RIPPLES: Tool for Change in Legacy Software

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

Abstract

Key parts of software change are concept location and change propagation. We introduce a tool RIPPLES that supports both. It uses the Abstract System Dependence Graph (ASDG) of the program, enriched by conceptual dependencies. A case study of NCSA Mosaic demonstrates the use of the tool. Precision and recall are used to evaluate the quality of support provided by RIPPLES.

1. Introduction

Staged model of software lifecycle [19] partitions software lifecycle into initial software development, evolution, servicing, phase out, and close down. In this paper, we focus on software changes that take place during servicing stage. For most systems in this stage, the architecture is deteriorating, and the documentation does not reflect the source code. Such systems are usually called legacy systems. Full comprehension is expensive and almost impossible, and this substantially impacts the process of software change. As a result, usually only minor changes are done during this stage.

Software change starts with a maintenance request, which is typically expressed in terms of domain concepts or program features that have to be modified. The requested change may be corrective, perfective, adaptive, or preventive [3], but in almost all cases it is formulated in terms of domain concepts.

The first part of change process is feature or concept location (abbreviated "location"), which has to be done before any actual change can be made to the system. Location is a process that maps relevant domain concepts to the software components [7]. Location may be easy in small systems that the programmer fully understands, but it can be a considerable task in large and complex legacy systems.

As there are always many different implementations for one concept, it is very difficult, if not impossible, to implement a fully automatic feature locator. The maintainer often performs location manually with the help of simple tools such as "grep". In our previous research [7], we looked for a more systematic process and proposed a computer-assisted search of Abstract System Dependence Graph (ASDG), where the maintainer and the tool have alternating and complementary roles. The result of the process is the set of components that implement the concept.

The components implementing the concept are often related to each other by data flow or control flow or definition-reference relationship and it is a goal of static program analysis to extract these relationships as accurately as possible. However in some instances, there are additional dependencies that we call conceptual dependencies. Two components in a conceptual dependence have a connection that is not discovered by static code analysis, yet both participate in the same concept and a change in one may require a change in the other. One example from our case study is the connection between the menu item and its call back function. In our case study, conceptual dependencies play a significant role. They have to be discovered manually during location and added to the ASDG.

The maintainer starts software modifications after concepts are located and conceptual dependencies are established. Modification of a component may cause the system to be temporarily inconsistent as the require-provide relationships between the changed component and its neighbors are no longer valid. To fix this, secondary changes are introduced in the neighbors but they may cause new inconsistencies. This situation continues until the system becomes consistent again [18]. This process is called change propagation (abbreviated "propagation").

In this paper, we describe a new tool RIPPLES, developed for servicing legacy systems. It supports both location and propagation and combines automatic static code analysis with human intelligence, as advocated by Brooks [5]. Brooks 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.” Based on that philosophy, RIPPLES is a tool that helps the programmers in their maintenance work rather than replacing them. It is the task of the programmers, not of the tool, to acquire all necessary domain and programming knowledge. The programmers also make all decisions: choose the starting component, make changes, determine the end of the process, etc. The tool only assists in these tasks.

In section 2, we describe Abstract System Dependency Graphs. Section 3 describes the tool RIPPLES. Section 4 describes case study of a change in NCSA Mosaic. Section 5 analyzes the efficiency of the tool. Section 6 lists other work in the area. Section 7 concludes the paper. The appendixes report selected parts of the case study in more detail.

2. Abstract system dependence graph

The basic data structure of tool RIPPLES is *abstract system dependence graph* (ASDG) [7,26]. ASDG represents dependencies among software components. For C programming language, the vertexes are functions, function arguments, and global variables and types. The edges are data flows, control flows, and define-use relationships. ASDG is derived from finer granularity System Dependence Graph (SDG) [11,12] that represents programs on the level of statements.

The vertexes of SDG are functions, variables, types, arguments, and program statements. The edges represent control and data dependencies among them. The extraction of SDG from the code in procedural languages is discussed in [10,11,12]. Sets Def(s) and Use(s) contain variables defined or used in a statement s. For a function definition, a Format_in vertex represents a format argument, and for each possible modified format argument, an additional Format_out vertex is added. For a function call, a call vertex is added, Actual_in vertexes are created for each actual argument, and Actual_out for each possibly modified actual argument. The actual and formal arguments are connected by dataflow edges. Between Actual_in and Actual_out vertexes of one procedure, there may be summary edges, which model the transitive data flow inside the procedure call.

To construct an ASDG, we need to delete statement vertexes of SDG. If there is no function call in statement s and s does not define or use any variable, s and all related edges are deleted. If s does define and use variables, add data flow edges from each vertex in Use(s) to each vertex in Def(s), delete s and related edges.

If there is function call in s, three operations are done before deleting statement s. First, add data flow edges from each vertex in the Use(s) to each vertex in Def(s). Second, add a calling edge from the calling function to the called function. Third, remove Actual_in and Actual_out vertexes and related edges. Add new summary edge for transitive data flow if there is dataflow through Actual_in or Actual_out vertexes.

This process allows us to base the extraction of ASDG on SDG algorithms. ASDG is used for both concept location and change propagation.

2.1 ASDG in concept location

Location is a computer-assisted search process [7]. It is a step-by-step process where in each step one component is selected for investigation. To facilitate the process, marks are set on vertexes and edges. A vertex can be "unmarked" (default and initial value), "candidate" (for investigation), "visited" (promising component, further investigation in this direction is needed), "located" (part of the result) and "unrelated" (to the feature). Marks on edges denote search direction. Three possible values are "unmarked" (default), "forward" (same as edge direction) and "reverse".

The programmer can choose one of several search strategies: Top-down strategy expands selected vertexes by adding called functions. Bottom-up strategy is the opposite of top-down strategy and expands selected vertexes by adding calling functions. Backward data flow strategy is employed when functionality of the system depends on specific values in specific variables. The search proceeds towards the origin of the values. Forward data flow strategy is the opposite and the programmer is checking for the destination of the values. Default strategy expands selected vertex by all neighbors.

2.2 ASDG in change propagation

Marks can also be used to support propagation process. The marks on vertexes can be "unmarked" (default and initial value), "candidate", and "changed". The edges can be "unmarked" (default), "forward" or "backward". The mark on an edge means the direction of the change propagation. During propagation, ASDG changes as the maintainer adds or deletes components or edges from the system.

2.3 Screen graph

To support change in large and complex system, only

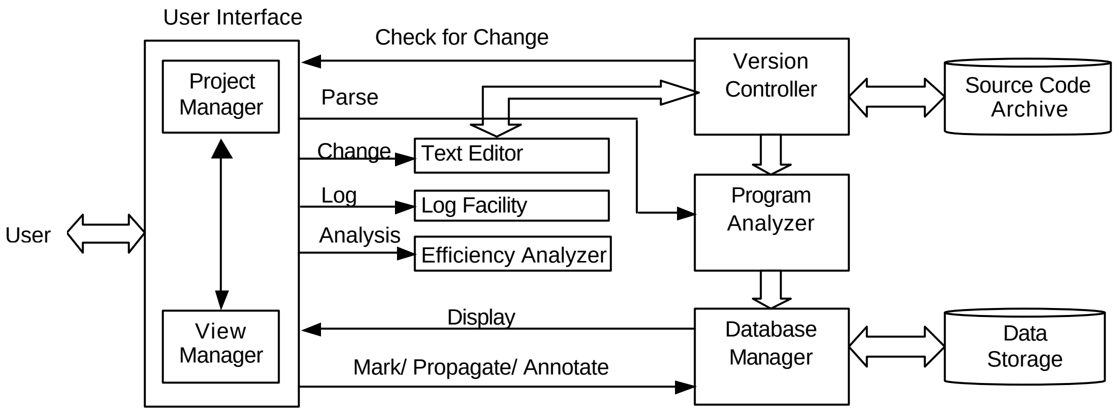


Figure 1. The Architecture of RIPPLES

a part of the ASDG is displayed on the screen. This part of the ASDG is called screen graph. Several operations manipulate it. Operation *expand* adds the neighbors of selected component and the related edges to screen graph. Operation *compress* hides neighbors of this component and all related edges from screen graph. Operation *hide* hides a component and all its edges. Its counterpart *display* adds any hidden component and its edges to screen graph. These operations do not change the underlying ASDG.

3. Tool RIPPLES

Tool RIPPLES extracts ASDG from C code and supports both concept location and change propagation. The architecture of RIPPLES is shown in Figure 1. RIPPLES is implemented in Tcl/Tk [22]. Figure 2 contains the user interface of RIPPLES.

The tool consists of the following major components:

Version controller manages the source code archive.

Project manager handles project information, file information, and environment setting, etc.

View manager controls zooming, layout, and event processing.

Log facility records all operations for instrumentation, efficiency measurements, and undo.

Efficiency analyzer measures efficiency of RIPPLES, see details in section 5.

Program analyzer parses the code, constructs ASDG and stores it in the database. It uses Datrix C/C++ parser [8].

Tool RIPPLES supports the following operations:

Location operations are *mark*, *unmark* and *locate*. At the beginning of location process, all components are "unmarked". As the location proceeds, the maintainer marks interesting components by operation *mark* that sets one component as a "candidate" for investigation. It can be applied to any component on screen graph. The marks can have different colors. Operation *unmark* is the opposite. Operation *locate* allows the user to label the vertexes as "unrelated", "located", or "visited".

Pattern matching for vertex names is also provided. The programmer specifies the vertex name and the name of the file it belongs to. Wild card "*" (any length of character combination) or "?" (any one character) are allowed in the query. All vertexes whose names match the description will be part of the result.

Propagation is supported by operations *change*, *add component*, *delete component*, and *skip*. The first three operations change the actual ASDG of the system.

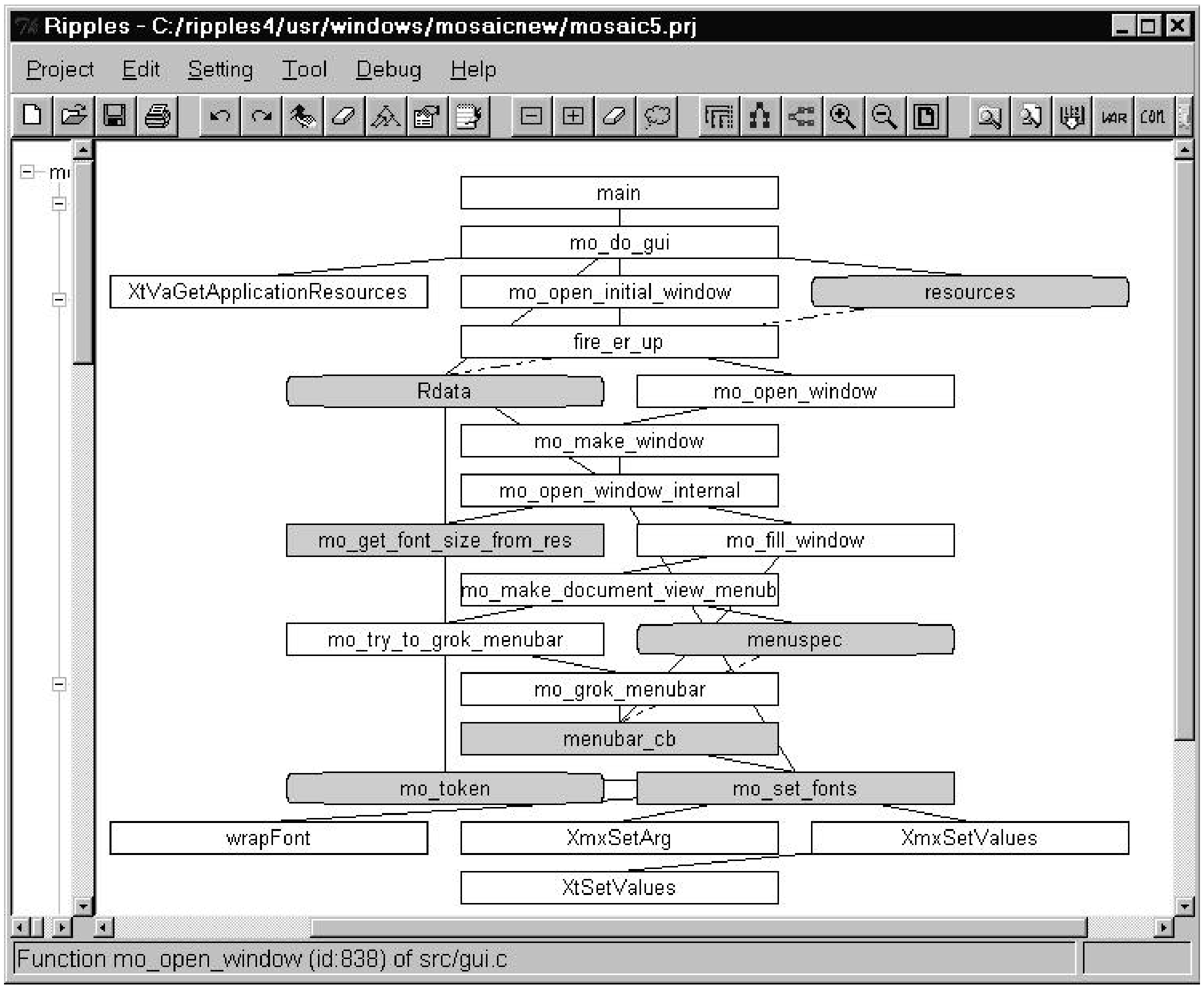
Operation *add component* adds a new component to both ASDG and screen graph. Operation *delete component* deletes an existing component and all its edges. Operation *change* can apply to any marked component on screen graph. After the maintainer changes the source code, the operation prompts the maintainer to input the edge(s) that are deleted or added. ASDG and screen graph is updated accordingly. After the change, all neighbors of the changed component are marked.

Operation *skip* is invoked when the marked component is visited and does not change. The programmer has an option either to stop the change, i.e. to simply erase the mark, or to propagate it to all neighbors, i.e. to mark all neighbors.

Conceptual dependencies are added to ASDG during the location process. Operation *add conceptual dependence* is for this purpose. The system displays the conceptual dependence using dashed lines. Conceptual dependencies have to be inserted by the programmer, as they are not extracted from the program by the analyzer.

4. Case Study of Change in MOSAIC

We conducted a case study of a change consisting of both location and propagation, using RIPPLES. The



Function mo_open_window (id:838) of src/gui.c

Figure 2: RIPPLES screen graph for location (Located Components Are Highlighted)

we studied was web browser NCSA Mosaic 2.5 for X Windows [17] and the task was to add one new font size to the standard set of fonts.

This case study is a self-observatory case study, done by the first author of the paper. Prior to the case study, he used Mosaic and that gave him rudimentary domain knowledge. He then read the user manual and on-line help to deepen the domain-level comprehension. This preparation was sufficient for the case study.

Domain knowledge

NCSA Mosaic web browser supports four font families: *Times*, *Helvetica*, *New Century*, and *Lucida Bright*. Three font sizes are provided for each font family: *regular*, *small* or *large*. This means that with all combination, there are 12 font types available. The default font is "Times Regular". Mosaic environment defines the following 17 properties and each of them specifies how Mosaic displays the corresponding content of html document:

```
Font, ItalicFont, BoldFont, FixedFont,
FixedBoldFont,        FixedItalicFont,
Header1Font, Header2Font, Header3Font,
Header4Font, Header5Font, Header6Font,
AddressFont, PlainFont, PlainBoldFont,
PlainItalicFont, SupSubFont.
```

For example, the Header2Font property defines how to display a level 2 header.

A new top-level window uses default fonts. The user can change the current fonts at any time. All twelve types are available for selection. New windows inherit the font types from their parent window.

For our case study, the maintenance request is to add a new font size *Tiny*. That means four new font types will be added: *Times Tiny*, *Helvetica Tiny*, *New Century Tiny*, and *Lucida Bright Tiny*.

As the font type setting is hard-coded in the system, we need to change the source code. In our case study, we first locate the concept "font type", then we change the code and finally we propagate the change.

4.1 Location

Our task is to locate where the font type is determined, and find how the properties are set and how the menu font operations manipulate these setting. It is a search process and the relevant part of ASDG is depicted in Figure 2.

We divided location into four subtasks for easier manageability. First subtask starts from the function main(). Each following subtask starts where the previous subtask finished. Also during the search, we add conceptual dependencies to the ASDG.

First subtask is to find the function that opens a new window. It is based on the fact that when we open a new browser window, it reads default font settings from the parent window. For this we adopt top-down strategy. The location starts from function *main()*. We found that function *mo_open_window()* is used to open a new window.

Second subtask is to find how the font properties are specified in the new window. We again adopted top-down strategy and started from *mo_open_window()*. Appendix A describes this subtask in more detail. We found that function *mo_set_fonts()* sets the properties for each font type.

Third subtask is to find what the default type is, and how and where it is set. We adopted backward data flow strategy. The search started from function *mo_set_fonts()*. We found that the default font is specified as string "Times Regular" in global variable *resources* of type *XtResources*. This subtask is described in Appendix B.

Fourth subtask is to find the connection between the font - related menu items and the font settings. In the Mosaic menu bar, the pull-down menu "Options" has an item "Font…", which sets the font type. In order to find this in the code, we adopted top_down strategy and started from mo_open_window(). We found that the *menubar_cb()* is used to specify the callback function for font menu operation. Function *mo_set_fonts()* is called to set the properties for each type.

Conceptual dependences, found in the location process, were added to the ASDG. One is between *resources* and *Rdata*. Both variables hold the same information – default font type, but each in different format. Both of them are actual arguments of function call *XtVaGetApplicationResources()* in function *mo_do_gui()*.

The other conceptual dependency is between *menubar_cb()* and *menuspec*. For each font related menu item in *menuspec*, there is one corresponding call-back function in *menubar_cb()*.

The concept "default font type" consists of the following components: *resources*, *Rdata*, *mo_get_font_size_from_res()*,*mo_token*, *mo_set_fonts()*, *menubar_cb()*, and *menuspec*. They are interconnected by both explicit and conceptual dependencies, see Figure 2.

4.2 Propagation

Our task is to add four font types: "*Times Tiny*", "*Helvetica tiny*", "*New Century tiny*", and "*Lucida Bright tiny*" to Mosaic.

The change begun from one of the located components, *mo_get_font_size_from_res()*, see Figure 3. This function maps the default font string to enumerated font type. Then the *mo_token* enumerated type was changed to add four types to it. The third change was to *mo_set_fonts()* and it added statements that set properties for the four new enumerated font types. Function *menubar_cb()* was changed to add the new call-back functions. Finally, the menu content was changed to add the four menu items to the menu. In total, five modifications were made to the source code. Some of the located components that contained the concept did not need any change. More details about the change are in Appendix C.

4.3 Observations from the case study

During the case study, the authors used both domain and programming knowledge to guide the search and the propagation. They also used Mosaic naming convention, where a component name derives from domain or functionality. The comments in the source code have also turned out to be useful.

To assess the help provided by RIPPLES, a comparison was drawn with an earlier case study of Mosaic, done by hand [7]. During that case study, the authors found that the search can become very complex. It is not uncommon to forget how and why we got to one specific component, and as a result the backtracking posed a particular problem.

RIPPLES provided us with an overall picture of the process that is very helpful. It also provided a lot of information in choosing a vertex to investigate. As a result, the second case study was finished in a much shorter period.

5. Measuring effectiveness of RIPPLES

For each location or propagation task, the following four numbers are retrieved: the number of edges

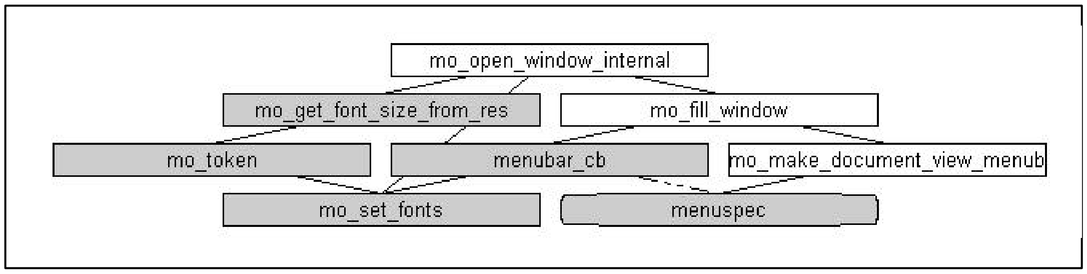


Figure 3. Screen graph for propagation (changed components are highlighted)

suggested by RIPPLES, the number of relevant edges, the number of relevant edges suggested by RIPPLES, and the number of edges suggested by RIPPLES and investigated. The first of these numbers is dependent purely on RIPPLES, while the remaining three numbers are produced by the programmer using it. As such, they are dependent on the user and his/her experience. Together they measure how well does RIPPLES support the human user.

Using these four numbers, we computed the recall, theoretical precision, and realistic precision, using formulas of [6] with an additional slight modification. In particular, we make a distinction between theoretical and realistic precision.

The theoretical precision is based on all possible continuations suggested by RIPPLES. Since RIPPLES is conducting static analysis of the program, it suggests a large number of possible continuations of the search, some of which can be ruled out immediately without reading the code of the components in question. If we do not consider the continuations that can be ruled out immediately, we get the realistic precision.

To rule out some continuations can be done very effectively. For example, when looking for a substantial concept like font type, we can immediately rule out all trivial functions. We can also rule out all functions that obviously deal with other concepts. Hence we are justified in making a distinction between theoretical and realistic precisions.

The formulas are the following:

$$\text{Recall} = \frac{\textit{The number of relevant edges suggested by RIPPLES}}{\textit{The number of relevant edges}}$$

$$\textit{Theoreticd precision} = \frac{\textit{The number of relevant edges suggested by RIPPLES}}{\textit{The number of edges suggested by RIPPLES}}$$

$$\text{Realistic precision} = \frac{\textit{The number of relevant edges suggested by RIPPLES}}{\textit{The number of edges suggested by RIPPLES and investigated}}$$

A recall rate is the percentage of relevant edges suggested by RIPPLES. Of the three measures, it is the most important one. Low recall rate means that the programmer must search the whole system for the missing edges. He/she has to use some other ways to find the edges, for example, read the source code or use some utility tool such as "grep". The following situation may cause a low recall rate: the ASDG analyzer fails to extract some edges; the two vertexes are connected by a conceptual dependence; the maintainer uses an inappropriate expansion strategy during location. To solve the low recall rate problem, we should work on all three above aspects: find a powerful and suitable ASDG analyzer, understand the system better, and adopt proper expansion strategy during location.

A low precision means that RIPPLES recommends irrelevant edges. The difference between theoretical and realistic precision depends on the judgement of the user. The continuations that are a part of the theoretical precision and not of realistic precision are only a small part of the programmer’s work because they can be ruled out quickly, and hence they do not impact adversely the programmer’s productivity. Therefore they are treated differently in this analysis. The following factors could results in low realistic precision: the maintainer does not have sufficient domain knowledge of the system; the maintainer does not have an overall search skill; the maintainer always chooses to expand to all neighbors.

Analysis of the location process reveals that both control flow and data flow expansions are used in the location. The data flow expansion was used when we dealt with a comprehension of specific data items, while top-down control flow expansion was used when we tried to understand a specific functionality or algorithm.

For subtask one, RIPPLES provides a total of 99 edges. It provides all 4 edges taken in the search and all other edges are ruled out immediately. So the recall rate is 4/4=100%, theoretical precision rate is 4/99=4.04%, and realistic precision rate is 4/4 = 100%.

In the second subtask, RIPPLES provides 48 edges. Only 13 of them are investigated and all other edges are excluded immediately. In this subtask, the recall rate is

13/13=100%. Theoretical precision is 13/48 = 27.08% and realistic precision is 13/13 = 100%.

In the third subtask, 3 steps are taken, all of them are suggested by RIPPLES. There are 94 possible choices. So the recall is 100%, the theoretical precision is 3/94=3.19%, the realistic precision is 3/3=100%.

There are a total of 7 relevant steps in the fourth subtask. They are picked up for investigation from 110 possible choices, of which we investigated nine. Two edges are found irrelevant so we backtrack two steps. The recall rate is 100%, theoretical precision is 7/110 = 6.36%, and the realistic precision is 7/9 = 77.78%.

For the entire location, we moved 22 times from one function to another through function call, moved through data flow 5 times, accessed data type information twice. So, a total of 29 unique edges are investigated of all 5779 edges in the system, of which 27 are relevant to our task. RIPPLES suggests a total of 351 edges for investigation. Of the 1662 functions and global variables in Mosaic, we visited only 23. Six components are located in this process as a part of the concept implementation. For the whole location process, recall rate is 27/27 = 100%, theoretical precision is 27/351 = 7.69%, and realistic precision is 27/29 = 93.10%.

Analysis of the propagation process: Five components are changed and four edges are involved. Among the four edges, one is conceptual edge. There are totally 71 edges suggested by RIPPLES and only the four are investigated. So the recall rate with conceptual dependencies is 4/4=100%, without them it is 3/4=75%. The theoretical precision is 4/71=5.63%, and realistic precision is 100%.

6. Other work

Our work builds on several efforts. Rajlich and Bennett [19] proposed a staged model for the software life cycle. During evolution stage, software engineers fully comprehend the system, and extend the functionality of the system to satisfy new requirements. In servicing stage, only small repairs will be done to the entire system. RIPPLES is aimed for servicing stage where full comprehension is not expected.

Biggerstaff et al. [2] defined and investigated the concept location, called "concept assignment" in their paper. To perform concept assignment, a prior knowledge of the specific domain, a plausible reasoning, etc are needed. As concepts and program are not in the same level of abstraction, human input is necessary. Their conclusion is that totally automated tool for concept assignment is probably impossible, but some degree of automation is helpful.

In our previous research [7], we studied location using the dependence graphs. The case study reported in [7] identified the requirements for RIPPLES.

Wilde et al. [24, 25] developed a program feature location technology called Software Reconnaissance. The technology is based on the analysis of test cases. The instrumented program is tested with two sets of test cases: one set of test cases with the feature, and the other set without. The feature location is done by analysis of the two sets of event traces. A tool *Recon2* supports this technique.

Jerding and Rugaber [14] use both static and dynamic analysis in program understanding. Static analysis is used to extract the system architecture, and dynamic analysis is used to analyze the behavior of specific components and their interactions. After running a program, they analyze the event trace and abstract the interaction pattern into various levels of abstraction. A visualization tool, ISVis, is developed for this purpose. They use ISVis in a Mosaic case study.

Horwitz and Reps [11,12] proposed System Dependence Graph to represent programs with multiple procedures. They use SDG in program slicing, program differencing and program integration. Harrold et al. [10] developed a method to construct SDG from Abstract Syntax Trees. Liang and Harrold [16] also extended SDG to object oriented programs and used it in program slicing.

Recall and *precision* are the most commonly used formulas to compute the information retrieval effectiveness [6]. The original concept of *recall* is defined as "the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in the database". Precision is defined as "the ratio of the number of relevant documents retrieved over the total number of documents retrieved". Both recall and precision are between 0 and 1. Researchers have recently used recall and precision in evaluating effectiveness of software tools. Koschke, etc. [15] used them for experimental evaluation of clustering techniques for component recovery. Antoniol, etc. [1] also used recall and precision to evaluate the performance of their tool, that maintains traceability links between source code and free text documents.

Software browsers [13, 20] extract and graphically represent program dependencies. However they do not support the process of the change and do not support marks that the process of change requires.

Wilde et al [23] investigated a FORTRAN legacy system from the early 70's and found that it has many properties different from Mosaic of early 90's. The data flows still play a prominent role in the location while control flows become less useful.

Graph layout algorithms relevant to RIPPLES are Sygiyama [21], Graphviz [9], and Star [4].

7. Conclusions and future work

The architecture and functionality of tool RIPPLES have been described in this paper. The case study of NCSA Mosaic shows that it can work on practical systems. Mosaic has been widely used by researchers as a "subject" for the research of maintenance tools and techniques. We believe it is a representative of early 90's C language legacies and there is a lot of legacy code with similar characteristics. With the usage of screen graph, we believe that RIPPLES is able to support the servicing of many large systems.

Precision and recall are important indicators of the tool quality. Of the two, recall is more important; for the programmer it is important to get the information about all possible continuations. Compared to that, low precision that provides additional unused continuations is not such a big problem, because those continuations can be easily ruled out.

The important research goal is to improve both precision and recall. Both will be improved by extracting more relevant information by program analysis. An intriguing question is posed by conceptual dependencies. They are the dependencies that are not retrieved by current static analysis techniques, yet they play an indispensable role in the change propagation.

The conceptual dependencies we encountered can be classified into two categories. In the first category, dependence involves data flow within library functions. Due to unavailability of source code of these functions, it is impossible to retrieve the dependence. For example in our case study, the conceptual edge between *resources* and *Rdata* could not be recovered by static analysis, because the code of library function *XtVaGetApplicationResources()* is not available.

In the other category, the two components are only connected in the application domain. An example in our case study is the dependency between *menubar_cb()* and *menuspec*. At this moment, it is not clear what – if any – analysis would automatically discover such a dependency. Further study of conceptual dependencies is our future goal.

8. References

[1] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, E. Merlo, "Tracing Object-Oriented Code into Functional Requirements", Proc. of International Workshop for Program Comprehension (IWPC)'2000, pp. 79-86.

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

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

[4] R. W. Bowdidge, W. G. Griswold, "Automated Support for Encapsulating Abstract Data Types", SIGSOFT Software Engineering Notes, vol. 19, no.5, pp. 97-110, Dec. 1994.

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

[6] J. A. Capon, "Elementary Statistics for Social Sciences", 1988, Wadworth publishing.

[7] K. Chen, V. Rajlich, "Case Study of Feature Location Using Dependence Graph", Proc. of the IWPC'2000, pp. 241-249.

[8] Datrix C/C++/Java parser web site (by Bell Canada): http://www.iro.umontreal.ca/labs/gelo/datrix/homepage.htm.

[9] Graphviz web site (by AT&T research): http://www.research.att.com/sw/tools/graphviz/.

[10] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient Construction of Program Dependence Graphs", Proc. of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'93), pp. 160-170.

[11] 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.

[12] S. Horwitz, T. Reps, "The Use of Program Dependence Graphs in Software Engineering", Proc. of the 14th International Conference on Software Engineering (ICSE), May 1992.

[13] Imagix web site: http://www.imagix.com.

[14] 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 Press, pp. 56-65.

[15] R. Koschke, T. Eisenbarth, "A Framework for Experimental Evaluation of Clustering Techniques", Proc. of International Workshop on Program Comprehension, 2000, IEEE Computer Society Press, pp. 201-210.

[16] D. Liang, M. J. Harrold, "Slicing Objects Using System Dependence Graphs", Proc. of International Conference of Software Maintenance (ICSM)'1998, November 1998, pp. 358-367.

[17] Mosaic web site (source codes and documents):

http://www.ncsa.uiuc.edu/SDG/Software/Mosaic

[18] V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting", Proc. of ICSM'1997, pp. 84-91.

[19] V. Rajlich, K. Bennett, "A Staged Model for the Software Life Cycle", IEEE Computer, July 2000, pp. 66-71.

[20] V. Rajlich, N. Damaskinos, P. Linos, W. Khorshid, "VIFOR: A Tool for Software Maintenance," Software Practice and Experience, Vol 20 (1), January 1990, 67-77.

[21] K. Sugiyama, S. Tagawa and M. Toda. "Methods for Visual Understanding of Hierarchical System Structures", IEEE Transactions on Systems, Man and Cybernetics, Volume 11, February 1981, pp. 109-125.

[22] Tcl/Tk resource web site: http://www.scriptics.com.

[23] N. Wilde, M. Buckellew, H. Page, V. Rajlich, A Case Study of Feature Location in Unstructured Legacy Fortran Code, Proc. Fifth European Conf. On Software Maintenance and Reengineering, IEEE Computer Society Press, 2001, 68-76

[24] N. Wilde and T. Gust, "Locating user functionality in old code", Proc. 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, pp. 49-62, 1995.

[26] Z. Yu, V. Rajlich, "Hidden Dependencies In Program Comprehension and Change Propagation", Proc. of Proc. of International Workshop on Program Comprehension, 2001, IEEE Computer Society Press, pp. 201-210.

Appendix A: Locate components that determine font properties

Function **mo_open_window()** calls *mo_make_window()* and *mo_load_window_text()*. The functionality of *mo_make_window()* is to make a new window from scratch. It creates an X window shell and then calls *mo_open_window_internal()*.

Function **mo_open_window_internal()** calls memory allocation function *malloc()* to create a structure of type *mo_window*, the internal structure for each window, and calls function *mo_fill_window()* to fill the window just created. Between these two function calls, two functions related to font processing are called: *mo_get_font_size_from_res()* and *mo_set_fonts()*. Function *mo_get_font_size_from_res()* is called when current window is the top window, i.e. without parent. The actual argument is *Rdata.default_font_choice*. *Rdata* is a global variable of type *AppData*, a C structure, while *default_font_choice* is a string field of this C structure. Function *mo_set_fonts()* is called when the font is not the *mo_regular_fonts*.

Function **mo_get_font_size_from_res()** maps the string font type to enumerated *mo_token* type. For example, the string "Helvetica Regular" is mapped to enumerated constant *mo_regular_helvetica*. There is one enumerated constant defined for each font type.

Function **mo_set_fonts()** is used to define font properties. It defines font properties for the given font type. Each property specification is a call to function *XmxSetArg()*, one of the arguments is the property name. X system function *XtSetValues()* is then called to transform the properties to X widget resources for later GUI interface usage.

In summary, functions *mo_get_font_size_from_res()* and *mo_set_fonts()* are used for font type definition and are located in this search.

Appendix B: Search what default font type is and how it is set

In last subtask, *mo_set_fonts()* is located. It is called by *mo_open_window_internal()*, which also calls function *mo_get_font_size_from_res() with actual argument* **Rdata.default_font_choice**. In this subtask, we want to find how this field of global variable **Rdata** gets its value.

The information flows to *Rdata* from library 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*. XtVaGetApplicationResource() is a library function, its source code is not available. We are not able to extract this data flow information. This edge is added as conceptual dependence by hand.

Inspection of global variable **resources** finds that the information is stored as a part of a structure, an array. Each element in the array describes one XtResource: its name, it default value, etc., and "defaultFontChoice" is one of them. Its initial value is "Times Regular".

A conceptual dependence is found between *resources* and *Rdata*. Both of them are actual arguments of *XtVaGetApplicationResources()*. We add a conceptual dependence to the ASDG to represent this connection.

Appendix C: Change propagation

We started the change propagation in function **mo_get_font_size_from_res()**, which is the one of the located functions, see Figure 3. This function maps the font string to the enumerated font type. We need to add statement to deal with four new font types. This is accomplished by addition of the following if-statements to the appropriate places:

```
if (strstr(lowerfontstr, "tiny")!=NULL) return mo_tiny_times;

if (strstr(lowerfontstr, "tiny")!=NULL) return
mo_tiny_helvetica;

if (strstr(lowerfontstr, "tiny")!=NULL) return
mo_tiny_newcentury;

if (strstr(lowerfontstr, "tiny")!=NULL) return
mo_tiny_lucidabright;
```

This requires a change in enumerated type **mo_token**. The following four enumerate constants will be added to it:

```
mo_tiny_times, mo_tiny_helvetica,
mo_tiny_newcentury, mo_tiny_lucidabright.
```

The third changed component is **mo_set_fonts()**: add statements to set properties for the four new enumerated font types. Function *mo_set_fonts()* is connected with *mo_token* by a dependence. The following statements are added to *mo_set_fonts()* before original line 392 in *src/gui-menubar.c*:

```
case mo_tiny_times:
    XmxSetArg (XtNfont, wrapFont("-adobe-times-medium-
                    r-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNitalicFont, wrapFont("-adobe-times-
                    medium-i-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNboldFont, wrapFont("-adobe-times-
                    bold-r-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNfixedFont, wrapFont("-adobe-courier-
                    medium-r-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNfixedboldFont, wrapFont("-adobe-
                    courier-bold-r-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNfixeditalicFont, wrapFont("-adobe-
                    courier-medium-o-normal-*-12-*-*-*-*-*-*-
                    *"));
    XmxSetArg (WbNheader1Font, wrapFont("-adobe-
                    times-bold-r-normal-*-16-*-*-*-*-*-*-*"));
    XmxSetArg (WbNheader2Font, wrapFont("-adobe-
                    times-bold-r-normal-*-15-*-*-*-*-*-*-*"));
    XmxSetArg (WbNheader3Font, wrapFont("-adobe-
                    times-bold-r-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNheader4Font, wrapFont("-adobe-
                    times-bold-r-normal-*-10-*-*-*-*-*-*-*"));
    XmxSetArg (WbNheader5Font, wrapFont("-adobe-
                    times-bold-r-normal-*-8-*-*-*-*-*-*-*"));
    XmxSetArg (WbNheader6Font, wrapFont("-adobe-
                    times-bold-r-normal-*-6-*-*-*-*-*-*-*"));
    XmxSetArg (WbNaddressFont, wrapFont("-adobe-times-
                    medium-i-normal-*-12-*-*-*-*-*-*-*"));
    XmxSetArg (WbNplainFont, wrapFont("-adobe-courier-
                    medium-r-normal-*-10-*-*-*-*-*-*-*"));
    XmxSetArg (WbNplainboldFont, wrapFont("-adobe-
                    courier-bold-r-normal-*-10-*-*-*-*-*-*-*"));
    XmxSetArg (WbNplainitalicFont, wrapFont("-adobe-
                    courier-medium-o-normal-*-10-*-*-*-*-*-*-
                    *"));
    XmxSetArg (WbNsupSubFont, wrapFont("-adobe-times-
                    medium-r-normal-*-6-*-*-*-*-*-*-*"));

    XmxSetValues (win->scrolled_win);
    win->font_family = 0;
    break;
```

Each above function call to XmxSetArg() defines one of the 17 properties of the font type. For example, the “Header2Font" property is defined as “-adobe-times-bold-r-normal-*-15-*-*-*-*-*-*-*”. Similar changes are made for the remaining font types.

Function **menubar_cb()** defines call back functions for all menu items. After calling *mo_set_fonts()* to define properties for all four new enumerate font types, *menubar_cb()* needs to add call back functions for all new menu items. The actual modification is to add the following statements in file *src/gui-menubar.c* before original line 709:

```
case mo_tiny_times:
case mo_tiny_helvetica:
case mo_tiny_newcentury:
case mo_tiny_lucidabright:
```

Finally, the four font types must be added to menu content specification. Function *menubar_cb()* and *menuspec* are connected with conceptual dependence. Global variable **menuspec** is used to specify menu content. It is an array of structures and each of them corresponds to one pull-down menu in the menu bar. “Options” is one of them. Global variable *opts_menuspec* specifies the content in the “Options” pull-down menu. It also is an array of structures and each of them corresponds to one menu item in the menu. “Font…” is one of them. Global variable *fnts_menuspec* specifies the content in the “Font…” menu item.

The actual change is made to *fnts_menuspec*: the following statements are added before original line 1001, 1005, 1009 and 1013 in file *src/gui-menubar.c*:

```
{"<Times Tiny",           'a', menubar_cb, mo_tiny_times },
{"<Helvetica Tiny",       'b', menubar_cb, mo_tiny_helvetica},
{"<New Century Tiny",     'd', menubar_cb,
mo_tiny_newcentury },
{"<Lucida Bright Tiny",   'f', menubar_cb,
mo_tiny_lucidabright },
```

In conclusion, we made five changes to the system. One conceptual dependency and three explicit dependecies are used in the change propagation. The rest of the dependencies of Figure 3 were not used in change propagation.