaclav Rajlich [d], LaTreva Pounds [e]

[a] *Department of Computer Science, University of West Florida, 11000 University Parkway, Pensacola, FL 32514, USA*
[b] *Lockheed Martin Integrated Systems, 8529 South Park Circle, Suite 300, Orlando, FL 32819, USA*
[c] *Micro Systems Inc., 35 Hill Ave. NW, Fort Walton Beach, FL 32548, USA*
[d] *Department of Computer Science, Wayne State University, Detroit, MI 48202, USA*
[e] *Dynetics Inc., 2 Clifford Dr., Shalimar, FL 32579, USA*

Received 17 April 2001; received in revised form 17 August 2001; accepted 19 November 2001

Abstract

Software engineers frequently need to locate the code that implements a specific *feature* of a program in order to fix a problem or add an enhancement. Several methods have recently been proposed to aid in feature location, notably the software reconnaissance method, which uses dynamic analysis of traces of execution, and the dependency graph method which involves static tracing of calling and data flow relationships in the program's dependency graph. Most studies performed so far on these methods have used relatively modern C code. However there is a large body of existing legacy software in Fortran and similar languages which is often much more poorly structured. This paper describes a case study to locate two features in a sample of poorly structured legacy Fortran code. Both methods were applied to locate the features, along with the well known "grep" text search method for comparison. Both the software reconnaissance and dependency graph methods located both features, although some difficulties were encountered and adaptations were needed due to the very tangled nature of the code. The "grep" search method worked well for one of the two features, but was ineffective for the second, more complex case.

© 2002 Elsevier Science Inc. All rights reserved.

1. Introduction

One of the problems most frequently faced in dealing with legacy software is the location of the code for a specific *feature*. Large programs provide many features for their users. A word processor provides features for file import, editing, cut and paste, printing, and so on. A telephone PBX switch provides features such as call waiting, call forwarding, speed dialing, as well as many features for accounting, switch configuration, etc.

Software engineers rarely have the luxury of analyzing an entire program before making fixes or enhancements. Instead they need to locate the most relevant code, understand it, and make the change so as to minimize unwanted side effects. Since the change is often related to a feature, they need to locate the code that implements the feature. Thus in adding a new kind of file import to the word processor, a responsible maintainer would locate and study the existing file import options and then make his modification match the existing design as closely as possible. If he can, he will reuse some of the existing code.

However in legacy systems a feature is often not implemented in a single place. Instead its code may be distributed over several different modules that interact in complex ways. The design for the feature has become a *delocalized plan* (Letovsky and Soloway, 1986). It can be very difficult to locate all the relevant parts of such a plan in a large program, especially one that has undergone

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

* Corresponding author. Tel.: +1-850-474-2542; fax: +1-850-857-6056.

E-mail addresses: nwilde@uwf.edu (N. Wilde), michelle.buckellew@lmco.com (M. Buckellew), hpage@gomicrosystems.com (H. Page), vtr@cs.wayne.edu (V. Rajlich), LaTreva.Pounds@dynetics.com (L. Pounds).

URL: http://www.cs.uwf.edu/~wilde.

0164-1212/03/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00052-3

many years of maintenance at the hands of different software engineers.

In an ideal world, features could be located using traceability documentation. Each feature would be an item in the software specification. A traceability matrix (or other cross-referencing system) would show how each specification item maps to each design item and, if necessary, an additional matrix might relate design items to code components. Unfortunately two problems emerge in practice. First, as the software evolves, the features that become relevant for maintenance may not map well into the requirements as originally conceived. Second, the traceability documentation is difficult to update and often is not maintained under the time pressures of ongoing work. Thus it is fairly rare to find good traceability documentation for existing legacy code.

Several methods have been proposed recently for locating features in existing code. The software reconnaissance method uses test cases to help locate features. The technique has been described more completely in (Wilde and Scully, 1995). 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 (Wilde and Casey, 1996; Agrawal et al., 1998). 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, 2001). There are also now commercial tools available that provide a Reconnaissance capability for C, such as χSuds™ from Telcordia (XSUDS, 2001).

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 and Rajlich, 2000). 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.

Finally, the "grep" text search method of feature location has traditionally been used for years by software engineers dealing with unfamiliar software. [1] The survey conducted by Sim, Clarke and Holt clearly indicates that "grep" and similar textual searches are very commonly used by programmers (Sim et al., 1998). Grep and similar tools provide very fast search for text strings matching a given regular expression. The tool is used to search for relevant comments, variable names, and so on that the software engineer hypothesizes as being present in the code. Each segment that is found must be studied, and this study may give rise to new queries to look for additional comments, subroutines or variables. One problem with the grep method is that there is no obvious stopping point if a meaningful comment or code segment is not found quickly. Grep may find that variable A is used to set B and C, B is used to set D, D to set E and F, and so on. Taken to an extreme, the grep method may degenerate into an exhaustive search of the call or data flow graph of the program.

As described previously, the software reconnaissance and dependency graph methods have so far chiefly been studied on C code. The C code studies have been performed on programs that are relatively modern in structure. There remains a large legacy of systems in earlier languages that are still in use and must still be maintained. More specifically, legacy Fortran code has characteristics which make it particularly difficult to understand and maintain, so it is of interest to see which methods are effective for that environment.

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

2. 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 (Lakhotia, 1993). Many models of program comprehension emphasize the need to relate code to problem domain concepts

[1] We name the method after the famous "grep" regular expression search tool which has been available for many years on almost all versions of the Unix operating system. Obviously any other good text search tool, including the search feature of an editor or word processor, could also be used.

(Brooks, 1983; Von Mayrhauser and Vans, 1995). 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 (Blazy and Facon, 1993). 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 (Weiser, 1982) 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 (Biggerstaff et al., 1994). 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 et al., 1978) and impact analysis (Queille et al., 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 provide a collection of papers covering different impact analysis approaches (Arnold and Bohner, 1996). 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 (1994) 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, as previously mentioned, this is fairly rare in practice.

Numerous visual aids have been proposed to aid in querying and browsing a program during program comprehension. Some early examples include the tools described in (Rajlich et al., 1988; Cleveland, 1989; Chen et al., 1990; Wilde and Huitt, 1991). Most of these systems, like most impact analysis tools, display information about relationships extracted from the code. Thus they will not normally allow a query about a user feature unless the programmer already has identified a code component related to the feature. Given such a starting point, browsers provide support for feature location, but they do not, of themselves, provide any particular methodology for addressing this problem.

3. The FASTGEN system used in the case study

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 CONVERT3, which is modest sized, but seems to be typical of the rest of the tool suite. CONVERT3 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. CONVERT3 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 and Unix workstations.

Version 3.0 of CONVERT consists of a single source code module (CONVERT3.F) containing 2335 lines of code and comments written in Fortran 77. Table 1 shows the sizes of the main program and its subroutines.

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
Total	2335

Raw line count includes comments and blanks.

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.

The poor structuring of CONVERT3 may be shown using the VIFOR tool (Rajlich et al., 1988). 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 (Fig. 1) and of COMMON usage (Fig. 2) were used to help guide the case study.

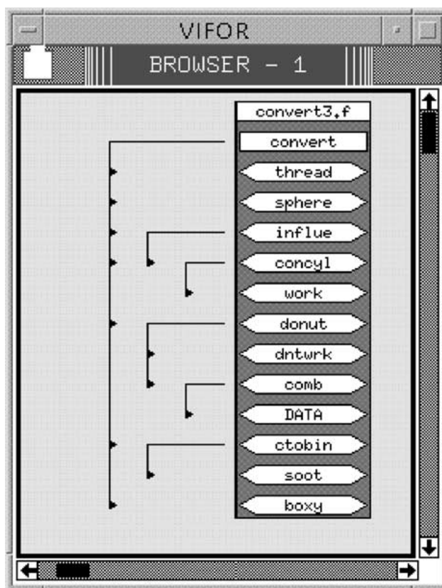

Fig. 1. Calling relationships of the CONVERT3 program. Each icon represents a Fortran subroutine in the CONVERT3.F module. The “hook lines” show the subroutines calls.

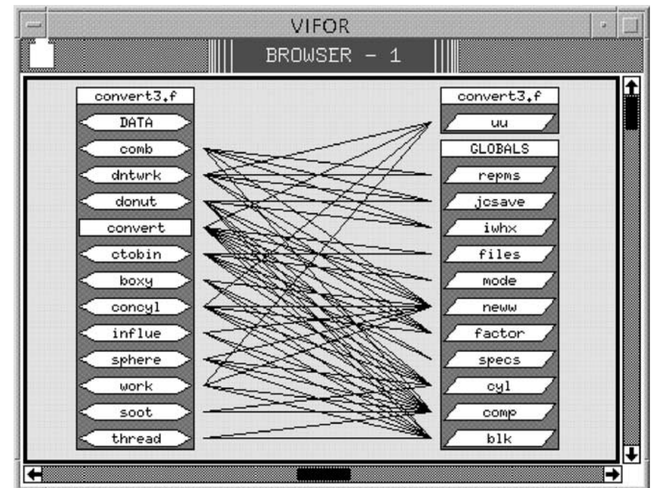

Fig. 2. COMMON block usage by subroutines in CONVERT3. The subroutines appear in the left column and the named common blocks in the right column. A line indicates that the block is referenced by the subroutine.

Fig. 2 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-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 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 (Jones and Aitken, 1994; Aitken et al., 1993). 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, dependency graph, and "grep" methods to two different features of CONVERT3. Four programmers participated in the study grouped in three "teams" with each team being assigned a search method and working independently. [2] The results from the teams were then compared.

Each of the three methods works best with some tool support. For software reconnaissance a rough instrumentor was written for Fortran 77 that allowed the analysis tool from the RECON2 system to be used. For the dependency graph method no good tool was available, but the VIFOR tool described previously provided some aid in laying out the large-scale structure of the program. For the "grep" method, obviously the `grep` tool of the Unix operating system was used.

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 consisted of a single 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 in the Appendix A) for legacy Fortran code.

Team C consisted of a single experienced Fortran programmer who had worked with the FASTGEN modeling system in the early 1980's, preparing geometric models and documenting some tools that use FASTGEN output. Thus she was familiar with the problem domain, but not specifically with CONVERT3 or its code. Team C was assigned the grep text search method.

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 three methods that might give insight into their applicability.

Obviously since the three 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* feature and a *sort* feature.

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 feature 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- and y-axis).

The sort feature 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 feature guarantees that all of these points are output 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. Data collection and analysis

To prepare for the study, each team was given a brief description of the steps of the method it was to follow. Each was also given a line numbered listing of the CONVERT3 program. As participants looked for each feature, they noted a sequence number in the margin of this listing each time they visited a particular point in the code. On a separate sheet they noted the reason for each visit. Thus it was possible to roughly reconstruct the thought processes followed in locating a feature. Each team also made general notes of difficulties encountered with its assigned method and of any adaptations that were necessary to deal with CONVERT3 code. After each team had finished its work, it met with the experimenters for a "debriefing", to compare results and to discuss general conclusions from the experience. The

[2] Since the software reconnaissance method required both Fortran experience and the creation of test cases, the software reconnaissance team grouped a programmer with little knowledge of Fortran but some knowledge of testing, with another who had more extensive Fortran knowledge. The other "teams" involved just a single programmer each.

overall conclusions of this paper are derived from these discussions.

A detailed explanation of the steps followed by each team is given in the Appendix A. The sort feature, turned out to be the simplest and all three teams were able to find it fairly easily, perhaps because it was primarily located in a single subroutine and controlled by a single variable.

The mirroring feature proved to be considerably more difficult because it involves complicated flow of control with two passes through a key code segment and with intermediate results saved on a scratch tape between passes. Team C (grep) did not gain enough understanding to make any changes to this feature. A comparison of the results for Team A (software reconnaissance) and Team B (dependency graph) would seem to indicate that both probably understood the mirroring feature well enough to have made a simple change such as changing the mirror axis from Y to Z. The dependency graph method perhaps had a slight advantage, since it was able to understand the control variables that governed the complex looping while Team A was still somewhat confused. However a more complex change, such as providing simultaneous mirroring on several axes, would have been problematic for both teams since major changes would be required in the obscure flow of control.

6. Conclusions

Software reconnaissance, dependency graph search and "grep" search are three methods for locating code that needs to be modified. It should 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 and "grep" methods are, in principle, somewhat more flexible since they involve a human-guided search of the program. They can thus be used to locate what Biggerstaff et al. called "concepts", human-oriented expressions of computational intent (Biggerstaff et al., 1994). 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 or "grep" method to locate it by starting from WRITE statements and tracing data flow backwards.

Any conclusions about the three methods must obviously be tentative, since they are based on a single case study of a single program. The following paragraphs and Table 2 give our current impressions as derived from this experience and earlier experience with C programs, but they must be considered as tentative informed opinion, not proven fact.

The "grep" method would seem to be the least reliable of the three methods, in that it depends vitally on the presence of matching comments or variable names at just the right points in the legacy program. This condition may sometimes be met, but certainly not always. In the specific case of CONVERT3 enough variables were matched to get a grasp of the sort feature, but the few fragments found for the mirroring feature were not enough to give any really useful insight.

On the other hand, the "grep" method had the considerable advantage of being very quick to use and of requiring only tools that were readily available. It should perhaps be thought of as a first method of attack on the feature location problem, which can be useful if it succeeds, but to be quickly abandoned for more advanced methods if it does not.

Next after "grep", the software reconnaissance method appeared to be the fastest method of location. It required instrumenting the code and writing a few test cases, but the return for this effort was a very quick marking of code that was relevant for each of the features studied. Software reconnaissance allows the maintainer to focus in very quickly on a small area of code. If applied to a well-structured program the located code may be understood fairly readily. In the specific case of CONVERT3, software reconnaissance easily

Table 2
Characteristics of the three feature location methods

Method	Description	Advantages	Disadvantages
Software reconnaissance	Compare code executed in traces "with" and "without" the feature	Fairly quick; focuses on a small fraction of the code	Feature must be controllable by input data; best for locally comprehensible code
Dependency graph search	Systematic search through program's dependency graph of control and data flow	Few limitations; provides understanding of the context of the feature	May require extensive search and be more time consuming; best for well modularized code
Grep	Text search for keywords in comments and variables	Very quick; tools readily available	Least reliable; requires good textual clues in program

found the subroutine that handles the sort feature and the location from which that subroutine is called.

However if the code is not "locally comprehensible" by study of the marked code and adjacent code, software reconnaissance may not provide enough context to allow a feature to be understood. For the mirroring feature the correct area of code was marked, but the two-pass algorithm made the markers very difficult to interpret. Code executed on the first pass is intertwined with code executed on the second and Reconnaissance, by itself, provides no help in distinguishing the two. Location is only the first step in comprehension, and in programs like CONVERT3 comprehension of code fragments may be very difficult.

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.

The dependency graph method is the most systematic of the three methods, and so may be the most time consuming. For well modularized code the search may be relatively efficient since a quick study of a module's comments and name can indicate that it is irrelevant to the feature being sought. In poorly modularized code such as CONVERT3, the search became more difficult and involved studying a large portion of the program before the feature was located.

However the advantage of this more systematic search was that, necessarily, more of the code was being understood as the search progresses. When the relevant fragments were finally located, their context was more clearly known. As well, the knowledge accumulated in one search can benefit another, as in the case of CONVERT3 where the study of mirroring yielded enough information as a byproduct so that the sort could be identified and understood almost immediately.

Each of these methods thus has its place in the Software Engineer's toolkit. The "grep" method may be the best starting point since it is so quick; if it is unsuccessful, little time has been lost. 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 all three techniques, domain knowledge is a great aid, whether acquired from documentation, colleagues, or through long hours studying the code.

Appendix A. Locating the two features

A.1. Locating the mirroring feature

The mirroring feature 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 min to setup 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 by studying the code adjacent to the marked code. 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.

In summary, the software reconnaissance method located the mirroring function quite rapidly and provided partial, but not complete, understanding of its implementation.

Team B found that the dependency graph search method could not be applied as originally described due to the lack of modularity in CONVERT3. Each Fortran subroutine was too large to be understood as a single

"chunk" so it had to be broken down into smaller fragments of a few lines each for study. Accordingly, Team B adapted the dependency graph search method to find the mirroring feature. First the user documentation was reviewed to try to refresh understanding of the feature (Jones and Aitken, 1994). Since mirroring is not exclusive to a single primitive type (i.e., triangles, box, wedges, etc.), it was likely that mirroring would be located in code that is not specific to one kind of primitive. 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 (TGM), some descriptive name or comment would lead Team B to the TGM 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 subgoals:

- Locate the input data that tells CONVERT3 to mirror a component. Jones and Aitken (1994) indicates that character positions 75 through 77 of an 80-character 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 input mirror code was used. Aided by Team B's domain knowledge of CONVERT3, it was then able to locate and understand the mirroring feature without much difficulty.

In summary, the dependency graph search method required considerable adaptation to handle this nonmodular code, and the search required an almost complete line-by-line study moving forward from the point of input. This search was greatly aided by previous domain knowledge of CONVERT3. The method was effective in locating the mirroring code and arrived at a good understanding of how it worked.

Team C ("grep") also prepared for the study by reading the CONVERT3 documentation and identifying the relevant input variables for the mirroring feature. The first "grep" query was for the string "MIRRORING", which produced no hits. The second try was "MIRROR" which produced one hit on a comment about a scratch file and several other hits on a 200 element array called MIRROR. By comparing a READ statement with the user input documentation, Team C determined that MIRROR was a flag indicating that a component is to be mirrored. The array is in COMMON and is set in five different subroutines, but used only once to set an element of another array called MIR.

The next string to try was thus MIR which (since MIR is a substring of MIRROR) produced all the previous hits, plus several more on the MIR array which is also in COMMON and is set in three subroutines and used in two. An important use seemed to be in a subroutine called CTOBIN so a query was made on that string. The CTOBIN subroutine was found, as was the place where CTOBIN is called, but a scan of this subroutine did not reveal any code related to mirroring so attention returned to the MIR array. At one location the values of variables named NEWNAM or ISQNAM were set depending on the value of MIR. However a search of the code near where these variables are set found no additional references to them. The "grep" method thus petered out at this point since the only remaining strategy would be to switch to data flow analysis and try to trace out NEWNAM and ISQNAM and any other variables that depend upon them.

In summary, the "grep" method failed to clearly locate the mirroring code or to provide any insight into this feature, because textual clues in the program were inadequate.

A.2. Locating the sort feature

The second part of the study examined the sort feature, 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 feature 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 test case setup and execution.

Team B used the dependency graph search method to find the sort feature following the same process used for mirroring. By consulting (Jones and 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 (Fig. 1), 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 subgoals:

- 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.

In summary, the dependency graph search method, as adapted to non-modular code, found the sort function quite quickly. The VIFOR call graph tool was useful in this case because a single subroutine dominates the sort feature, but domain knowledge was needed to identify this SOOT subroutine as a renamed SORT.

Team C (“grep”) began with a search for the string “SORT”. This produced five hits on comments and several hits on the ISORT variable which, as previously mentioned, is identified in the documentation as a flag to enable sorting. One of the hits, in CTOBIN, showed that when this flag is set the SOOT subroutine is called. A quick look at comments in SOOT showed that it is the sort routine.

References

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. ASI Systems International, Fort Walton Beach, FL.

Arnold, R., Bohner, S., 1996. Software Change Impact Analysis. IEEE Computer Society Press, Los Alamitos, CA.

Biggerstaff, T., Mitbander, B., Webster, D., 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. In: IEEE Second Workshop on Program Comprehension July, 1993. IEEE Computer Society Press, Los Alamitos, CA, pp. 48–54.

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

Chen, K., Rajlich, V., 2000. Case study of feature location using dependence graph. In: Proceedings of the International Workshop on Program Comprehension, 2000. IEEE Computer Society Press, Los Alamitos, CA, pp. 241–249.

Chen, Y.-F., Nishimoto, M., Ramamoorthy, C.V., 1990. The C information abstraction system. IEEE Transactions on Software Engineering 16, 325–334.

Cleveland, L., 1989. A program understanding support environment. IBM Systems Journal 28 (2), 324–344.

Jones, S.L., Aitken, E.D., 1994. Convert3.0 User’s Manual. ASI Systems International, Fort Walton Beach, FL.

Lakhotia, A., 1993. Understanding someone else’s code: analysis of experience. Journal of Systems and Software 32 (December), 269–275.

Letovsky, S., Soloway, E., 1986. Delocalized plans and program comprehension. IEEE Software 3 (3), 41–49.

Queille, J.-P., Voidrot, J.-F., Wilde, N., Munro, M., 1994. The impact analysis task in software maintenance: a model and a case study. In: Proceedings of the IEEE International Conference on Software Maintenance—1994. IEEE Computer Society Press, Los Alamitos, CA, pp. 234–242.

Rajlich, V., Damaskinos, N., Linos, P., Silva, J., 1988. Visual support for programming-in-the-large. In: Proceedings of the International Conference on Software Maintenance. IEEE Computer Society Press, Los Alamitos, CA, pp. 92–99.

RECON2—tool for C Programmers, 2001. Available from <http://www.cs.uwf.edu/~recon/>.

Sim, S.E., Clarke, C.L.A., Holt, R.C., 1998. Archetypal source code searches: a survey of software developers and maintainers. In: Proceedings of the International Workshop on Program Comprehension, 1998. IEEE Computer Society Press, Los Alamitos, CA, pp. 180–187.

Turver, R.J., Munro, M., 1994. An early impact analysis technique for software maintenance. Journal of Software Maintenance 6 (1), 35–52.

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

Weiser, M., 1982. Programmers use slices when debugging. Communications of the ACM 25 (7), 446–452.

Wilde, N., Casey, C., 1996. Early field experience with the software reconnaissance technique for program comprehension. In: Proceedings of the IEEE International Conference on Software Maintenance—1996. IEEE Computer Society, Los Alamitos, CA, pp. 312–318.

Wilde, N., Huitt, R., 1991. A reusable toolset for software dependency analysis. Journal of Systems and Software 14, 97–102.

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

XSUDS—Telcordia Software Visualization and Analysis Toolsuite, 2001. Available from <http://xsuds.argreenhouse.com/>.

Yau, S.S., Collofello, J.S., MacGregor, T., 1978. Ripple effect analysis of software maintenance. In: Proceedings of Compsac 78. IEEE Computer Society Press, Los Alamitos, CA, pp. 60–65.

Norman Wilde is a full professor of computer science at the University of West Florida in Pensacola, FL. His research interests include software maintenance and program comprehension with a special emphasis on dynamic analysis techniques such as software reconnaissance. Dr. Wilde received his Ph.D. in mathematics and operations research from the Massachusetts Institute of Technology.

Michelle Buckellew is a software engineer at Lockheed Martin Integrated Systems in Orlando, Florida. Her work involves developing software for the Joint Air-to-Surface Standoff Missile (JASSM). She received an M.S. in software engineering from the University of West Florida.

Henry Page is a senior software engineer at Micro Systems, Fort Walton Beach, Florida. He has over 20 years Fortran programming experience, including 17 years with target vulnerability and weapons effectiveness simulations. He has a Bachelors of Applied Science from Troy State University and a Masters of Software Engineering from the University of West Florida.

Vaclav Rajlich is a full professor and former chair in the Department of Computer Science at Wayne State University. His research interests include software change, evolution, comprehension and maintenance. Among many other projects, his research group was the developer of the original VIFOR browser for Fortran programs in the late 1980s. Rajlich received a Ph.D. in mathematics from Case Western Reserve University.

LaTreva Pounds is a modeling and simulation engineer at Dynetics, Inc., Fort Walton Beach, Florida. She has a B.S. from Troy State University and is a Software Engineering M.S. degree candidate at the University of West Florida. She recently completed a thesis on Testing Distributed Object Oriented Software.