

Analyzing xfig Using the Rigi Tool Suite

Johannes Martin* Kenny Wong† Bruce Winter* Hausi Müller*

University of Victoria*
Department of Computer Science
Victoria, BC, Canada

University of Alberta†
Department of Computer Science
Edmonton, AB, Canada

Abstract

In connection with a workshop titled 'A Collective Demonstration of Program Comprehension Tools' held during the CASCON conference in 1999, an experiment was conducted on how well expert users of program comprehension tools were able to perform specific program understanding and maintenance tasks on the xfig drawing program using these tools. This paper reports on the experiences of the users of the Rigi reverse engineering tool suite.

1. Introduction

1.1. The Rigi Reverse Engineering Tool Suite

Rigi [3, 2, 4, 8, 7] is an interactive, visual tool designed to help developers better understand and redocument their software. Rigi includes parsers to read the source code of the subject software and produce a graph of extracted artifacts such as procedures, variables, calls, and data accesses. To manage the complexity of the graph, an editor allows the software engineer to automatically or manually collapse related artifacts into subsystems. These subsystems typically represent concepts such as abstract data types or personnel assignments. The created hierarchy can be navigated, analyzed, and presented using various automatic or user-guided graphical layouts.

The discovered structural information is useful for making informed development and management decisions. The information serves as documentation that is up-to-date and accurate because it is derived from the actual source code. Thus, Rigi helps to understand legacy software systems where the existing documentation may be missing or lacking. Rigi aids reengineering tasks that need to discover design information in existing software.

Rigi also includes a tool to view the subject software's source code using a web browser. The tool produces a hy-

perertext version of the source code that includes hypertext links for related source artifacts (for example, each function call has a link to the definition of the function that was called).

1.2. Previous Usability Experiences

In past user studies [6, 5], novice users were trained in 20-40 minute sessions to use the Rigi graph editor for viewing and exploring software systems that had previously been imported into the tool and prepared by experienced users. These experiments have shown that novice users can gain a basic understanding of the tool through relatively short training. In order for a new user to import small software systems into Rigi and to use the tool for program understanding tasks another one hour training is necessary.

To understand large software systems, it is important to be able to break down the complexity of a system by building subsystem decompositions. This does not only require the knowledge of the Rigi system, but a thorough understanding of software architecture and decomposition techniques. Rigi supports this tasks through interactive tools and a scripting language, that is especially helpful when dealing with industrial size systems. The interactive tools for building subsystem decompositions are intuitive in their use and easy to learn. The scripting language however (an extension of *tcl/tk*), is not very well documented, and developers not familiar with *tcl/tk* will need some time to be able to use it.

Parsing the source code of a large software system often poses a problem, in particular if the system uses non-standard features of a programming language that are not supported by the parser. Some time is also required to adjust the build environment of a software system to work with the Rigi parsers instead of a regular compiler. In the recent past, efforts have been made to facilitate the parsing process and to provide a catalog of standard techniques for parsing and decomposing large software systems with Rigi.

1.3. Participants in the Experiment

Three members of the Rigi team at the University of Victoria participated in the experiment. All of them had used the *xfig* drawing program occasionally. Their knowledge of the Rigi tools varied:

Kenny Wong is a long time member of the Rigi team. He gained a lot of experience in the practical application of the tool in past examinations of software systems. In particular, he is familiar with the scripting capabilities of the Rigi graph editor.

Johannes Martin is the maintainer of the Rigi C parser and quite familiar with parsing C programs for use with Rigi.

Bruce Winter is a novice user of the Rigi tools.

2. Results

2.1. Documentation and Evaluation

Rigi was used for visualizing the structure of the *xfig* system. We identified clusters using file containment dependencies and naming conventions. Figure 1 shows the top-level subsystem decomposition. We identified four major subsystems:

User Interface is concerned with the views and controls presented to the user.

Drawing Primitives contains routines for drawing the various objects that are supported by *xfig*.

Editing Operations encapsulates the control flow for editing operations offered to the user.

File I/O handles the storage of figure files on disk and conversion to and from foreign file formats.

During parsing, we identified a number of files that were not used with our configuration of *xfig* at all (`f_readfigure.c`, `f_readxpm.c`, `w_fontbits2.c`, `w_fontbits3.c`, `i18n.c`). They may either contain unused functionality (dead code) or be significant only in different *xfig* configuration (for example on other hardware or software platforms).

The Rigi tools also produced a hyperlinked version of the *xfig* source code that can be viewed using Netscape. We used it during the subsequent maintenance tasks.

As Rigi does not have builtin support for source code evaluation, the freely available metrics tool *Routine Analyzer* in combination with Unix file utilities such as *find* and *egrep* were used for the evaluation task. The Routine Analyzer tool indicated that although there were large files with

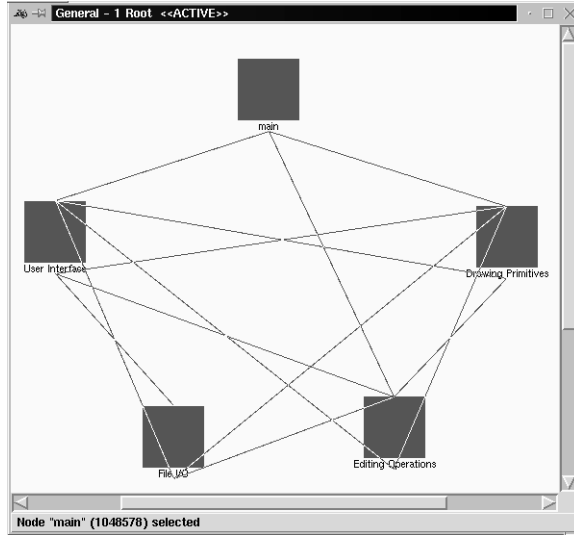


Figure 1. Subsystem Decomposition of *xfig*

many functions, the sizes of those functions were not overly long. Five `gotos` were discovered, but since they did not significantly increase the complexity of the program, it was decided not to replace them.

2.2. Maintenance Tasks

2.2.1. Modifying the Existing Command Panel

Using the Unix *egrep* utility and the hyperlinked version of the *xfig* source, we identified the file (`w_cmdpanel.c`) that needed to be changed in order to implement the new menu structure. The `cmd_switches` data structure represented the current command panel. We modified this data structure and added three additional data structures (`file_menu_items`, `edit_menu_items`, and `view_menu_items`) for the new submenus as well as the appropriate call backs. Finally, the modified code was compiled and tested.

2.2.2. Adding a New Method for Specifying Arcs

Using similar methods as for the previous task, we identified the file `d_arc.c` in the Drawing Primitives subsystem as responsible for the arc drawing primitive. The `create_arcobject()` function needed to be modified in order for the new method of specifying arcs to be implemented. After making the necessary changes, the code was compiled and tested.

3. Experience Report

Due to problems with the Rigi C parser, it took an initial three hours to parse the source code of xfig. They were due to a minor error in our configuration of the parser that we did not identify right away (even though it is documented). Parsing was the only real problem encountered during our study; otherwise the tool ran stable and could be adjusted to the particular environment using the built-in scripting capabilities.

The real time needed to parse the source code into the Rigi file format (RSF[1]) was about 20 minutes – about as long as it takes to compile the xfig source code. The resulting RSF file is about 4MB in size. The Rigi graph editor reads it in almost instantaneously. Within the Rigi graph editor, we used standard scripts to eliminate system and standard library functions from the graph (these are usually not of interest during a maintenance activity, but useful when porting a system to a new platform). Custom scripts were used to decompose the system using file containment relations and naming conventions.

Once the source was parsed, we were able to use Rigi source code manipulation tools to insert hyperlinks into the xfig source code, so it could be viewed and navigated in Netscape comfortably.

For the maintenance tasks, we used the hypertext enhanced view of the source code and standard Unix tools (*egrep*) for locating the sources that needed to be changed.

4. Lessons Learned

Even though efforts have been made over the last years to facilitate the task of parsing large software systems, the parsing process still posed the biggest problem in this experiment. Better feedback of the parser to the user during the parsing process and tighter integration into the rest of the Rigi tool suite might help to avoid similar problems in the future.

The scripts we used for the subsystem decomposition have been used in similar forms before, but in a typical case, they are rewritten from scratch, because there is no catalog of well documented scripts.

The experiences gained during this experiment emphasize the need for better documentation and usage examples for our tools. Some efforts to collect and record data from users and developers are already underway. Over the last year, we have created mailing lists for users of Rigi. Though only few Rigi users have chosen to subscribe to these lists, they have already proven useful for Rigi users in that they can exchange experiences and tool extensions. By archiving these lists we gather important information that will help us in better documenting our tools.

As Rigi is primarily a code visualization and reverse engineering tool, it did not lend itself easily for the maintenance tasks suggested in the experiment. Although Rigi helped in locating the code to be changed, other tools had to be used for the actual implementation of the changes. Thanks to Rigi's scripting capabilities, these tools can be closely integrated with Rigi to ease recurring maintenance tasks.

References

- [1] J. Martin. RSF file format. Technical report, University of Victoria, Aug. 1999. <http://www.rigi.csc.uvic.ca/list-archives/rigi-developer-archive/2000-02-10-15:26:16-19165>.
- [2] H. Müller, K. Wong, and S. Tilley. Understanding software systems using reverse engineering technology. In *Proceedings the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences*, 1994. <http://www.rigi.csc.uvic.ca/Pages/publications/ussuret.pdf>.
- [3] H. A. Müller and K. Klashinsky. Rigi — A system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, pages 80–86, 1988.
- [4] H. A. Müller, M. A. Orgun, S. R. Tilley, and J. S. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, Dec. 1993. <ftp://ftp.rigi.csc.uvic.ca/pub/papers/jsm.ps.Z>.
- [5] M.-A. Storey, K. Wong, and H. A. Müller. How Do Program Understanding Tools Affect How Programmers Understand Programs? In *Proceedings of WCRE'97*, pages 12–21, Amsterdam, Holland, Oct. 1997.
- [6] M.-A. D. Storey, K. Wong, P. Fong, D. Hooper, K. Hopkins, and H. A. Müller. On Designing an Experiment to Evaluate a Reverse Engineering Tool. In *Proceedings of the 3rd Working Conference on Reverse Engineering*, Monterey, CA, Nov. 1996.
- [7] S. R. Tilley. *Domain-retargetable reverse engineering*. PhD thesis, Department of Computer Science, University of Victoria, 1995. <http://www.rigi.csc.uvic.ca/Pages/publications/dtre.pdf>.
- [8] K. Wong, S. R. Tilley, H. A. Müller, and M.-A. D. Storey. Programmable Reverse Engineering. *International Journal of Software Engineering and Knowledge Engineering*, 4(4):501–520, Dec. 1994. <http://www.rigi.csc.uvic.ca/Pages/publications/progreveeng.pdf>.