Case study: Reconnaissance techniques to support feature location using RECON2

Suhaimi Ibrahim, Norbik Bashah Idris
Centre For Advanced Software Engineering,
Universiti Teknologi Malaysia,
Kuala Lumpur, Malaysia
(suhaimi,norbik)@case.utm.my

Aziz Deraman
Faculty of Technology & Information System,
Universiti Kebangsaan Malaysia,
Selangor, Malaysia
a.d@pkrisc.ukm.my

Abstract

Change requests are often formulated into concepts or features that a maintainer can understand. One of the main issues faced by a maintainer is to know and locate "where does this program implement feature X". However, these features are implicitly available in the code and scattered elsewhere that make them undoubtedly difficult to manage. A technique called software reconnaissance was originally inspired by industrial maintainers about the need for better ways of locating software features in large systems. This paper presents the authors' experience in using the software reconnaissance technique and tool called RECON2, developed by the university of West Florida. Our objective is to understand how the technique and tool work and to further suggest some enhancements with respect to software understanding strategies.

Keywords: *software understanding, change request, software reconnaissance, concept location, dynamic analysis.*

1. Introduction

In many organizations, maintenance tasks are quite costly and tedious to manage. More worse, documentations and other written materials are notoriously out-of-date and unreliable. Source code is considered the most reliable source of information available. However, the knowledge of interest for change is implicitly available in the code and scattered elsewhere that make it undoubtedly difficult to understand and locate. It is more likely that the functionality is coded as a *delocalised plan*. Soloway et al. have shown that maintainers can very easily mislocate and misunderstand such plans leading to serious maintenance errors as pieces of related code are physically located in non-contiguous parts of a program [1]. This makes code-level understanding a key activity in the maintenance task.

Software understanding or software comprehension is the process of recovering high-level, functionality-oriented information from the source code. Program comprehension is an essential part of software evolution and software maintenance: software that is not comprehended cannot be changed [2]. In the maintenance phase alone it has been estimated that programmers spend half or more of their time analyzing code or documents to try to understand the behavior of the system being maintained [3]. In particular, these programs have been maintained by many programmers with different programming styles over a number of years may be unnecessarily complex and difficult to manage.

Clearly, the software understanding process is an important activity so any approach towards assisting the comprehension can considerably reduce software costs. The study of software understanding is very important in order to know what are the elements of the knowledge required by the maintainers and how they construct a strategy towards achieving their objectives. Many researchers have proposed several cognitive models describing the comprehension strategies when understanding a program.

In bottom-up theory of program comprehension, the programmer's understanding is based on abstractions or chuncks of knowledge structures [4]. Chunks are parts of code that the programmer recognizes for example, "sort" numbers, "update" records, etc. These chunks are further aggregated into larger chunks representing higher level goals. So, large chunks contain smaller chunks nested within them. The programmer pieces together his understanding of the program by combining chunks into increasingly large chunks.

Soloway and Ehrlich [5] observed the programmer's understanding program in top-down strategy starting from the global structures of the program and refined further into a hierarchy of smaller abstractions until a complete goal is achieved. Top-down understanding requires some per-existing knowledge of the program in order to start exploration. Both bottom-up and top-down program comprehension theories are complementary and have been combined into unified models [6].

Rajlich [2] suggests a different view of program comprehension that does not rely on the top-down or bottom-up dichotomy, but one is based on the role of concepts. As programs have become larger, it has become ever less feasible to achieve complete comprehension. Instead, experienced programmers tend to use an "as-needed" strategy in which they attempt to understand only how certain specific concepts are reflected in the code [7,17]. In "as-needed" strategy, programmers work on a particular program task at hand and attempt to locate for certain knowledge of understanding based on program dependence and relationships. Data flows and control flows of program components are examined in order to search for concept or feature locations.

Reconnaissance technique was originally inspired by the industrial maintainers about the need for better ways of locating software features in large systems. It was a result of discussion and comments from the maintainers in handling a maintenance task. For example, work at the Bell Communications Research Centre where a large PBX telephone switch was maintained, Northrop-Grumman Melbourne Systems that built radar systems, etc. The maintainers indicated that one of their key problems was understanding where different features of the change requests implemented.

The concept locations can be used to handle the user's change requests. The change requests are often formulated into domain concepts or features that a maintainer can understand. One of the main issues faced by a maintainer is to know and locate "where does this program implement feature X". These features need a special technique and tool to locate. One technique that has been developed to help locate concepts in code is *Software Reconnaissance* [8]. In this paper, we want to present our experience in using a reconnaissance technique and its associated tool called RECON2 [9], developed by the university of West Florida.

Section 2 of the paper discusses the theoretical aspects of the concept location scenarios. Section 3 presents the software reconnaissance technique. A case study of GI system (Generate Index) is presented in section 4. Section 5 highlights some results of the case study followed by some lessons learnt. Section 6 presents some related work. Section 7 contains the conclusions and future work.

2. Concept location scenarios

In most software engineering processes, complete comprehension of the whole program is unnecessary and often is impossible especially for large programs [14]. Change requests often need to be formulated to some domain concepts or features that express the knowledge in terms of labels of program functionalities.

For example, "credit card" can be considered as a concept that is equivalent to object in an object-oriented program. There are concepts that are too trivial to have a class of their own. For example, the concept "discount" may be implemented as a single integer within class "sale" rather than having its own class.

Concept location is a process of locating a concept in the code and is a starting point for the desired program change. It is relatively easy to handle in small systems, which the programmer fully understands. For large and complex system, it can be a considerable task. The concept location assumes that the maintainer understands the concepts of the program domain, but does not know where in the code they are located.

For example, we want to change "a radio button selection window" found in a web browser application to a "pull down menu", we need to understand the concept of both "radio button" and "pull down menu" applications before hand. Then we can search for the location where the radio button selections are implemented in the code. During the course of locating a concept, the maintainers assimilate new facts that easily fit into their pre-existing knowledge.

Frequently in program comprehension the programmer understands domain concepts more than the code. Concept location is needed whenever a change is to be made. Let us consider the above GUI (graphical user interface) task with little problem extension.

"Change the radio button selection window to a pull down menu and apply it to credit card services to provide better views"

In order to make such required changes, the user must find in the code the location where the concepts of "radio button" and "credit card" are placed – this is the start of the change.

Based on this starting point, a programmer can explore some other program statements within the context of the same feature location. During the exploration, a programmer may need to branch to some other parts of the program in order to trace all the related statements for the desired concept. The related statements can be marked to indicate the boundaries from the unrelated statements.

3. Reconnaissane technique

Reconnaissance technique [11] is a dynamic search method rather than static search to locate concepts. Dynamic search involves the execution of the code with some test cases. Some prefer dynamic search as it is more focused and can extract most syntactic information that the static search may miss. The reconnaissance technique is based on the implementation of the instrumented program statements.

The instrumented statements are additional statements created to indicate which parts of the program conditions, for example if, while, case, etc were traversed during the execution of test cases. The target program is initially instrumented to put all "markers" executed at each condition. Then the target program is run with test cases to produce a set of markers of "with" desired feature.

The target program is run for second time with different set of test cases to produce another set of markers of "without" desired feature. The "marker" components can then be analyzed to locate the feature by taking the set of components executed in the "with" desired feature and subtracting the set of components executed in the "without" desired feature. The tests can be repeated for several time using different set of test cases for both "with" and "without" desired feature to ensure the focus of the feature location.

4. A case study

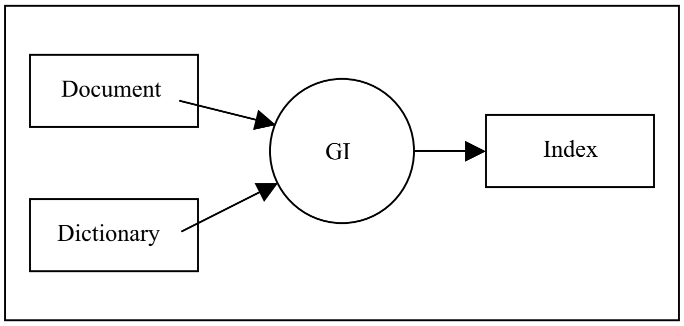
As part of reconnaissance study, we conducted a case study on RECON2 that applied to the Generate Index (GI) project, written in C. GI is a complete but 'crude' program of small size system (450 LOC) specially developed to train our M.Sc students working on software maintenance project. The idea of GI is to generate the indexes of document resemble to the reference indexes of our text books but with some slight variations. The change request was to incorporate the occurrences of an index on the same page as single indexes to appear in the output (index file). The feature location task was to find where the code associated to indexes located in the program. As first task, we had to understand the domain concept of GI.

4.1 Domain understanding of GI

GI works in Dos environment. It needs to be enhanced based on some change requests. The basic process of GI is shown in Figure 1. A text document, *doc* and a dictionary, *dic* were used as input to generate an index file, *indx* as depicted in the Listing 1a and 1b. GI is executed with the following command.

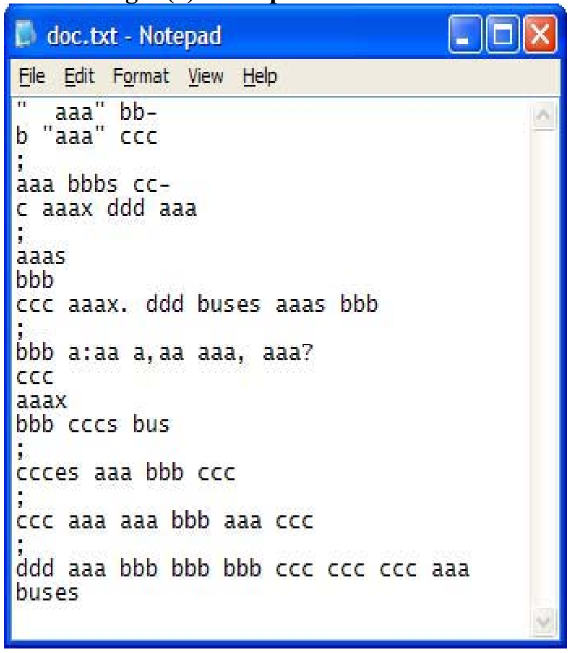
> *gi doc.txt dic.txt indx.txt err_file.txt*

Figure 1 : process of Generate Index



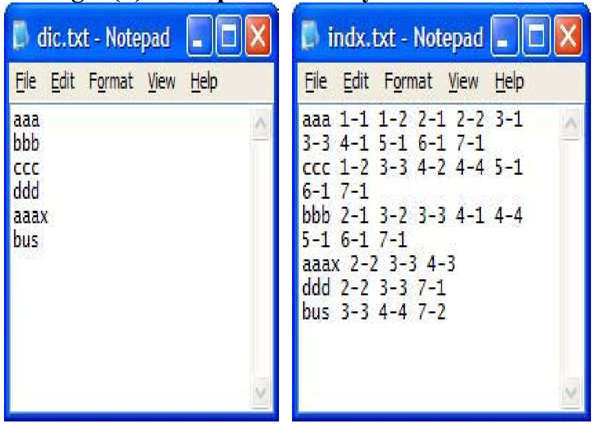
Each word in *doc* is examined of its occurrences against the words in *dic*. If they match then the corresponding word will be dumped into *indx* as an index. *Err_file* is just made available to capture errors if any abnormal situations occur, for example the input file is not found.

Listing 1 (a) : Sample document of GI



```
"  aaa" bb-
b "aaa" ccc
;
aaa bbbs cc-
c aaax ddd aaa
;
aaas
bbb
ccc aaax. ddd buses aaas bbb
;
bbb a:aa a,aa aaa, aaa?
ccc
aaax
bbb cccs bus
;
ccces aaa bbb ccc
;
ccc aaa aaa bbb aaa ccc
;
ddd aaa bbb bbb bbb ccc ccc ccc aaa
buses
```

Listing 1 (b) : Sample dictionary and index of GI



```
aaa
bbb
ccc
ddd
aaax
bus
```

```
aaa 1-1 1-2 2-1 2-2 3-1
3-3 4-1 5-1 6-1 7-1
ccc 1-2 3-3 4-2 4-4 5-1
6-1 7-1
bbb 2-1 3-2 3-3 4-1 4-4
5-1 6-1 7-1
aaax 2-2 3-3 4-3
ddd 2-2 3-3 7-1
bus 3-3 4-4 7-2
```

Note that the original GI program produces *indx* that consists of indexes, each followed by a page number and a line number. The subsequent occurrences of itself may proceed to different line numbers of within the same page or different pages.

The sample of the generated indexes is shown in the *indx* file (Listing 1b). For the sake of simplicity, the following symbols

'.' ':' ' ' ',' '"' ';'

in the *doc* (Listing 1a) are used as the *separators* between *words* with ';' as a page break. This program provides a trivial example of an index system that can be enhanced further.

The contents of output in *indx* reflects how the system works. For example, the word "aaa" occurs on on page 1-line 1, page 1-line 2, page 2-line 1, page 2-line 2, etc. The word "bbb" occurs on page 2-line 1, page 3-line 2, page 3-3, etc. while the index "bus" occurs on page 3-line 3, page 4-line 4 and page 7-line 2. Note that, the words "bus" or "buses" as appeared in *doc* are treated to be the same and should be managed by the index "bus" in *dic*. It just applies to singular words that end with 's', not others.

4.2 Feature identification

For our case study, we are dealing with the following change request.

Change the program to only consider a single occurrence of identical indexes if they occur on the same page in the document.

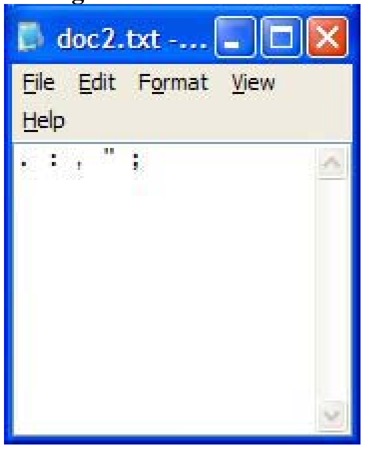
This means no more line numbers involved. We only consider a single occurrence of indexes on a page. Please take note that we are going to search for the feature location and impact of the intended features not the actual change.

Before using the RECON2, we need to understand the change request and dismantle it into some explicit features. Our first issue now is to identify what are the features could we extract from the change request. Based on the scenario described above we can derive the functionality as consisting of the following features;

Words and *separators*

The reason we say that is, from *words* we can derive the indexes and *words* would be of no use without the *separators* in between. However, we are interested in *words* as a feature not the *separators*.

Listing 2 : The N-Test Cases



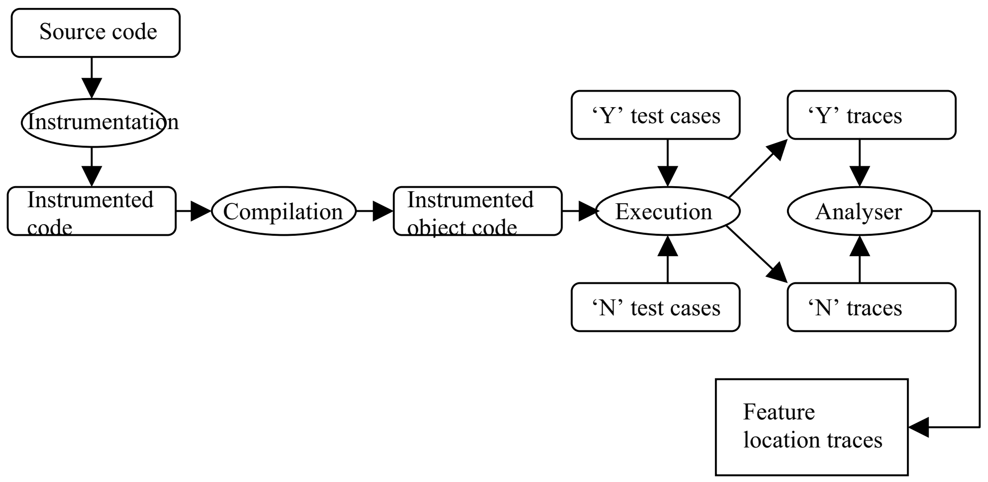
```
. : , " ;
```

But our first impression of the document as appeared in the *doc* seems that the contents are jumbled up with *words* and *separators*. *Words* are said to be the intended features, while *separators* are the unintended ones. Our attempt now is to get the *separator* location out of the document in the code. So, how do we manage this ?

One way to solve this is to design or construct two types of test cases, one is to identify all stuffs in the document (*words* and *separators*) and another one is to identify the *separators*. Then we can think of extracting the *separator* location from the document location.

Our second issue now is to design or contruct the test cases for both *document* and *separators*. This issue will be explained further in the following section.

Figure 2: The implementation of RECON2



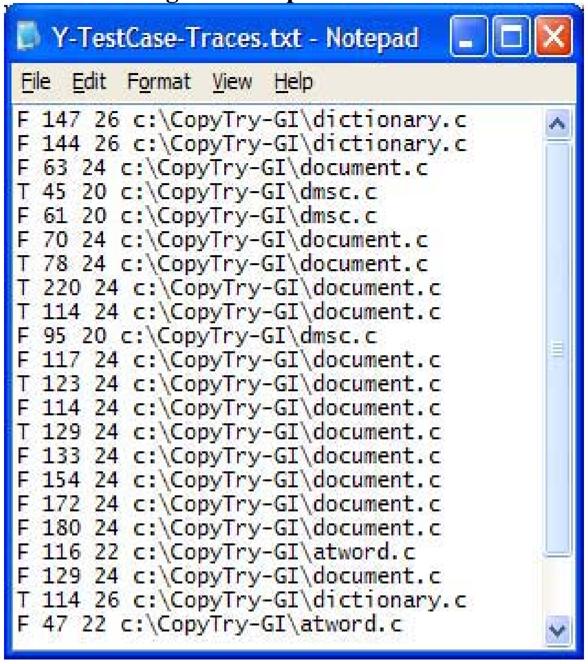
4.3 RECON2 approach

To support the above change request, we arrange our work into several subtasks using RECON2 (Figure 2).
- Perform instrumentation
- Test case selection
- Analyze traces

a. Perform instrumentation

Initially, RECON2 makes an instrumented copy of the user's target program before we implement a particular program feature. Instrumentation process adds to the original source code, some statements on each program condition such as if, while and case components so that the 'marker' components can be traced when analyzing a search.

Listing 3 : A snapshot of Y-traces



```
F 147 26 c:\CopyTry-GI\dictionary.c
F 144 26 c:\CopyTry-GI\dictionary.c
F 63 24 c:\CopyTry-GI\document.c
T 45 20 c:\CopyTry-GI\dmsc.c
F 61 20 c:\CopyTry-GI\dmsc.c
F 70 24 c:\CopyTry-GI\document.c
T 78 24 c:\CopyTry-GI\document.c
T 220 24 c:\CopyTry-GI\document.c
T 114 24 c:\CopyTry-GI\document.c
F 95 20 c:\CopyTry-GI\dmsc.c
F 117 24 c:\CopyTry-GI\document.c
T 123 24 c:\CopyTry-GI\document.c
F 114 24 c:\CopyTry-GI\document.c
T 129 24 c:\CopyTry-GI\document.c
F 133 24 c:\CopyTry-GI\document.c
F 154 24 c:\CopyTry-GI\document.c
F 172 24 c:\CopyTry-GI\document.c
F 180 24 c:\CopyTry-GI\document.c
F 116 22 c:\CopyTry-GI\atword.c
F 129 24 c:\CopyTry-GI\document.c
T 114 26 c:\CopyTry-GI\dictionary.c
F 47 22 c:\CopyTry-GI\atword.c
```

b. Test case selection

Our strategy on test case selection is to identify the actual locations or boundaries of the needed codes in the program. Two feature locations are discussed here

i) document location

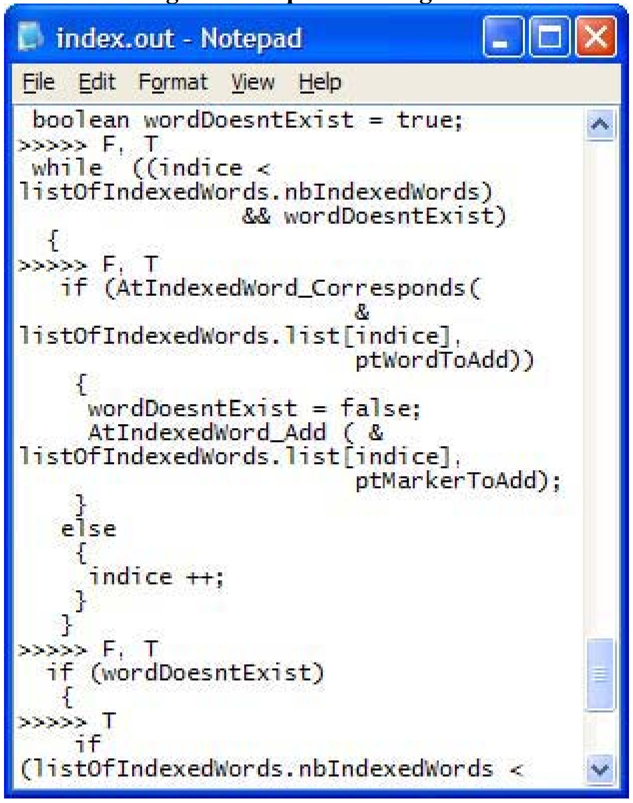
ii) separator location

From the above discussion, we recognized the document as consisting of all sorts of *words* and *separators*. We firstly constructed a set of test cases that led to the implementation of code to cover all the *words* and *separators*. We called it *Y-test cases*. We then created another set of test cases that covered only the *separator*s. We called it *N-test cases*. We considered the *doc* in Listing 1a as the *Y-test cases* and the *doc1* in Listing 2 as the *N-test cases*.

The test cases should be specially designed as they will determine the location to be searched. Too many test cases may affect the accuracy of feature location as this makes the resulting traces more difficult to analyze. The least and well chosen test cases will be useful as it makes the search more focused and close to the needed code. So, we classified two types of test cases as
i) Test cases "with" the feature (*Y-test cases*).
ii) Test cases '"without" the feature (*N-test cases*).

RECON2 executes the *Y-test cases* to produce the *Y-traces* and executes the *N- test cases* to produce the *N-traces*. Both traces contain the status of program conditions that includes the Boolean values, line numbers or positions of the affected program conditions in the module, module pointers and the physical location of the module involved (Listing 3). The traces reflect the detailed execution of the test cases.

Listing 4 : A snapshot of target traces



```
 boolean wordDoesntExist = true;
>>>>> F, T
 while  ((indice <
listOfIndexedWords.nbIndexedWords)
                 && wordDoesntExist)
   {
>>>>> F, T
    if (AtIndexedWord_Corresponds(
                        &
listOfIndexedWords.list[indice],
                        ptWordToAdd))
      {
       wordDoesntExist = false;
       AtIndexedWord_Add ( &
listOfIndexedWords.list[indice],
                        ptMarkerToAdd);
      }
    else
      {
       indice ++;
      }
    }
>>>>> F, T
   if (wordDoesntExist)
    {
>>>>> T
      if
(listOfIndexedWords.nbIndexedWords <
```

c. Analyze traces

We use an analyser provided by RECON2 to analyze the difference between the *Y-traces* and *N-traces*. Listing 3 shows some sample traces of *Y-test cases*. Conceptually it takes an extraction of *N-traces* out of *Y-traces* then the *difference* will be the occurrence of *N-traces* that differs from the *Y-traces*. We can also perform the analysis on individual *Y-traces* and *N-traces* to see their tracing impacts on the source code modules.

During the analysis, the result of the *difference* traces are directly annotated into the source code modules by automatically placing the markers ">>>>>" on the affected program conditions. Each marker is followed by a symbol "T" or "F" or both "T F" (Listing 4).

The "T" indicates the current program condition always gives a true Boolean while implementing a test case. The "F" indicates the program condition is false i.e it never occurs. While the "T F" is to indicate that both Booleans apply.

The "T F" situations could occur when the tracing gives different values at different times while traversing the path. The behavior of the program during execution may cause to some path repetition or looping that would change the variable status especially when a maintainer uses many test cases of different types in one run.

5. Results

The Y-test cases generated a *Y-trace* file of 62 pages in size. While the N-test cases generated a *N-trace* file of smaller size of about 5 pages only. All the traces are based on the complexity of the test cases we use that might involve looping, branching and repetitions of program paths. We run a RECON2 analyzer to analyze the difference between the *Y-traces* and the *N-traces*. The results of the tracing analysis are shown in Table 1.

Table 1 : Result of RECON2 analysis

Files	No. of functions	Y-TEST CASES (No. of annotated marks)	N_TEST CASES (No. of annotated marks)	Diff.
mmims_main.c	1	4	4	-
dmsc.c	4	4	4	-
dictionary.c	3	11	11	-
document.c	3	16	15	3
atmarker.c	4	3	1	2
atword.c	4	5	3	4
atindexedword.c	5	5	-	5
index.c	3	6	2	5
TOTAL	27	54	40	19

The *diff* in the table 1 shows that the difference in terms of the number of traces how the *separator* is different from the *document*. The result is encouraging.

From the table 1, we found the total number of tracing markers ">>>>>" after extracting the N-test cases from Y-test cases is 19 as compared to the original traces, 54 of Y-test cases and 40 of N-test cases. This means a maintainer can reduce his effort by just examining those files affected by the markers rather than examining the whole parts of the program.

The file *mmims_main.c, dmsc.c* and *dictionary.c* were not affected by the intended feature location as no difference can be derived from Y-traces and N-traces i.e all the traces are common to both Y-traces and N-traces.

The number of traces in *document.c* had drastically reduced to 3 out of 15 of N-test cases and 16 of Y-test cases. It also seems that there were no *N-test case* traces found in the *atindexedword.c* after the implementation of N-test cases. This is due to the fact that no influence of the separators in the program file, so the feature *words* make full use of the program dependence and relationships in *atindexedword.c*.

In overall, we noticed that the traces in the affected files were greatly reduced from the original Y-test which means the search strategy is more focused.

Remarks and lessons learnt

Some points can be concluded with regard to the application of reconnaissance techniques and RECON2 tool.

i) Generally only a few test cases are needed if they are well chosen. It is important to make the test cases "with" the feature as similar as possible to the test cases "without" the feature to avoid accidentally including irrelevant components in the trace.
ii) Reconnaissance techniques are useful for a starting point of concept location and regression testing. As it involves the dynamic search of the program, it can focus the search process and reduce the time for code-level analysis.
iii) RECON2 provides some elements of supporting a program task at hand, "as-needed" strategy which is useful as a basis to handle a large program.

However, the drawback is the reconnaissance techniques are based on the test cases. Very often that the test cases cannot be easily designed or selected. Many functionalities or features may not be easily formulated into test cases. Furthermore, the set of "without" test cases were just not rich enough to exclude the unwanted branches.

Another issue is the use of software reconnaissance would expect a maintainer to have some pre-existing knowledge of the program and application. Without this knowledge would be almost impossible for a maintainer to construct and choose the best test cases possible.

6. Related works

Many researchers have been dealing with the change impact analysis and it seems beginning to establish since the last two decades. The glorious records are discovered in a collection of prestigious papers and bibliographies in [12]. Some dependence graph and slicing techniques such as *program dependence graphs* (PDG), *system dependence graphs* (SDG), *abstract system dependence graphs* (ASDG), etc contribute to the static and dynamic search strategies of impact analysis.

As change impact analysis deals with the estimation of the program size prior to change, the reconnaissance process takes the complementary action to provide a starting point to locate concepts of software change. In describing the feature location process, the cognitive models of program understanding are useful [2,4,5,6,13,14].

Lokhtia [10] concluded that partial comprehension of software is sufficient for practical maintenance work. Mayrhauser suggested that tools for performing partial comprehension will be helpful. Chen and Rajlich [14] develops a reconnaissance tool that incorporates both static and dynamic search of using top-down exploration.

This search expects a programmer to have some pre-existing knowledge of the program. The programmer has to decide on certain hypotheses in order to reach or locate the desired features. This exploration seems quite flexible although it is time consuming as the programmer has to walkthrough the program.

Wilde [15] developed a reconnaissance tool, RECON2 on the expectation to locate concepts based on "as-needed" strategy. He claims that the tool is faster as it can work automatically on the concept location based on test cases.

Agrawal [16] developed a system test called λSuds to incorporate understanding, debugging and testing. λSuds stores an execution trace, which records how many times each test has exercised a particular software component (functions, blocks, decisions, data flow association) and expects pre-existing knowledge of program understanding.

As the reconnaissance techniques can automatically execute the test cases for tracing and analyzing, it does not allow the maintainer to manoeuvre the search. In software understanding, the intervention of maintainers is still useful to a certain extent. The maintainers might want to skip or proceed with certain hypotheses and do forward or backward during searching process. So, our future work is to see the possibility of incorporating both dynamic and static analysis into the change impact process.

7. Conclusion and future work

We presented some mechanisms of dynamic analysis adopted by the reconnaissance techniques. Reconnaissance techniques are potential to locate features and focus on a search process. The ability to dynamically analyze the traces within program components greatly reduces the maintainer's work of manually searching for their discrepancies in the code.

The software understanding of "as-needed" strategy has a potential to support the code-level maintenance of large system as it can focus on a program task at hand. However, it expects a maintainer to have some pre-existing knowledge of the program functionalities, otherwise the software change is almost impossible to implement.

Reconnaissance techniques can help provide a 'crude' estimate of feature 'size' which might be useful for cost estimation. Currently, we are working on the change impact analysis. The ability of reconnaissance techniques to search for feature location could be used to identify the size of the proposed change. Our study on reconnaissance gives a good insight into the dynamic analysis that will be useful in our future work.

References

[1] Soloway E., Pinto J., Letovsky S., Littman D., Lampert R.. "Designing documentation to compensate for delocalized plans", ACM, 1988, Volume 31, No. 11, pp. 1259-1267.

[2] Rajlich V., Wilde N., "The Role of Concepts in Program Comprehension", Proceedings of 10th International Workshop on Program Comprehension, 2002, IEEE, pp. 271-278.

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

[4] Letovsky S., "Cognitive Processes in Program Comprehension. In Empirical Studies of Programmers", E. Soloway and S. Iyengar, eds., 1986, Ablex, Norwood, NJ, pp. 58-79.

[5] Soloway E., and Ehrlich K., "Empirical Studies of Programming Knowledge", IEEE, 1984, SE-10, Volume 5, pp. 595-609.

[6] Mayrhauser A.V. and Vans A.M., "Industrial Experience with an Integrated Code Comprehension Model", Software Engineering Journal, Sept 1995, pp. 171-182.

[7] Koenemann J., and Robertson S., "Expert Problem Solving Strategies for Program Comprehension", Proceedings of conference on Human Factors im Computer Systems, CHI, ACM Press, May 1991, pp. 125-130.

[8] Wilde N., Scully M., "Software Reconnaissance: Mapping program features to code", Journal of Sofware Maintenance: Research and Practice, 1995, Vol. 7(1995), pp. 49-62.

[9] "RECON – tool for C programmers", http://www.cs.uwf.edu/~wilde/recon/

[10] Lakotia A., "Understanding someone else's code: Analysis of experience", Journal of System and Software, 1993, Vol. 23, pp. 269-275.

[11] Wilde N., Casey C., "Early Field Experience with the Software Reconnaissance Technique for Program Comprehension", WCRE, IEEE, 1996, pp. 270-276.

[12] Bohner S.A. and Arnold R.S., “Software change impact analysis”, IEEE Computer Society Press, 1996, Los Alamitos, California.

[13] Brooks R., “Towards a theory of the comprehension of computer programs”, International Journal of Man-Machine Studies, 1983, Volume 18, pp. 543-554.

[14] Chen K., Rajlich V., “Case study of feature location using dependence graph”, IWPC2000, IEEE, 2000, pp. 241-247.

[15] Wilde N., Page H., Rajlich V., “A case study of feature location in unstructured legacy Fortran code”, IEEE, 2001, pp. 68-76.

[16] Agrawal H., Alberi J.L., Horgan J.R., Ghosh S., Wilde N., “Mining system tests to aid software maintenance”, IEEE, 1998, pp. 64-73.

[17] Lukoit K., Wilde N., Stowell S., Hennesey T., “TraceGraph: Immediate visual location of software features”, IEEE, 2000, pp. 33-39.