Case Study of Feature Location Using Dependence Graph*

Kunrong Chen, Václav Rajlich
Department of Computer Science
Wayne State University
Detroit, MI 48202 USA
rajlich@cs.wayne.edu

Abstract

Software change requests are often formulated as requests to modify or to add a specific feature or concept. To implement these changes, the features or concepts must be located in the code. In this paper, we describe the scenarios of the feature and concept location. The scenarios utilize a computer-assisted search of software dependence graph. Scenarios are demonstrated by a case study of NCSA Mosaic source code.

1. Introduction

In software maintenance and evolution, change requirements are often formulated as requests to modify or to add specific program concepts or features [2, 26]. An example of such a request is “Add a new external viewer to Mosaic web browser”. Before any actual change can be made to the system, software programmers must locate the implementation of the concepts (“external viewer”) in the source code.

Concept location is a process that maps domain concepts to the software components. The input is the maintenance request, expressed in natural language and using the domain level terminology. The output of the mapping is a set of components that implement the feature or concept, see Figure 1. Feature or concept location is relatively easy in small systems, which the programmer fully understands. For large and complex systems, it can be a considerable task.

The difficulty of feature location is caused by several factors. One is that the input and output of the location process belong to different levels of abstraction: the input is in domain level and the output is in implementation level, see Figure 1. To make the translation from one level to another, extensive knowledge is required, including domain knowledge, programming knowledge, knowledge of algorithms and data structures, knowledge of the software components and their interactions, etc. This knowledge is hard to formalize and the programmer who has this knowledge must participate in the location process.

*) This research was partially supported by NSF grant # CCR-9803876

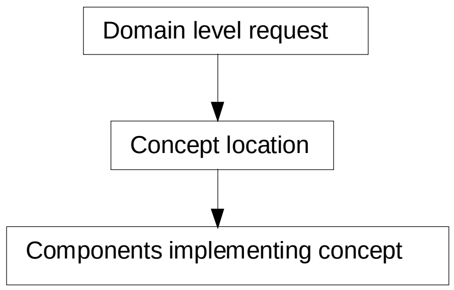

Figure 1. Concept location process

Another factor is the high cost of comprehension of large legacy systems, which makes a full comprehension of the program impractical and unnecessary [5, 12]. The programmers must limit the scope of their investigation and comprehend only parts, while making sure that the comprehension is sufficient for the maintenance task. They need a scenario that divides the comprehension into steps and gives feedback from each step. The feedback will help them to decide whether they are on the right track. This paper presents such a process.

We present a computer-assisted search process, with different and alternating roles for the computer and for the programmer. The process belongs to the category of “Intelligent Assistance” as advocated by Brooks [3] who claimed that “intelligence amplifying systems can, at any given level of available systems technology, beat AI systems. That is, a machine and a mind can beat a mind-imitating machine working by itself.” When dealing with feature location, we are convinced that a programmer assisted with tools can achieve more than a totally automated feature location tool.

Section 2 of the paper describes the dependence graph used for feature location. Section 3 investigates the theoretical aspects of the feature location scenarios. A case study of NCSA Mosaic software system is presented in section 4. Section 5 contains the references to the other work. Section 6 contains conclusions and future work. The Appendices present details of the case study.

2. Abstract System Dependence Graph

In [18], *procedure dependence graph* (PDG) was introduced. PDG is a graph representation of a procedure where vertices are statements or regions of the code. Data dependence edges represent possible flow of data in a procedure. Control dependence edges represent conditions on which a statement or region depends [10]. Every procedure has a special *entry* vertex representing the entry of the procedure. The control and data flow information is useful for data flow analysis, data flow testing, slicing, etc. [6]

A *system dependence graph* (SDG) [9, 10] is made up of several procedure dependence graphs. Additional mechanisms are defined to deal with function call and parameter passing. A *call* edge is added to connect the call and definition of a procedure. Algorithms are developed to construct SDG or PDG from a program [6, 7, 8, 9, 10, 18].

When dealing with feature location, the programmer usually does not need access to the statement level information. A higher level of abstraction of the program is more helpful. We propose an a*bstract system dependence graph* (ASDG) that can be constructed using a subset of the information of the SDG. The fact that ASDG is a subset of SDG guarantees that the algorithms used to construct SDG can be used to construct ASDG also.

In the C language, the ASDG consists of vertices that represent components, i.e. functions and global variables. Function call is represented by *call* edge and *data flow* edge represents flow of data from a function to a global variable and vice versa. Formally, we define ASDG in the following way: Let C be the set of components in the software, and C = F ∪ G where F is a set of functions and G is a set of global data. For d,e ∈ C, edge <d,e> denotes a dependence of component d on e. If d, e ∈ F then <d,e> is call edge, if d ∈ G or e ∈ G, then <d,e> is data flow edge. An ASDG is a set D of dependencies in the system.

The set of all components used in D is defined as comp(D) = {e | there exists d, such that <e,d> ∈ D or <d,e> ∈ D}. The neighbors of a vertex d are defined as neigh(d) = {<e,d> | there exists e such that <e,d> ∈ D} ∪ {<d,f> | there exists f such that <d,f> ∈ D}. Please note that we use the notation of [15, 21] for graphs.

An example of ASDG and its source code are in figure 2.

3. Feature Location Scenarios

In each step of the search, one component is chosen for visit. All visited components and their neighbors constitute a *search graph* [4, 17]. At the beginning, the search graph contains only the starting component. Each visit to a component expands the search graph, and the process continues until all the components implementing the feature or concept are located.

The search graph is the part of the ASDG that was visited during the search. There is always one active component selected for the next step. Formally, S is a search graph if and only if there exists d ∈ comp(S) such that s<d> ∈ S (d is selected component) and (S - s<d>) ⊂ D.

The tasks related to the search are divided into the programmer's tasks and the tool's tasks, see Figure 3.

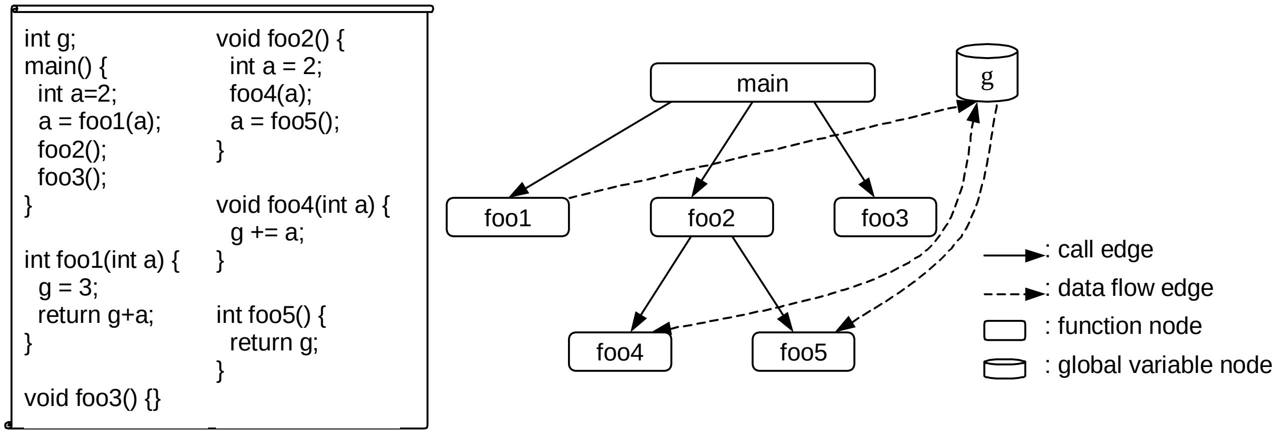
```
int g;                  void foo2() {
main() {                  int a = 2;
  int a=2;                foo4(a);
  a = foo1(a);            a = foo5();
  foo2();               }
  foo3();
}                       void foo4(int a) {
                          g += a;
int foo1(int a) {       }
  g = 3;
  return g+a;           int foo5() {
}                         return g;
                        }
void foo3() {}
```

Figure 2. A sample program and its ASDG

The programmer's role is to make decisions that direct the search. In particular, the programmer has to do the following:

Locate starting component: The assumption is that at the beginning, little is known about the system. The starting point is often the top component, i.e. function main(), because the top component summarizes all the requirements of the entire system. Other possible starting points for the search are the results of a dynamic analysis [24], or a component whose name is similar to the concept sought [1], or a randomly picked component.

Choose a component for visit: In every step, one component is selected for a visit and expansion of the search graph. The programmer explores the source code, dependence graph, and documentation, in order to understand the component and decide whether if it is relevant or unrelated to the feature.

Check if goal is reached: The programmer checks whether all components dealing with the feature have been found.

The supporting software tools do the following tasks:

Extract dependence graph of the system: Dependence graph is extracted from the source code by the program analyzer. Because the source code does not change during the feature location process, the analyzer is invoked only at the beginning.

Update the search graph: After the programmer visited a component, the tool will add it to the search graph. Based on the search graph, the programmer can backtrack, undo, or redo some of his/her previous operations.

The *scenario* of a feature location is a sequence of search graphs S_1, S_2, … S_n, which starts with a single-component search graph and ends with the feature located. At the beginning of the search, $S_1 = \{s<t>\}$.

If a step is not a backtracking one, the selected component will be investigated and the search graph will expand. Formally, $S_{i+1} = (S_i - s<c>) \cup \{m, s<d>\}$ where $m \in$ neigh(c) and either $m = <c,d>$ or $m = <d,c>$.

If the step is a backtracking step, that means current search direction is not promising and the user returns to a previously visited component. More formally, for backtracking step, $S_{i+1} = (S_i - s<c>) \cup s<d>$ where $d \in$ comp(S).

When programmer visits a component, he/she decides whether the component is related to the feature and whether to expand the search graph. There are several strategies of search graph expansion:

Top-down strategy [19, 20] expands search graph by called functions. The scenario starts with a function main() that summarizes the requirements for the whole program. If the sought functionality is not implemented directly there, the programmer recursively visits called functions until the desired functionality is found.

Bottom-up strategy is the opposite of top-down strategy and expands the search graph by calling functions.

Backward data flow strategy is employed when functionality of the system depends on specific values in specific variables. The programmer is searching for the origin of the values and visits the variables or functions that provide those values.

Forward data flow strategy is the opposite and the programmer is searching for the destination of the values.

The search is guided by the programmer's knowledge of the semantics of the components already visited. It can be divided into several subgoals.

Instead of direct search for a specific feature, the programmer searches for another closely related feature that is easier to find, and at the same time it is related in a predictable way to the feature searched. An example of this is in our case study.

.

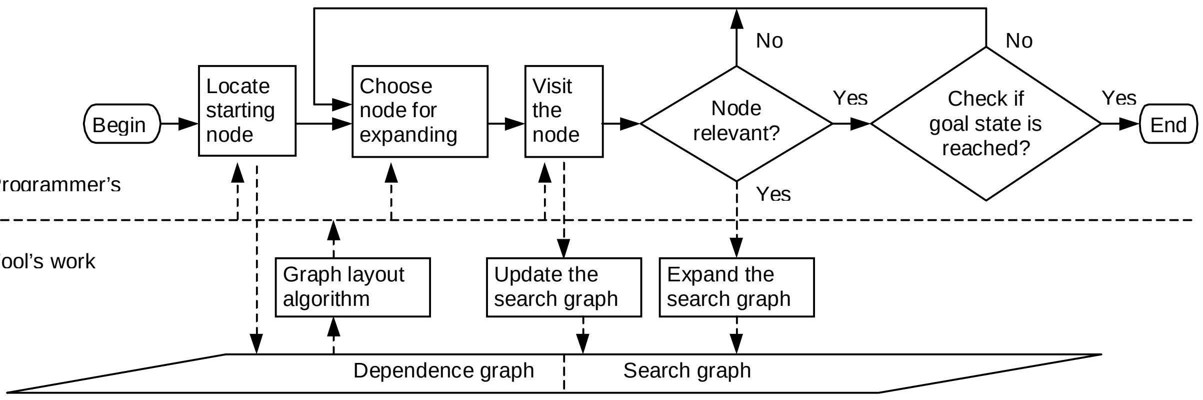

Figure 3. Feature location as a computer-assisted search

4. Case Study

As a part of our research, we conducted a case study based on NCSA Mosaic web browser for the X Window System Version 2.5 [16]. The change requirement was to add a new audio type to Mosaic. The feature location task was to find how Mosaic deals with video or audio files and to locate all components that manage file type and processor registration. As the first task, we had to understand the domain and its terminology.

Domain understanding: NCSA Mosaic uses the term “external viewer” for a program that Mosaic calls to view file formats it does not support internally, such as image or audio files. Mosaic browser retrieves information from two kinds of servers, leading to two different algorithms. One kind of server tells the browser the MIME (Multipurpose Internet Mail Extensions) type of the transferred data. MIME is an Internet standard, which is used to specify the type of the object being transferred across the Internet. Mosaic

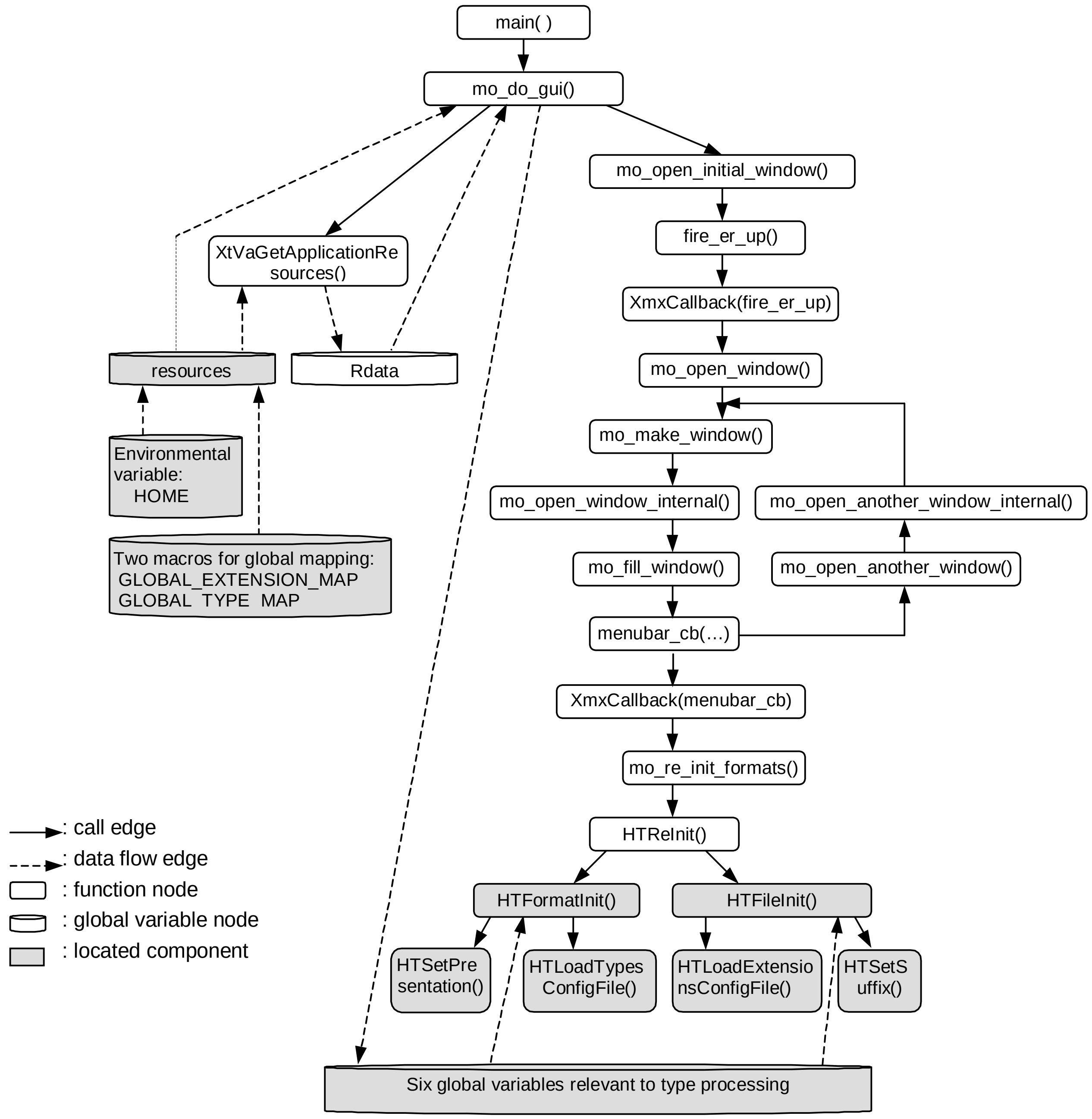

Figure 4. Search graph of Mosaic

uses "mailcap" file to specify the external viewer for the file. A mailcap file is a configuration file that maps file types to external viewers. One line in an example mailcap file looks like

video/mpeg; mpeg_play %s

where “video/mpeg” is the file MIME type, “mpeg_play” is a program that will view the file, and “%s” will be replaced with the actual file name. There can be a *global* mailcap and a *personal* mailcap files. Entries in the personal mailcap take precedence over entries in the global mailcap, which in turn take precedence over the built-in defaults. So if we want to change global and personal mapping, we just need to edit the file. For default mapping, we need to change the source code.

Some servers do not specify the file type of the document, which is being sent. In that case, Mosaic determines the incoming file type from the file name extension and invokes appropriate external viewer. Mapping between file types and extensions is kept in an extension map file. One line in an "extension map" file is

image/jpeg jpeg jpg jpe

where image/jpeg is a file MIME type, and jpeg, jpg and jpe are file extensions that will map to this file type. There are *global* extension map files and *personal* extension map files. Entries in personal extension map take precedence over entries in global extension map, which in turn take precedence over built-in defaults specified in source code.

Refined task, based on the domain terminology, is the following:

Locate in the code where

- the default, personal, and global mappings are set
- the type of incoming document is determined by protocol or by extension

The task is extensive and therefore it is divided into subgoals.

First subgoal is based on the fact that if we open a new window, it has the same browsing functionality as the original one. This implies that the new window has the same mappings as the old window and the mappings must be copied immediately after the opening. Therefore our first subgoal is to find the function that opens a new window.

For this, we adopt top-down strategy and start from function main(). The search graph from this case study is in Figure 4. Scenario of this phase is described in Appendix A, and through it we locate function mo_open_window() that is used to open a new window.

Second subgoal is to find where and how the mappings are set. The mappings are set after the window opens and before any document is loaded. Because this is a functionality connection, we continue top_down strategy, starting in mo_open_window(), see Appendix B.

After several steps of the search, we find ourselves in mo_open_window() again. Hence somewhere along the way, we must have taken a wrong turn and must backtrack. On the second try, we reach functions HTFormatInit() and HTFileInit() that deal with the settings and this completes the second subgoal.

The functions deal with six global variables, see Figure 4. However we need to know where the values of the six global variables come from and this is the **third subgoal** of the search. For that, we need to trace the data flows ending in the six global variables. Our strategy is backward data flow, see Appendix C.

The result of the scenarios is a partial comprehension of the system. Of the 984 functions in Mosaic, we visited only 22. Hence of all Mosaic code, we visited about 2%. In the scenarios, we moved 21 times from one function to another through function call, and moved backward through the data flows six times. The data flow based scenario was used when we dealt with a comprehension of specific data items, while control flow based scenario was used when we tried to understand a specific functionality or algorithm.

The actual change will follow closely the feature location scenario, particularly its third part.

5. Related work

In previous research, Biggerstaff et al. [1] defined "concept assignment problem" as the problem of “discovering human-oriented concepts and assigning them to their realizations”. He investigated the concept assignment process and claimed that because concept and program are not in the same level of abstraction, the human input is necessary. To perform concept assignment, a prior knowledge of the specific domain, a plausible reasoning, etc are needed. His conclusion is that totally automated tool for concept assignment is probably impossible, but some degree of automation is helpful. He performed case studies, where static analysis and both formal and informal information (for example names in the program) are used to locate concepts in the program.

Wilde et al. [24, 25] developed a program feature location technology called Software Reconnaissance, based on the analysis of test cases. The program is instrumented and two sets of test cases are established: the test cases that execute the feature, and the test cases without the feature. The feature location is determined by analysis of the two sets of event traces. This technique is very efficient for finding starting components in feature location, and it seems to be a natural candidate for a combination with our technique on the search of the static dependence graph.

Erdös and Sneed [5] confirm that it is unnecessary for a programmer to fully comprehend a program before maintenance. They proposed seven questions to be answered by the programmer before he or she can maintain the program. These questions are about domain knowledge, control flow, and data flow information.

Lakhotia [12] performed two case studies on practical systems: modification of GNU C Compiler (gcc) and modification of Wisconsin Program Integration Systems (WPIS). He used function call graph, and used command *grep*, *more*, and *emacs* editor. He described the feature location process and concluded that partial comprehension of software is sufficient for practical maintenance work.

Similarly as in [1, 5, 12, 24, 25], we assume that the program documentation - if it exists at all - does not support concept location and hence the concept location scenarios are necessary. That is the case for many practical systems.

von Mayrhauser and Vans [22, 23] studied processes of program comprehension and frequencies of individual actions of the programmers. They also suggest that a tool performing partial comprehension will be very helpful in maintenance work.

Littman et al. [14] investigate the strategies to use for feature location. They described two feature location strategies: the systematic strategy and as-needed strategy. To adopt a systematic strategy, the programmer needs to understand the behavior of the program totally before any change can be made. On the other hand, only the necessary part of the system is investigated. It is shown that the as-needed strategy does not provide sufficient knowledge for the programmer about the casual interactions of the programmer's functional components, thus often leads to unsuccessful modifications.

Jerding and Rugaber [11] use both the static and dynamic analysis in program understanding. They use static analysis to extract the system architecture and dynamic analysis to analyze the behavior of specific components and their interactions. Their method is to analyze the event trace and abstract the interaction pattern into various level of abstraction. They also

Concept or feature location can be used in different maintenance jobs, including program comprehension and software evolution. Location process is usually used with the change impact analysis. Change impact analysis [2] tries to estimate the size of the future change, while concept location just locates a concept in the code - related but different notions.

In dependence graph research, Ottenstein and Ottenstein [18] proposed PDG to represent program in software development environment. Horwitz, and Reps, [9, 10] extended PDG to SDG for program with multiple procedures. They also proposed SDG for program slicing, program differencing and program integration. Harrold et al. [7] proposed a method to construct PDG from abstract syntax tree of a program. Harrold et al. also extended SDG to object oriented programs and investigated methods for object-oriented program slicing [13].

6. Conclusion and future work

In our case study, we studied search scenarios for feature location, using the dependence graphs. The case study identified the requirements for an integrated supporting tool. The tool must be able to analyze the code and extract ASDG. Tool interface should display both the dependence graph and the search graph. Since the search graph is essentially a subset of ASDG, they both can be displayed in one layout. The user should be able to easily access the information for each vertex and edge, including both the code and the documentation. The tool should support varied search expansion strategies, including top-down or bottom-up and forward or backward data flow strategies. The user should be able to select the starting component, with function main() being the default. A tool that can perform all these tasks is being developed in our research lab.

We believe that the process of software change, of which feature location is a starting part, is an important frontier of maintenance and evolution research. The tools supporting feature location offer a promise of improving productivity of maintenance programmers and quality of the resulting code.

7. References

[1] T. Biggerstaff, B. Mitbander, and D. Webster, "Program Understanding and the Concept Assignment Problem," Communications of the ACM 37, No. 5, 72-83 (May 1994).

[2] S. Bohner and R. Arnold, "An Introduction to Software Change Impact Analysis", Software Change Impact Analysis, IEEE Computer Society, 1996.

[3] F. P. Brooks, Jr., "The Computer Scientist as Toolsmith – II", Computer Graphics, Vol. 28, pp. 281-287, November, 1994.

[4] E. Charniak, D. McDermott, “Introduction to Artificial Intelligence”, Addison-Wesley Publishing Company, 1984

[5] K. Erdös and H. M. Sneed, “Partial Comprehension of Complex Programs”, Proceedings of the 6th International Workshop on Program Comprehension, Ischia, Italy, June 1998, pp. 98-105.

[6] M. J. Harrold, B. Malloy, and G. Rothermel, “Constructing Program Dependence Graphs using a Parser”, ACM International Symposium on Software Testing and Analysis, 1993.

[7] M. J. Harrold, G. Rothermel, and S. Sinha, “Computation of Interprocedural Control Dependence”, Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'98), pp. 11-20, March 1998.

[8] M. J. Harrold, B. Malloy, “A Unified Interprocedural Program Representation for a Maintenance Environment”, IEEE Transactions on Software Engineering, vol.2, no.3, July 1993, pp. 270-285

[9] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs”, ACM Trans. Programming Languages and Systems, Vol. 12, No. 1, Jan. 1990, pp. 26-60

[10] S. Horwitz, T. Reps, “The Use of Program Dependence Graphs in Software Engineering”, Proceedings of the 14th International Conference on Software Engineering, May 1992.

[11] D. Jerding and S. Rugaber, “Using Visualization for Architectural Localization and Extraction”, Proceedings of the Fourth Working Conference on Reverse Engineering, October 1997, the Netherlands, IEEE Computer Society, pp. 56-65

[12] A. Lakhotia, “Understanding Someone Else’s Code: Analysis of Experience”, Journal of Systems and Software, Vol. 23, pp. 269-275 (1993).

[13] D. Liang, M. J. Harrold, “Slicing Object Using System Dependence Graph”, Proceedings of ICSM’98, November 1998.

[14] D.C. Littman, J. Pinto, S. Letovsky and E. Soloway, Mental Models and Software Maintenance, Proceedings of the Conference on Empirical Studies of Programmers, Albex, Norwood NJ, pp. 80 - 98, 1986

[15] D. Le Metayer, "Describing Software Architecture Styles Using Graph Grammars," IEEE Trans. On Software Engineering, 1998, pp 521-533.

[16] Mosaic web site (source codes and documents): http://www.ncsa.uiuc.edu/SDG/Software/Mosaic

[17] N. Nilsson, “Principles of Artificial Intelligence”, Morgan Kaufmann Publishers, Inc., 1980

[18] K. J. Ottenstein, and L. M. Ottenstein,. “The Program Dependence Graph in a Software Development Environment”, Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, 1984.

[19] V. Rajlich, “MSE: A Methodology for Software Evolution”, Journal of Software Maintenance, Vol. 9, 1997, pp.103-124.

[20] V. Rajlich, “A model for Change Propagation Based on Graph Rewriting”, Proceedings of ICSM’97, Bari, Italy, October 1997.

[21] V. Rajlich, "Theory of Data Structures by Relational and Graph Grammars," In Automata, Languages, and Programming, Lecture Notes in Computer Science 52, Springer Verlag, Berlin, 1977, pp. 391-511.

[22] A. Von Mayhauser and A. Vans, “Program Understanding Processes During Corrective Maintenance of Large Scale Software”, Procs. International Conference on Software Maintenance ’97, Sept. 1997, Bari, Italy, pp 12-30.

[23] A. Von Mayhauser and A. Vans, “Program Understanding Behavior During Enhancement of Large-scale Software”, Journal of Software Maintenance: Research and Practice, Vol. 9, pp 299-327 (1997).

[24] N. Wilde and T. Gust, “Locating user functionality in old code”, Proceedings of Conference on Software Maintenance 1992, Orlando, Florida, 1992, pp. 200-205.

[25] N. Wilde, Michael Scully, “Software Reconnaissance: Mapping Program Features to Code”, Software Maintenance: Research and Practice, Vol. 7, 49-62, 1995.

[26] S.S. Yau, R.A. Nichol, J.J. Tsai and S. Liu, “An Integrated Life-Cycle Model for Software Maintenance”, IEEE Trans. Software Engineering, 15(7), 1988, pp 58-95.

Appendix A: Search for window opening

Inspection of the function **main()** reveals that it calls several other functions, but only mo_do_gui() is relevant to our task, because it is the real main routine of the application. Other functions are small functions, used for exception handling, new process creation, socket communication initiation, and it is unlikely that they would deal with window opening.

Function **mo_do_gui()** does not deal with the window opening, but calls directly the functions mo_open_initial_window() and XtVarGetApplicationResources(), and accesses several global variables. The function mo_open_initial_window() seems to open a window directly or indirectly, so it is the next visited function.

Function **mo_open_initial_window()** sets a timeout that calls fire_er_up() after 10 milliseconds. We guess that fire_er_up() does the actual work.

Definition of function **fire_er_up()** is not obvious, but after macro expansion the definition of XmxCallback(fire_er_up) is actually the definition of fire_er_up(). It calls mo_open_window() with the home document URL as the argument.

Function **mo_open_window()** opens a window and views a given URL. This completes the first part of the search.

Appendix B: Search for mappings

Function mo_open_window() calls **mo_make_window()** to make a window from scratch. Function mo_make_window() creates an X window shell and then calls mo_open_window_internal() to open a browser window.

Function **mo_open_window_internal()** creates a structure of type mo_window, calls mo_fill_window(), and adds this window to the window list.

Function **mo_fill_window()** takes the structure and fills in data values, for example title, menu creation and setting. It calls several functions of X window library and also sets up the menu bar and pulldown menus. For every item in the pulldown menus, such as, “New Window”, it specifies the callback function as menubar_cb().

The definition of XmxCallback (menubar_cb) in the source code is actually the definition of **menubar_cb()** after macro expansion. This function calls a processing function for each menu item. We choose to search function **mo_open_another_window()**, which is matched to menu item “New window”.

This function opens a new window and loads the default document into it. It calls **mo_open_another_window_internal()** which calls mo_make_window() that was visited before. Hence we have to backtrack and try another branch.

We backtrack to **menubar _cb()** and try another direction: menu item “Reload Config Files” and the related function **mo_re_init_formats**().

Function mo_re_init_formats() is a simple function that calls function **HTReInit()**, which in turn calls three functions: HTFormatInit(), HTFileInit() and HTList.c::HTList_delete(). Of the three functions, HTList.c::HTList_delete() is used to delete a list and is unrelated to our goals. The remaining two functions will be inspected.

Function **HTFormatInit()** calls HTLoadTypesConfigFile() twice and HTSetPresentation() many times. Every time HTSetPresention() is called, it sets a default mapping between one MIME type and its processing program. In the source code, total of 38 default mappings are set. Most of them are commonly used pairs of MIME type and their processing programs. Function HTLoadTypesConfigFile() is called twice with global variable *personal_type_map* and *global_type_map* respectively. From the name of function and global variable, we infer that they establish personal and global type mapping, respectively. The calling sequence is default mapping first, then global mapping, and then personal mapping. This calling sequence decides the precedence of the three mappings. Global boolean variable *use_default_type_map* decides whether we call default mappings or not.

Function **HTSetPresentation()** adds one more mapping to the mapping list. This mapping is between a MIME type and its processing program.

Inspecting further **HTLoadTypesConfigFile()**, we find it reads type mapping file line by line and creates or updates mappings between file types and processing programs.

Inspection of HTFormatInit() and the functions it calls reveals that they establish the mappings between the file MIME types and their processing programs.

We expect it establish the mapping between extension and MIME type. Inspection reveals that Function HTFileInit() has the similar working mechanism as HTFormatInit(). It calls function HTLoadExtensionsConfigFile() twice and function HTSetSuffix() many times. Every call to HTSuffix() sets a default mapping between one extension and its MIME type. Function HTLoadExtensionsConfigFile() is called twice with global variable *personal_extension_map* and *global_extension_ map* respectively. From the name of function and global variable, we think they establish personal and global extension mappings, respectively. The calling sequence is default mapping first, then global mapping, and then personal mapping. This calling sequence decides the precedence of the three mappings. Global boolean variable *use_default_extension_map* decides whether we call default mappings or just skip them.

Function **HTSetSuffix()** adds one more mapping to the mapping list. This mapping is between file extension to its relevant MIME type.

Inspecting further function **HTLoadExtensionsConfigFile()**, we find it reads line from extension mapping file and creates or updates one mapping for every extension.

As the result of this part of the search, we located six functions HTFormatInit(), HTSetPresentation(), HTLoadTypesConfigFile(), HTFileInit(), HTLoadExtensionsConfigFile(), HTSuffix() that deal with the mappings. The values of the mappings originate from six global variables personal_type_map, global_type_map, use_default_type_map, personal_extension_map, global_extension_map and use_default_extension_map.

Appendix C: Search for source of mappings

The six variables are assigned in function **mo_do_gui()** by values of global data Rdata. The information flows to Rdata from function **XtVaGetApplicationResources()** which is called by mo_do_gui(). In this function call, variables "Rdata" and "resources" are two actual arguments. They store the same MOTIF resource information in different formats and the function converts the data between these two formats. The data flow is directed from "resources" to "Rdata".

When we inspect the data **resources**, we find all default values of the six variables. So the six global variables actually get their default values from here. Two macros and one environmental variable are used to specify the default values. They are macro GLOBAL_TYPE_MAP, macro GLOBAL_EXTENSION_MAP, and environmental variable HOME are used.

Finally, the located components that implement the feature consist of functions HTFormatInit(), HTSetPresentation(), HTLoadTypesConfigFile(), HTFileInit(), HTLoadExtensionsConfigFile(), HTSuffix(), and variables personal_type_map, global_type_map, use_default_type_map, personal_extension_map, global_extension_map, use_default_extension_map, resources, macros GLOBAL_TYPE_MAP and GLOBAL_EXTENSION_MAP, and environmental variable HOME, see Figure 4.