Locating Features in Vim:
A Software Reconnaissance Case Study

Andrew Fantozzi
University of West Florida

For:
CEN 6015
Dr. Norman Wilde, Instructor
November 6, 2002

Abstract

Software Reconnaissance is a technique for locating features in code. It is a process inspired by frustrated maintainers of old code. The University of West Florida has developed a tool called RECON written for this purpose.

In this study, the current version of the toolset (RECON3) is applied to the text editor VIM, a clone of the popular Unix editor VI. Three features of the program are successfully located using the RECON3 tools.

While none of the code is located without some thought and study on the part of the software engineer, code is discovered through development of effective test cases and by surveying trace files generated by running instrumented target programs.

This study is unique in that the author is a graduate student using RECON3 for the first time, and not a professional software engineer. As a result, the reader gets an accounting of exactly what it takes to download this tool and use it.

The RECON3 toolset was found to be effective and useful for Software Reconnaissance.

Introduction

Software Reconnaissance is a technique to help maintainers find features in unfamiliar code [Wilde.95]. In this paper, I describe the application of the software reconnaissance tool RECON3 to the popular text editor "Vim" (VI improved). Graduate students at the University of West Florida have developed the RECON3 toolset over the past ten years under the direction of Dr. Norman Wilde. This study is an excellent test of the software as Vim is nearly double the size of the next largest program RECON3 has been tested on.

Software Reconnaissance with RECON3

Software reconnaissance using RECON3 is composed of three main steps: instrumenting and compiling the source code, running the instrumented code to produce traces of the program, and analyzing the trace files to draw conclusions about the program being analyzed.

In order to instrument the source code (see figure 1-a), it is fed into a language specific RECON3 instrumentor. Currently, instumentors exist for C/C++ and Fortran, with plans to include Ada and Java in the future. This tool adds function calls into the target code to functions located within r3ctmi.c, a C file in the RECON3 source code. The instrumented code is then compiled using any standard compiler (GCC in this study). At run time, when these functions in r3ctmi.c are called, data are generated concerning code features such as function entry and exit points, decisions, and exit calls. These data are placed in shared memory and subsequently gathered up and written to a trace file by the trace manager.

The second step is accomplished by running test cases designed to isolate the feature that the maintainer seeks to locate on the instrumented program. Past studies have shown that most features can be located with one or two well thought out test case pairs [WILD.02]. In order to attain optimum results, the test case pairs should be as simple as possible, with each pair consisting of one test implementing minimal functionality of the program, and a second test composed of exactly the same input as the first test, plus implementation of the feature the maintainer is seeking to locate [WILD.02]. Therefore, the source code containing the feature should be locatable by

taking the set of code executed in test two and subtracting the set of code executed in test one.

The third step is to analyze the trace files that have been generated and to draw conclusions about the location of pertinent source code in the target program. This is done with the help of the Trace Graph tool. This program graphically displays the trace data contained in a trace file. A red rectangle is displayed each time a line of code executes for the first time and a black rectangle is displayed each subsequent time (see figure 2).

It should be noted that this software reconnaissance process requires several tools from the RECON3 toolset, the use of which must be learned. After a brief period of acclamation, however, the RECON3 tools were found to be intuitive and dynamic. They are most helpful in situations where little is known about the target program because they locate starting points for manual examination of the target code rather then the exact location of any desired feature. The effectiveness of the RECON3 toolset can be directly linked to the software engineer's skill in developing test cases to be run on the instrumented code.

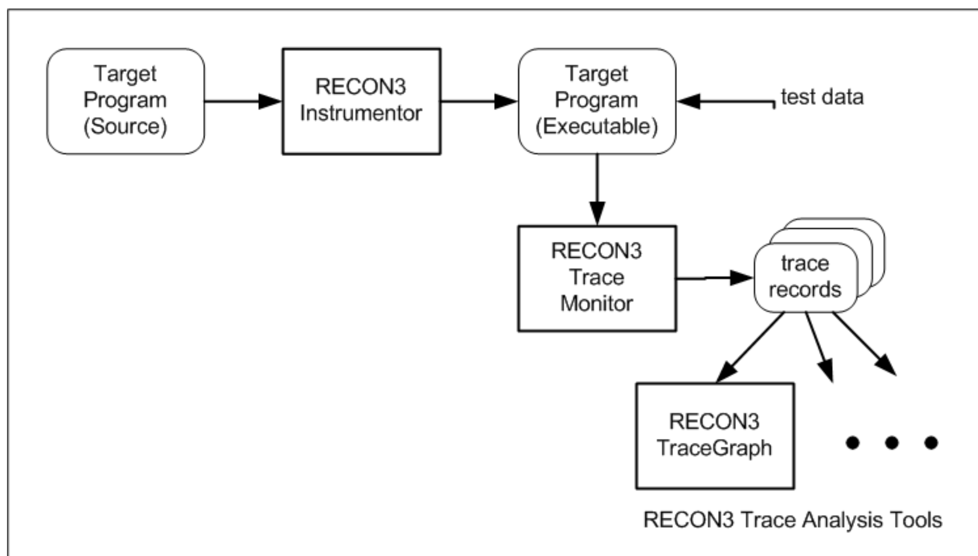

Figure 1.

A simplified diagram of the RECON3 software reconnaissance process

Building the RECON3 Tool Set

In order to install RECON3 on a UNIX account at the University, a compressed file was downloaded from the RECON3 web site onto a Windows PC, then FTP'ed into Unix. In order to avoid corrupting the files, one must ensure that a *binary* connection is used to FTP the files.

Once the file is transferred into Unix, one must then expand it and set about modifying the r3.h file as directed by README.txt. R3.h is a header file that is included in all compilations of instrumented code and contains user settings. Among these, the user must choose between the basic and the extended architecture. For a preliminary run through, the basic architecture was selected. This architecture creates trace records that are immediately written to a trace file. This is as opposed to the extended architecture, which generates data that are written to shared memory and later written to trace files by the trace manager. Furthermore, the extended architecture also allows the software engineer to control tracing on the fly and to create traces of multithreaded target programs. Though the extended architecture would be used on Vim, the basic architecture was selected as a way to become familiar with the program without getting into the advanced features at first.

Typically, RECON3 would decompress into its directory structure with the Unix *tar* command. In this case however, the tar file could not be expanded. The university Systems Administrator diagnosed the problem as a Unix configuration issue. It was determined that the account that the program was being installed on was still listed as an undergraduate student account on the Unix system, and therefore allocated only a negligible amount of disk space, which prevented expansion of the file.

A second problem was encountered during the compilation of the basic architecture. Several error messages were returned during the compilation process, so it was assumed that something was still awry with the expansion of the compressed downloaded file. Several C files that make up RECON3 were listed as missing or unavailable. By consulting with Dr. Wilde, it was determined that RECON3 was attempting to include all of the files that make up RECON3, even those not required to build the basic architecture. Therefore, the error messages could be ignored because none of the missing files were required.

After the issue of the Basic Architecture was resolved (by ignoring the error messages), the final step of the build process was to instrument a small calculator program written in C to get the feel of the C instrumentor.

The calculator program was an undergraduate programming assignment written by the author in a C programming class. It contains less than 100 lines of code and has one function each for addition, subtraction, multiplication, and division. It also has a function for collecting input and one for a user menu. This program was a perfect learning tool for the RECON3 toolset because it was simple to understand and was well known to the author as RECON3 beginner. As such, it was much easier to see how the target code was being instrumented and how effectively target functionality could be located.

The Target Program: Vim

According to its creator(s), Vim is "a highly configurable text editor built to enable efficient text editing [MOOL.02]." It is an improved version of the VI editor distributed with most UNIX systems, which is what the name Vim stands for: VI i [M] proved. It is widely popular as a programming editor and considered by some to be a complete integrated development environment. Written in the Netherlands by Bram Moolenar, it is also highly regarded for its portability, running on Windows, Unix, Atari, Macintosh, Dos, MachTen, OS/2, RiscOS, and VMS [MOOL.02]. The source code consists of over 240,000 lines of code, and is well documented.

Building Vim

In order to use RECON 3 on Vim, the C code that makes up the program would have to be instrumented. Therefore, two versions of Vim would have to be created, one instrumented with the RECON C Instrumentor, and one uninstrumented. Maintaining these two versions allows for comparisons to be drawn between the two versions of the code. In addition to these, an original built version of Vim was kept that was not manipulated at all. These three folders were named */Inst, /Uninst,* and */Orig*. In order to set up these files, it was assumed that Vim could be decompressed into the /*Orig* directory and then the decompressed directory structure could simply be copied into

/*Uninst* and */Inst*. It was not known that in so doing, the Unix cp command updated the date of each file copied to the current date by default. Then, when attempting to build the version of Vim in the */Uninst* file, the compilation would fail when make attempted to compile only those files that had changed since the current version publish date. The solution was to merely recopy the files using *cp* with the *–p* (preserve) modifier enabled, which preserved time stamps and other aspects of the files. The target program was then built using the Unix make command without further incident.

The next phase of the software reconnaissance was to instrument Vim. The first step was to verify the location of all of the C code required to building Vim. It was noted that all of the source code was located in the */src* directory.

Because the C Instrumentor works with a command line interface that instruments one C file at a time, a script is used when multiple source files are present. A tool in the RECON3 toolset called the Instrumentation Script Builder generates the script automatically. In the second step, this tool generated a script that ran the C Instrumentor on each source code file in the *Uninst/src* directory and placed a newly instrumented copy of each file in the *Inst/src* directory. It was discovered during the implementation of this step that Vim was written largely in non-ANSI C. Therefore, only the decisions could be instrumented, and not entries, returns, or exits. Additionally, the script copied (without instrumenting) any other non-C files in the *Uninst/src* directory to the *Inst/src* directory. This operation worked very well and no problems were encountered with the script file or r3Cinst.

The third step was to add the RECON3 source code files into the *Inst/src* directory that would need to be included in the compilation. These files were then added to the *Makefile* so that the instrumented Vim could be built using *make*. Because the *Makefile* contains code for building Vim on several platforms, it took careful review of the code to identify the correct place to insert the RECON3 files into the compilation scripts for the Vim files.

As will happen with complex unfamiliar software, the target program did not compile on the first attempt. The problem was solved by adding the *.o* (object) version of the RECON3 files to a list of object file names in the *Makefile.* Before a second attempt was made at compilation, the graphical user interface and X11 components of

Vim were also disabled in the *Makefile* in order to build a more compact version of Vim and to eliminate code that would not execute in this build.

Once the changes were made, the instrumented version of Vim compiled successfully without any warning messages. The only problem subsequently encountered was the help files not winding up in the directory that Vim expected them to be in at run time. This was only a minor inconvenience, as the help documentation is available online at www.vim.org.

Feature location 1

RECON3 was first used to locate code pertaining to the "u" (undo) command in the Vim editor. As stated earlier, the best sets of test cases are those that implement minimal functionality on the first test with each subsequent test implementing similar or identical functionality plus the desired feature of the target code. With that in mind, test case 1.1a consisted of three sentences of text. Once the last word was entered, the cursor was backspaced to a random letter and the "x" command (delete a character) was executed.

It was necessary to execute a command that could be undone with the "u" command in test 1.2a. The "x" command was chosen specifically because it was another one-character command. It was assumed that by choosing another one-character command that command-parsing functions would not be added to the search locations for target code.

Test 1.2a was identical to test 1.1a, but after the "x" command was executed, the "u" command was executed to undo the deletion. The resulting trace files were then analyzed using Trace Graph. Both files were much larger and had far less in common than expected. The difference was so significant that the decision was made to rerun the test cases with one important difference. For the second round of test cases, the text file was created ahead of time and the "x" command in test 1.1b and the "x" and "u" commands in test 1.2b would be isolated from the text entry functionality.

The second round of tests proved much better at isolating code for the "u" (undo) function. Approximately 10% of the instrumented code that was executed in test 1.2b

was not executed in test 1.1b. This was a vast improvement over the first set of test cases in which over 40% of the instrumented code was executed in test 1.2a and not in 1.1a.

In order to further reduce the quantity of trace data and further isolate the "undo" code, the trace data was then examined in a text editor. It was noted that nearly half of the lines of trace data referenced the file *screen.c.* A review of *screen.c* revealed that it contained the code responsible for displaying the text on the screen and storing text in a tree data structure so that only altered or newly generated text would be regenerated to the screen.

```
/*
 * screen.c: code for displaying on the screen
 *
 * Output to the screen (console, terminal emulator or GUI window) is minimized
 * by remembering what is already on the screen, and only updating the parts
 * that changed.
 *
 */
```

It was concluded that though this code was called often during the test cases run, it was not directly related to the "undo" functionality. In order to eliminate *screen.c* from the test cases, the uninstrumented version of *screen.c* was substituted for the instrumented version of *screen.c* and Vim was remade.

Test cases 1b and 2b were then rerun and the anticipated results achieved. The trace files were reduced in length by nearly half and became much more manageable. The trace files were reexamined using Trace Graph. The following figure shows a screen capture of the four test cases. The red boxes in the rightmost (fourth) column indicate lines of code containing decision statements being executed for the first time when compared against the boxes in the third. The boxes in the first and second column are from tests 1a and 2a, and do not factor into our analysis.

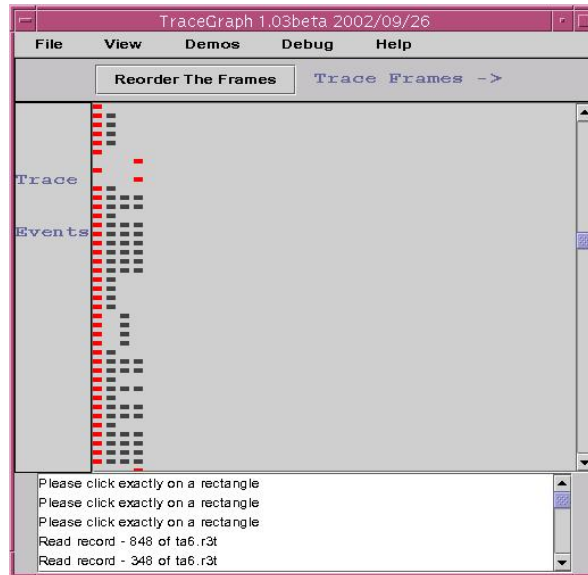

Figure 2.

A cursory examination of the red boxes in the fourth column revealed that over half represented code located in the *undo.c* source file. *Undo.c* was therefore an excellent candidate for further study. A text editor was then used to explore *undo.c* and it was readily apparent that it contained the code pertaining to the undo command. The following code fragment contains decision statements for saving executed commands:

```
/*
 * Save the lines between "top" and "bot" for both the "u" and "U" command.
 * "top" may be 0 and bot may be curbuf->b_ml.ml_line_count + 1.
 * Returns FAIL when lines could not be saved, OK otherwise.
 */
    int
u_save(top, bot)
    linenr_T top, bot;
{
    if (undo_off)
        return OK;

    if (top > curbuf->b_ml.ml_line_count ||
                            top >= bot || bot > curbuf->b_ml.ml_line_count + 1)
        return FALSE;   /* rely on caller to do error messages */

    if (top + 2 == bot)
        u_saveline((linenr_T)(top + 1));

    return (u_savecommon(top, bot, (linenr_T)0));
}
```

Study of the undo command revealed the extensive memory structure used to store executed commands. Vim is noted for its ability to undo and redo commands back and forth without losing any of the commands executed [MOOL.02].

Feature location 2

The second feature that was located in Vim's source code was the "/" (search forward) command. This tool is used to locate a string of text within a document. In order to search for code pertaining to search string, a simple text paragraph was again used. In accordance with lessons learned in the first feature location, the text file was created before the test case tracing was begun. Once the text file was opened with the instrumented version of Vim, the cursor was backspaced to a random letter and the "r" (replace letter) command was executed.

The second test case implemented the same (one letter) "r" command followed by the "/xi" command to search for the string "xi" (which was known to exist in the text.) As seen in this screen capture from Trace Graph, there are again extensive differences between the two traces despite the similarities of the tests.

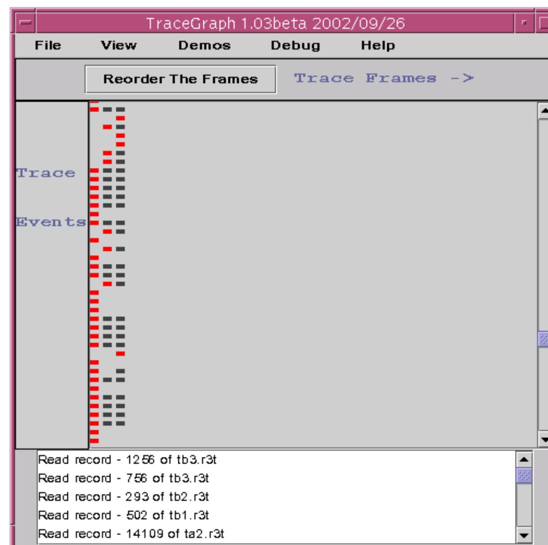

Figure 3.

As in feature location 1, there was a considerable amount of code executed in test 2.2 that was not executed in 2.1. Therefore, more would have to be done to isolate the search forward command.

A third test case (2.3) was added that executed the "w" command, which moves the cursor to the next word. The three trace files were then studied using Trace Graph. There still remained approximately 25% of the instrumented decisions executing in test case 2 that did not execute in test case 2.1 or 2.3. This was far less favorable than in feature location one, where the isolated code made up only 10% of the total code.

It was noted by sampling the red boxes that despite the quantity, one source code file turned up more often than any others, *search.c*. This was again an easy clue for a software engineer to follow. It stood to reason that such a source file would contain code relating to searching, and it did. The following code fragment contains the decision statement most often executed in the trace files generated:

```
 * Forward search in the first line: match should be after
 * the start position. If not, continue at the end of the
 * match (this is vi compatible) or on the next char.
 */
if (dir == FORWARD && at_first_line)
{
    match_ok = TRUE;
    /*
     * When match lands on a NUL the cursor will be put
     * one back afterwards, compare with that position,
     * otherwise "/$" will get stuck on end of line.
     */
    while ((options & SEARCH_END)
            ?  (nmatched == 1
                && (int)endpos.col - 1
                        < (int)start_pos.col + extra_col)
            : ((int)startcol - (ptr[startcol] == NUL)
                        < (int)start_pos.col + extra_col))
    {
        /*
         * If vi-compatible searching, continue at the end
         * of the match, otherwise continue one position
         * forward.
         */
        if (vim_strchr(p_cpo, CPO_SEARCH) != NULL)
        {
            if (nmatched > 1)
            {
                /* end is in next line, thus no match in
                 * this line */
                match_ok = FALSE;
                break;
            }
            matchcol = endpos.col;
            /* for empty match: advance one char */
            if (matchcol == startcol
                                && ptr[matchcol] != NUL)
            {
```

This code segment is a piece of a loop containing code that dictates the handling of null characters. Each space between words would require a decision to be made regarding null character interpretation. In order for a software engineer to modify the search functionality, every bit of *search.c* would have to be examined and understood, as well as all of the functions that interact with it.

Feature location 3

The third and final feature that was located in Vim's source code was the "dw" (delete word) command. This was perhaps the best test of RECON3's usefulness as any

software engineer worth his or her salt might have located the first two features based on file names alone.

Testing began with a text file consisting of three to four sentences of random text created before tracing began, as in the other two feature locations. Test case 3.1a executed the “d)” (delete remainder of line) command on a random line of text within the text document. Test case 3.2a then executed the same “d)” command followed by the “dw” command (the feature to be located in the code).

The trace files generated by tests 3.1a and 3.2a were then examined using Trace Graph. The results were difficult to interpret, as there were once again scores of instrumented decisions executing for the first time in trace 3.2a from ten or more different source files.

The tests were then refined and rerun to produce new trace files. Test 3.1b executed the “x” (delete character) command on the same simple text file on a random character. Then test 3.2b executed the “d)” (delete remainder of line) command, followed by a new test (3.3) which executed the “dw” (delete word) command which was the desired functionality.

This set of test cases differed from the previous two feature locations because the test 3.3 did not repeat the execution of the command in test 3.1b and/or test 3.2b. This departure from the pattern was not intentional, but wound up working anyway.

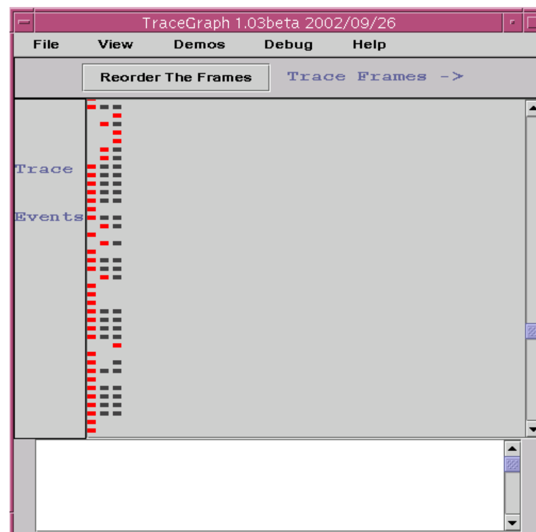

Figure 4.

As before, the trace files generated were still substantial and contained traces referencing instrumented decisions in several source files. Two of the instrumented source files stood out as possible locations for the delete word source code: getchar.c and memline.c.

Getchar.c was inspected first using a text editor. The comments at the beginning of the file indicated that it was not likely to contain code relate to delete word.

```
/*getchar.c
 *
 * functions related with getting a character from the user/mapping/redo/...
 *
 * manipulations with redo buffer and stuff buffer
 * mappings and abbreviations
 */
```

Memline.c was then examined in the same fashion and turned out to be the location of the source code.

```
/*
 * memline.c: Contains the functions for appending, deleting and changing the
 * text lines. The memfile functions are used to store the information in blocks
 * of memory, backed up by a file. The structure of the information is a tree.
 * The root of the tree is a pointer block. The leaves of the tree are data
 * blocks. In between may be several layers of pointer blocks, forming branches.
 *
 * Three types of blocks are used:
 * - Block nr 0 contains information for recovery
 * - Pointer blocks contain list of pointers to other blocks.
 * - Data blocks contain the actual text.
 *
```

This commented section left no doubt that memline.c contained the source code for delete word. A sample section of code located is as follows:

```
/*
 * See if it is the same line as requested last time.
 * Otherwise may need to flush last used line.
 */
    if (buf->b_ml.ml_line_lnum != lnum)
    {
        ml_flush_line(buf);

        /*
         * Find the data block containing the line.
         * This also fills the stack with the blocks from the root to the data
         * block and releases any locked block.
         */
        if ((hp = ml_find_line(buf, lnum, ML_FIND)) == NULL)
        {
            EMSGN(_("E316: ml_get: cannot find line %ld"), lnum);
            goto errorret;
        }

        dp = (DATA_BL *)(hp->bh_data);

        ptr = (char_u *)dp + ((dp->db_index[lnum - buf->b_ml.ml_locked_low]) &
DB_INDEX_MASK);
        buf->b_ml.ml_line_ptr = ptr;
        buf->b_ml.ml_line_lnum = lnum;
        buf->b_ml.ml_flags &= ~ML_LINE_DIRTY;
    }
```

This piece of code is part of a loop that locates text in a memory structure. This code also illustrates above average extents of the code documentation.

Conclusions

Above all, it was concluded that the RECON3 toolset is an effective system for locating features in source code. In three out of three cases, the author was able to locate the source code for the features selected in Vim. Of course there are factors that weight this study.

RECON3 was being used for the first time by a student programmer with no experience with source code larger than 1000 lines. This would seem to indicate that RECON3 must therefore be rather easy to use and fairly effective.

While it is true that RECON3 is effective, it was not easy to use at first attempt. There are several tools (the instrumentors, trace manager, trace graph, and the instrumentation script builder) which must be learned, all of which are written for software engineers without much "hand holding." However, once the user attains a degree of comfort with the tools, they are not difficult to use.

One must also consider the organization of Vim itself. Vim is exceptionally well documented and organized source code, especially considering that it is freely distributed. Another first time user might not be as fortunate, though neither of these mitigating factors should overshadow the positive results attained here.

Second, this study confirmed the attention to detail with which Vim was written. With over 240,000 lines of code, an no development budget, Vim is still neatly organized into approximately 100 source files, all of which contain extensive documentation.

Finally, this study highlighted the essential role that good test cases play in software reconnaissance. Not only must cases be simple, they must work together to isolate features in the code.

Bibliography

[GUND.95] Gunderson, Alan; Wilde, Norman; Casey, Christopher; "Locating Features in InterBase: A Software Reconnaissance Case Study at GTE Government Systems," report SERC-TR-77-F, Software Engineering Research Center, University of Florida, Gainesville, FL 32611, March 1995.

[MOOL.02] Moolenar, Bram, et al, "Vim FAQ," *Vim Online Doc: Vim FAQ*, October 2002, http://vimdoc.sourceforge.net/cgi-bin/vimfaq2html3.pl, (accessed 11/2/02).

[WILD.92] Wilde, Norman; Gomez, Juan A.; Gust, Thomas; Strasburg, Douglas; "Locating User Functionality in Old Code," *Proc., IEEE Conf. On Software Maintenance*, Orlando, November 1992. pp. 200-205.

[WILD.95] Wilde, Norman and Scully, Michael C., "Software Reconnaissance: Mapping Program Features to Code," *Journal of Software Maintenance: Research and Practice*, Vol. 7, No. 1, January 1995, pp. 49-62.

[WILD.96] Wilde, Norman and Casey, Christopher, Douglas; "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension," *Proc., IEEE Conf. On Software Maintenance*, Monterey, November 1996. pp. 312-318.

[WILD.02] Wilde, Norman, et al, "(various)README.txt," *The RECON3 Toolset - Descriptions*, March 2002, http://www.cs.uwf.edu/~recon/recon3/r3wToolSet.htm, (accessed 11/2/02).