Visualizing the Evolution of Software Systems

KAC: Extensions to the Rigi System

Mark G. Adams (mgadams@se.math.uwaterloo.ca)
James P. D. Keast (james@keiths.uwaterloo.ca)
David H. Chen (dhchen@neumann.uwaterloo.ca)

http://keiths.uwaterloo.ca/james/courses/746h/evolution/

2 December 1998
Department of Computer Science
University of Waterloo

at
Waterloo, Ontario, N2L 3G1

ABSTRACT

The problem of maintaining an evolving software system is one software engineers continually struggle with. A primary difficulty which arises as software systems grow is merely being able to understand them. A great deal of work has been done in the area of software architectures[6, 3] which has helped in this understanding. Many techniques exist for redocumenting an architecture and for visualization of this architecture[4, 11, 8, 10]. However we were unable to find a tool for visualizing the *evolution* of a software system. This paper presents an approach for examining the evolution of software systems and extensions to the *Rigiedit*[11] tool to support the visualization of software system evolution.

Keywords: **Software Architecture, Software System Evolution, Evolution Visualization, Rigiedit**

A final project prepared for **Computer Science 746H.** ***Topics in Software Engineering.***
Instructor: Mike Godfrey

Contents

List of Figures

1 Introduction

The evolution of a software system is the process by which a system changes in order to meet new demands of the marketplace and end users. Nearly all software systems which are successful will eventually evolve. The problem of maintaining these systems during their evolution is one software engineers continually struggle with. As systems evolve it is not uncommon for their documentation to become extinct. Software engineers are left with systems which have evolved to the degree that nobody understands them. When it comes time to make major changes to these systems or even to bug fixes, we are faced with the nearly insurmountable problem of merely understanding them.

Work in the area of reverse engineering and the visualization of software architecture attempts to assist with this understanding problem. A common practice is to extract an architecture from the source code and use this to fuel the redocumentation effort. A great deal of work has been done in the area of classifying these architectures in order to support the understanding effort[3, 6].

In contrast, very little work has been done in the area of documenting the evolution process itself. We believe that understanding how a system has evolved will help to understand the design decisions made and ultimately help in the understanding of the current state of the system. In addition examining the evolution of the software system may identify erroneous design decisions which may not have been evident at the time (*hindsight being 20/20*).

To this end we will explore the idea of visualizing the evolution of a software system by comparing later versions of a system with an earlier version. The approach described in this paper will certainly aid in the understanding of a system's evolution. Comprehension of the evolution process will provide crucial information for understanding the current status of the software system. Our approach is supported by extensions we have made to *Rigiedit*[11] which we use for visualization. The extensions which make up the `KAC` tool are detailed in this paper and examples of their use are provided.

1.2 Overview

The remainder of this report is divided as follows. In Section 2 we briefly present the related work in the areas of reverse engineering, software reflexion and software system evolution. Section 3 contains an overview of our approach which leads into Section 4 where the tool extensions are presented in detail. Three case studies are presented in Section 5 which explain how our approach and tool extensions work in practice. In section 6 we examine some of the limitations of our approach and suggest some areas for future work. Finally Section 7 provides a summary of our experiences in using our approach and our tool extensions.

1.3 What do all these boxes, lines and arrows mean?

In this paper visualization of architectures is done using the *Rigiedit*[11] tool. Boxes or nodes represent either subsystems or files. The distinction is made by the names given to the boxes. The lines between boxes represent call, data or include relationships (others are possible, but they are the relationships which the authors felt were most useful in understanding the system). The direction of these relationships is identified by their locations on the boxes. Lines leaving the bottom of a box are relationships *from* that box while lines entering from the top of the box are relationships *to* that box.

2 Related Work

2.1 Reverse Engineering

One of the reverse engineering approaches which forms a basis for our work makes use of clustering information. Clustering information is generated from a combination of human input and numerical computation. Several examples of this approach include *Rigi*[11] and Schwanke's statistically-based architectural recovery technique[7]. Our work differs from this approach in that we allow the software engineer to specify the high level subsystem decomposition and we provide tool support to assist in this process. In addition our work is concerned with more than just reverse engineering the architecture of a software system; we are interested in visualizing the evolution of the software system.

2.2 Source Model Extraction

Much of the work done in the area of reverse engineering relies heavily on the ability to extract models from the source code. Useful information that can be extracted includes: call graphs, file dependencies and data referencing information[5]. In addition, many tools such as PBS[10], Rigi[11] and CIA[8] exist for visualizing these extracted architectures. Our approach makes use of an underlying source model extraction but does not restrict this model at all. The model can be extracted using any parser but the results must be stored in Rigi Standard Form (RSF).

2.3 Reflexion Models

Murphy and Notkin describe an approach called reflexion modeling[4] that bridges the gap between source and high-level models. The reflexion model is designed to address the problem of accurately documenting and visually representing the source level models without actually looking through thousands or even millions of lines of code. A reflexion model takes user-created high-level model and generates a new model that reflects the accuracy of the high-level model by stating the missing relationships, the unexpected relationships and the existing relationships. Our work is based in part on Murphy's paper in that we also compare two models and then describe the difference. Murphy's approach is to reflect the high-level model and the source-level model. In contrast, we describe the changes made between the source-level model of an earlier version and the source-level model of a later version.

2.4 Evolving Software Systems

Successful software systems will inevitably evolve in order to meet the new goals of users. It is therefore necessary to be able to manage this evolution. There are many approaches which try and deal with this change. Often systems evolve by adding new code, modifying existing code, or deleting obsolete code. It is therefore desirable to be able to construct and maintain a high level model of the system that reflects the changes in the system as it evolves. Ernst et al. approached this problem by “Dynamically Discovering Likely Program Invariants”[2] in a system to support program evolution. They focused on the invariant shifts (data structure changes, sub-routine interface change, etc.) in a system as it evolves. Our work emphasizes instead the relationship shifts between entities in a system as the system evolve from one version to another. Like Ernst we provide tools to automatically extract relevant system evolution information.

3 Our Approach

Understanding an evolving software system is more often than not a very difficult task. We believe that being able to compare multiple versions of the architecture of a software system is a useful approach for gaining an understanding of its evolution. Indeed, this is the process followed by anthropologists when studying the evolution of humans.

Comparing multiple versions of a software system is best performed by representing the versions graphically. This graphical representation can then be examined and navigated to determine the differences between the versions. Information about these differences can then be used to identify dependencies which have been created or removed by the evolution process. By highlighting any altered dependencies, we are able to easily pinpoint where the software system's architecture has been altered (for example, by creating a dependency between two modules which had previously been unrelated).

Many tools exist for visualizing the architecture of a software system at a given point in time including *CIA*[8], *PBS*[10] and *Rigi*[11]. However, we were unable to find anything that was particularly useful for visualizing the architecture of a software system as it evolves over time. Thus we decided to extend *Rigiedit* to support this evolutionary visualization.

3.1 Comparing Versions

A basic requirement for examining how a software system has evolved is to have at least two different versions of the source code available to be examined. Once the relationships between facts in each of the versions have been identified (as discussed in section 3.3), the relationships in the earlier version can be compared to the relationships in the later version. Each relationship can then be classified into one of three categories:

1. **Old.** These relationships are identical in both versions of the software system.
2. **New.** These relationships exist in the newer version of the software system but do not exist in the older version.
3. **Missing.** These relationships exist in the older version of the software system but do not exist in the newer version.

Maximum benefit can be derived by first creating a modular (subsystem) decomposition of the software system. Then the new (or missing) relationships can be highlighted to indicate where there are new (or missing) dependencies between the various subsystems.

These new (or missing) dependencies could be indications that the architectural style for the software system has been broken or otherwise compromised in those locations. Each new (or missing) dependency can then be examined to determine exactly what change resulted in its creation (or removal).

3.2 Visualizing Evolution

While merely listing the changes between the architectures of various versions of a software system is of some help, it would be even more beneficial to be able to visualize these differences. Ideally we would like to be able to both visualize the architecture and also be able to readily distinguish between the old, the new and the missing relationships. The latest version of *Rigiedit* (5.4.4) provides the means to visualize the architecture as well as providing the basis for distinguishing between the three types of relationships based on their colour. By extending *Rigiedit* to recognize the three types of relationships instead of just one (for example, **oldCall**, **newCall** and **missingCall** instead of just call), we are able to easily distinguish between these types of relationships.

One assumption which is being made in this approach is that the underlying subsystem decomposition of the software system doesn't change significantly between the versions being examined. If the architecture changes significantly, then the concept of a new or an old relationship is rendered meaningless.

3.3 Fact Extraction Process

The extensions to *Rigiedit* described in this paper are entirely independent of the origination of the *Rigi Standard Form* (RSF) fact files. Any parser or fact extractor may be used so long as the facts are encoded in RSF. The current extensions are concerned with the following RSF relationships:

1. **Call.** This is a relationship which indicates that a particular function calls another function.
2. **Data.** This is a relationship which indicates that a particular function uses a data item such as a global variable.
3. **Include.** This is a relationship which indicates that a given file includes another file (generally a header file).

In addition, the extensions assume that the **file** attribute of all of the nodes is set. This attribute indicates in which file the node is contained (for example, the source file in which a function is defined). This attribute is necessary in order to automatically raise facts to the file level.

As stated above, our extensions are not dependent upon the origins of the RSF files. For the case studies presented here, the facts were extracted using the *CIA*[8] toolset. In each case, the database was built automatically by a separate program and a set of stock queries was run. The results of these queries were then converted to RSF. More information on this process can be found in our earlier work[1].

4 Visualization Extensions (KAC)

Rigiedit is the visualization tool which comes with the *Rigi*[11] system. This tool was chosen because it is easily extendable using *TCL/Tk* scripts. Figure 1 shows the main *Rigiedit* window. Our extensions are loaded by typing `source KAC.tcl` into the RCL* window as shown.

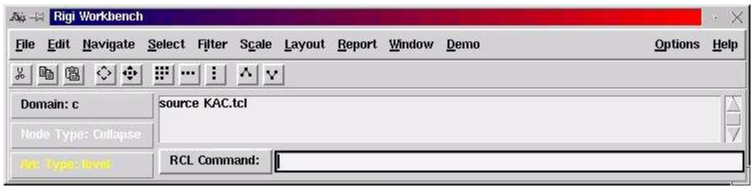

Figure 1: The main *Rigiedit* window.

Three major extensions were made to the tool:

1. The capability to raise all facts to the file level was added.
2. The ability to create a subsystem allocation was added.
3. The ability to compare two RSF fact files to create a third (comparison) file was added.

Each of these extensions is available through the top level dialog box as shown in Figure 2.

The **File Level Raiser Tool** reads through the loaded RSF and builds a list of all of the files represented. This is possible as when the RSF was created (by converting the results of the CIA queries), the file attribute of each node was recorded. Once this list is built, the nodes from each file are automatically selected and collapsed into a single node which is named after the file in question. This functionality was always a part of *Rigiedit* but the operation was entirely manual and very time consuming. This extension results in the operation being fully automated.

Figure 2: The KAC Extensions dialog.

The second major extension was the **Subsystem Allocator Tool**. This tool provides a graphical interface (shown in Figure 3) for allocating units to the particular subsystems. Subsystems may contain source files and other subsystems. The allocation may be saved and reloaded†. This is particularly useful when examining various versions of the same software system; the allocation needs only to be done once and can subsequently be reloaded. The tool also provides the ability to execute the allocation which raises the file level facts up to the subsystem level.

*RCL=Rigi Command Language, a superset of TCL/Tk

†This assumes that the two versions contain the same files, which is a limitation of the approach discussed in Section 7.

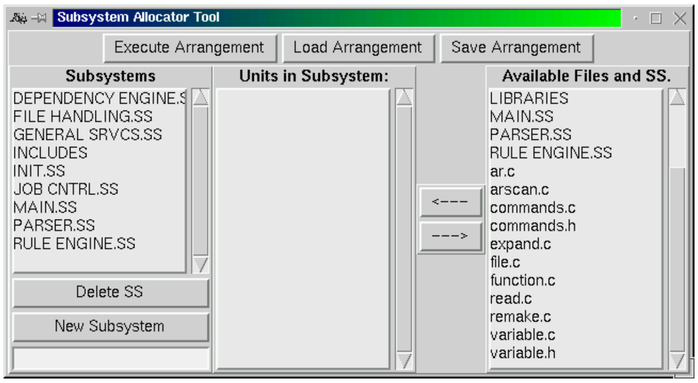

Figure 3: The SubSystem Allocator dialog.

A third extension was the ability to compare two RSF files. The idea is that the RSF files of two versions of the software would be automatically compared and a third *compare-RSF* file would be generated. This compare-RSF file contains several new relationships which are: **oldCall**, **newCall**, **missingCall**, **oldData**, **newData**, **missingData**, **oldInclude**, **newInclude** and **missingInclude**. Old relationships are relationships which existed in the older version of the software and continue to exist ion the new version. New relationships are those which are present in the newer version but were not present in the older version. Missing relationships are those which were present in the older version but are missing in the newer version.

These new relationship types were easily added to the *Rigiedit* system and allow for the evolution of a system be examined. For example you can compare two versions of a software system and then use the features of *Rigiedit* to look at the missing calls or the new calls. In this way you can visually see how the software system has evolved.

A final extension to *Rigiedit* dealt with a failing of the system. Once facts have been raised to the subsystem level you can bring them back to the file level by double clicking on a subsystem. However this only shows you the inside of the subsystem. Incoming links from suppliers and outgoing links to consumers are not shown. To compensate for this the ability was created to select multiple subsystem nodes and expand them. In this way you can see the inter-subsystem relationships at the file level. It is also possible with a single click to re-collapse things back to the subsystem level. This approach is discussed and used in the case studies which follow.

5 Case Studies

In order to validate our approach, three case studies were performed. Each of the authors chose a system and extracted facts about it using the using the *CIA*[8] toolset and converted the results to RSF as described in out previous work[1].

In the first case study we examined the Make[9] system, a commonly used utility for managing the build process of software systems. This software system was chosen because the authors have had previous experience extracting its architecture. The case study benefited from the detailed analysis presented in our earlier work[1]. In particular, the subsystem decomposition suggested in our earlier work was used. This case study allows us to demonstrate how the tool can be used by someone generally familiar with the architecture of a particular version of the software to find out information about the system's evolution. This first case study also serves as a complete example of our approach and tool extensions.

The remaining two case studies were performed on systems new to the authors. Case study two looked at the GNU Wget World Wide Web utility[9]. Finally case study three examined the Grep[9] system a commonly used text manipulation tool. These last two case studies are important because they show how our approach and tool extensions can be used to find out information about the evolution of a software system that the tool user is previously unfamiliar with.

5.1 Case Study One: The Make System

Several versions of the Make system[9] are available. In this case study we examine versions 3.75, 3.76, 3.76.1 and 3.77 by comparing them using the extensions to *Rigiedit* described above.

5.1.1 Comparing Version 3.75 and 3.76

In our first case study we examine our approach with versions 3.75 and 3.77 of Make. The RSF files for these versions are compared using the versioner tool and the resulting RSF is loaded up into *Rigiedit*. The file level tool is run to bring all of the facts to the file level and the subsystem allocator is used to further raise the facts to the subsystem level (note that this subsystem allocation was based on the detailed examination presented in [1].) The resulting RSF shows all of the new relationships discussed earlier (e.g. **oldCall, newCall, etc.**). A screen shot is shown in Figure 4.

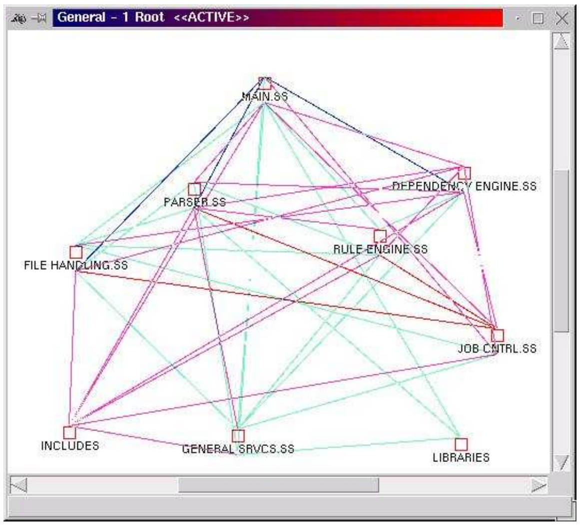

Figure 4: A comparison Make version 3.75 and 3.76 at the subsystem level.

Clearly this level of detail is not very useful. There is simply too much information represented. In practice, we believe that this tool would be used to answer particular questions rather than to give a complete picture. For example, "Are there any function calls in version 3.75 which are not in version 3.76?" (i.e. **missingCalls**) or "Are there any function calls in version 3.76 which are not in version 3.75?" (i.e. **newCalls**). The unextended *Rigiedit* easily supports these sorts of filtering questions once the new relationships are added to the proper configuration files. We are able to filter by node type and arc type; to answer these questions we want to filter by arc type.

First we look to see if there are any **missingCall** relationships. We quickly determine that there aren't any. This tells us that no function calls have been removed between versions 3.75 and 3.76. Information about **missingCall** relationships is useful for a number of reasons. In particular, if a node has only **missingCall** relationships (i.e. no **oldCalls** or **newCalls**) then most likely the node represents *dead code*. Thus the system has evolved to no longer need this particular node. The extended tool can be used to identify dead code for removal.

Next we are interested in finding out if there are any **newCall** relationships. We set the filters accordingly and get the image shown in Figure 5. This information is clearly

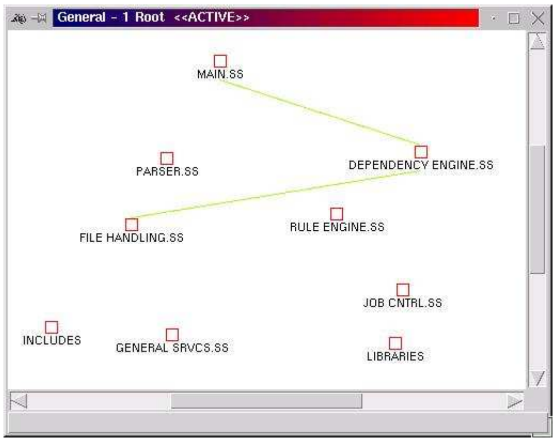

Figure 5: The **newCall** relationships between versions 3.75 and 3.76 of Make.

more useful than the information provided by Figure 4. We can easily see that there are **newCall**s between the `Main.SS` and `Dependency Engine.SS` subsystems and also between the `Dependency Engine.SS` and `File Handling.SS` subsystems. While this information is useful, we would like to know more. We want to know what files within these subsystems are involved in these **newCall** relationships. *Rigiedit* allows us to expand these subsystems and see inside but the external relationships can't be shown. For example, Figure 6 shows the inside of the `Dependency Engine` (note this figure is from [1] and shows all of the relationships arcs).

In order to make the tool more useful we provided the extensions which allow the user to select the subsystems they are interested in and have them expanded to the file level. The inverse of this operation is also provided. Expanding the subsystems we are interested in here yields the image in Figure 7.

If we want more information about a particular relationship, we can click on the link and use the *Rigiedit* feature to get more information as shown in Figure 8. This information box shows that the new call relationship between `remake.c` and `file.c` is a function call from `f_time` to `rehash_file`. This kind additional information is available for all arcs and nodes. Note that information about filtered relationships such as **oldCall** is also shown.

Just as we examined the relationship involving call relationships, we are also able to examine the software system by looking at the data reference relationships. There were no

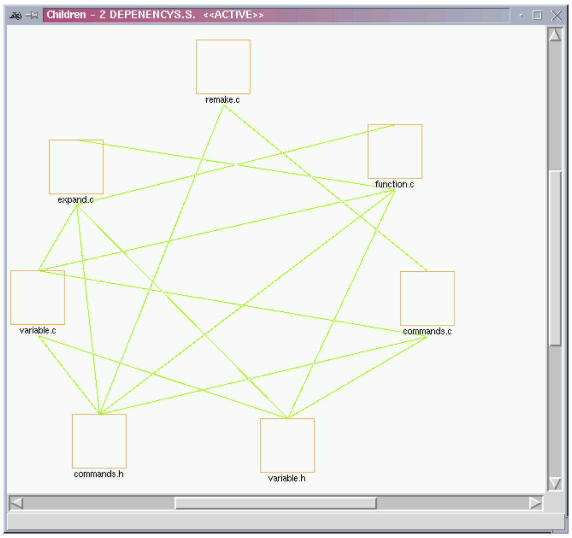

Figure 6: Inside the `Dependency Engine` subsystem of Make.

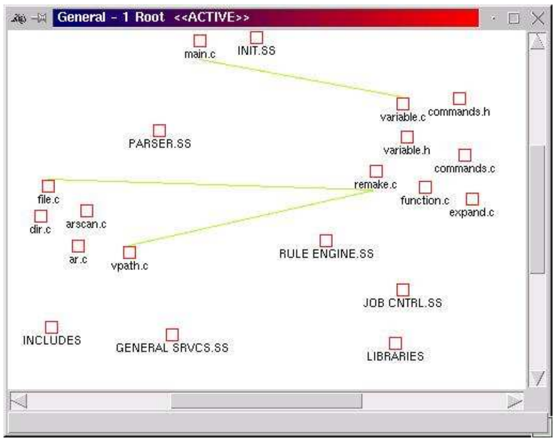

Figure 7: The **newCall** relationships between versions 3.75 and 3.76 of Make shown at the file level.

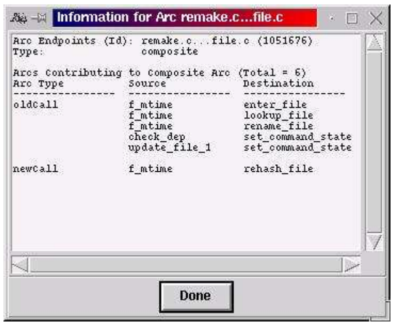

Figure 8: A *Rigi* information window showing information about a particular **newCall** relationship.

missingData relationships in this case. This means that all the data references present in version 3.75 are also present in version 3.76. A missing relationship here would be useful in that it might for example identify that particular subsystems are no longer dependant on each other. This sort of information would be useful in an evolving system since a subsystem on which nothing depends can be more easily changed.

In the same way it is useful to examine new data reference relationships which have developed as the result of evolving from version 3.75 to 3.76. These relationships are important because they may show new dependencies between subsystems which did not exist before. Figure 9 shows the **newData** relationships present here. From this figure, we are able to see that there are three **newData** relationships at the subsystem level. The `Main.SS` subsystem is referencing data defined within the `Include.SS` subsystem and the `Dependency.SS` subsystem. The `Dependency.SS` subsystem is also referencing data defined in the `Include.SS` subsystem.

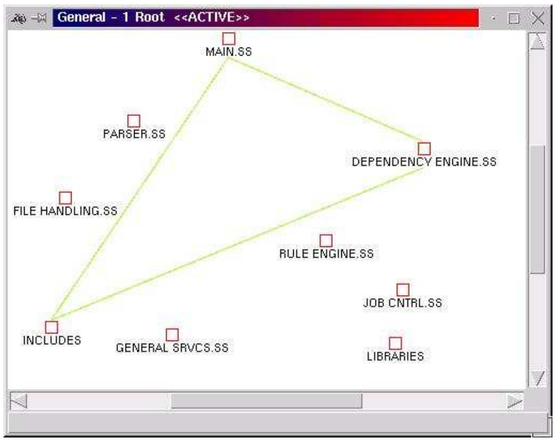

Figure 9: The **newData** relationships between versions 3.75 and 3.76 of Make.

Once again we find that this information is useful but we would like to know which files are involved in these relationships. The extended tool allows us to easily expand the subsystems of interest to get the results shown in Figure 10. From this figure we can see that `main.c` in the `Main.SS` subsystem is referencing data defined in the file `make.h` and `variable.h`. We can also easily see that `remake.c` of the `Dependency.SS` subsystem is referencing data defined in `make.h`. If we require more information about any of these relationships, we can select them and ask *Rigi* to provide more information as we did for **newCalls** (shown in Figure 8).

Finally we are able to examine the difference in the include relationships. Using the extended tool we can quickly determine that the only differences in this area is that there is a **newInclude** relationship between the `Main.SS` and `Include.SS` subsystems. Examining this more closely we see in Figure 11 that `version.c` (which is within the `INIT.ss` subsystem inside the `Main.SS` subsystem) is including `config.h`. This relationship shows that there is a new dependency involving the `config.h` header file.

5.1.2 Comparing Version 3.76 and 3.76.1

For the second part of this case study, we attempt to test our tool over a very short evolution interval. To do this we compare‘ versions 3.76 and 3.76.1 of Make. The results of this

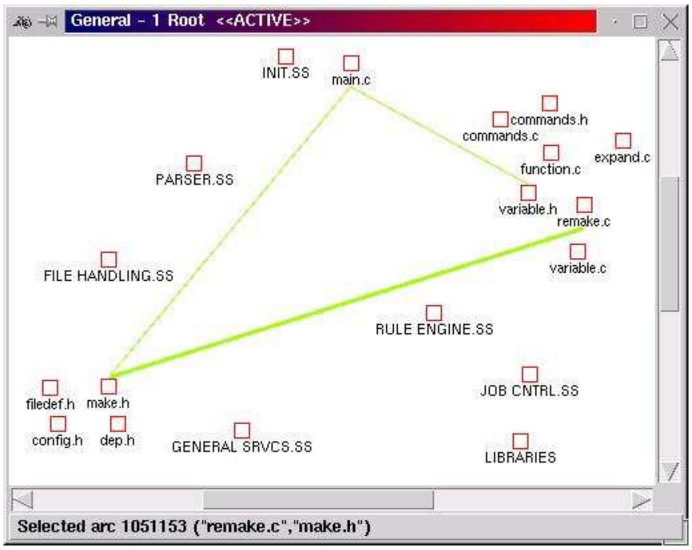

Figure 10: The **newData** relationships between versions 3.75 and 3.76 of Make shown at the file level.

comparison are at first disappointing. The tool did not find any differences between the RSF of these two versions. This means there are no new or missing call, data, or include relationships. Initially we felt this was a weakness in our approach. However, the fact that nothing has changed is itself a useful piece of information.

This kind of information can confirm to a software system maintainer that no dependencies have changed between versions. In such a case this information surely would not be a surprise to the maintainer; the tool and approach can be used to ensure that the evolution process hasn't accidently changed something.

5.1.3 Comparing Version 3.75 and 3.77: A more detailed Analysis

For our final look at the Make system we examine two versions separated by a longer evolution period. Although the difference is only 0.02 in version numbering, nearly two years separate these versions. In addition change log information suggests that the lead developer was different for each version. These two versions have differences in all categories; however, we will examine only the relationships related to function calls.

At the subsystem level, we can see from Figure 12 that there are two **missingCall** relationships. Function calls which existed between the `Dependency Engine.SS` subsystem and the `Parser.SS` and `General Srvs.SS` subsystems in version 3.75 are not present in version 3.77. This information could merely indicate the evolution of the software or it could identify an error. For example a function call may have been commented out during debugging and not put back in.

We gain more information by looking at these missing relationships at the file level as shown in Figure 13. Once we have expanded the subsystem in question we can see that the missing call from the `Dependency Engine.SS` subsystem and the `Parser.SS` subsystem is a call from `expand.c` to `read.c`. Similarly we see that the missing call from the `Dependency Engine.SS` subsystem to the `General Srvs.SS` subsystem is actually a set of missing calls.

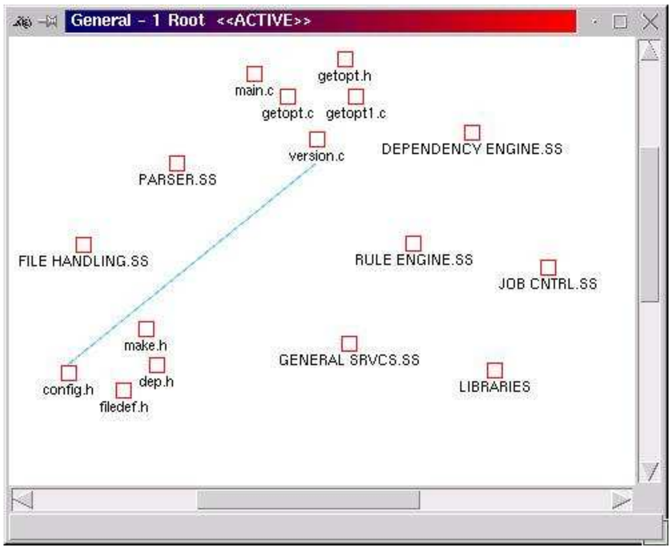

Figure 11: The **newInclude** relationships between versions 3.75 and 3.76 of Make shown at the file level.

Each of `expand.c`, `variable.c` and `remake.c` are no longer calling `misc.c`.

Expanding the `Dependency Engine.SS` subsystem also allows us to see that there are missing calls within this subsystem. The extensions to the tool allow us to quickly determine what relationships if any are missing and to graphically represent these relationships in *Rigi*. A maintainer can easily find out more about any of these missing relationships. For example, clicking on the link between `expand.c` and `misc.c` and requesting more information about it produces the *Rigi* information dialog shown in Figure 14.

This dialog box provided by the unextended tool gives us lots of information, including information about the new relationships we have added. In particular we see that the **missingCall** is a call from `variable_expand` (in `expand.c`) to `find_percent` (in `misc.c`). In this same dialog box we can see that there is a **newCall** relationship between these same two files. A new function called `variable_expand_` is now calling `find_percent`. With only this information we can speculate that `variable_expand_` has replaced `variable_expand`. At the very least this information would identify an area of interest. The maintainer could then examine the **newCall** relationships shown in Figure 15.

There is a lot of information provided in this figure. Probably this is too much to help someone track down particular details. *Rigi* supports many kinds of filtering, so it is very easy to select only the `General Srvs.SS` and `Dependency Engine.SS` subsystems and hide everything else. This is what we did, and we expanded the two subsystems of interest to give us the information shown in Figure 16. This view clearly shows the **newCall** relationships between the files in the `Dependency Engine.SS` subsystem and `misc.c`. Notably, it shows the new call from `expand.c` to `misc.c` which was evident in the information dialog of Figure 14.

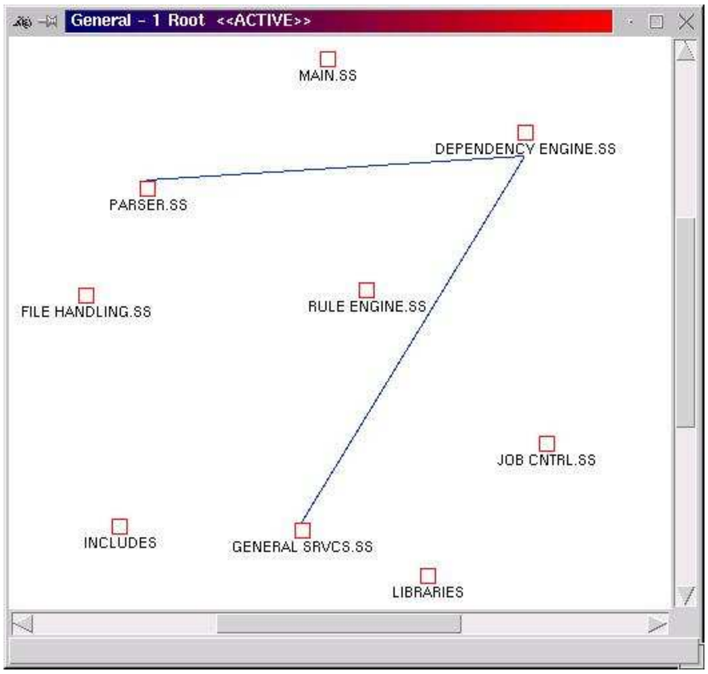

Figure 12: The **missingCall** relationships between versions 3.75 and 3.77 of Make.

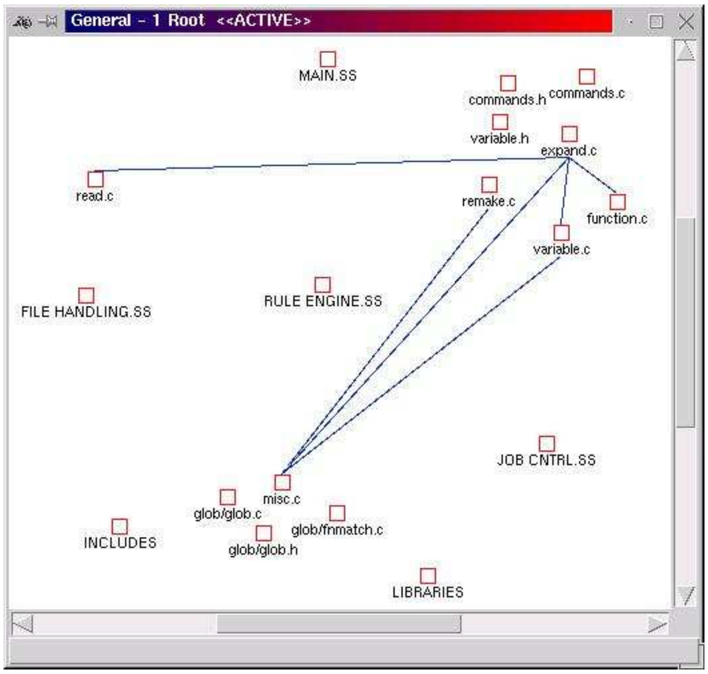

Figure 13: The **missingCall** relationships between versions 3.75 and 3.77 of Make shown at the file level.

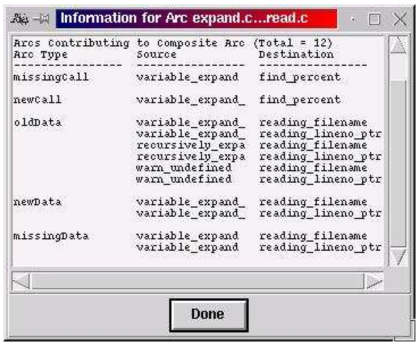

Figure 14: Information about missing relationships between versions 3.75 and 3.77 of Make.

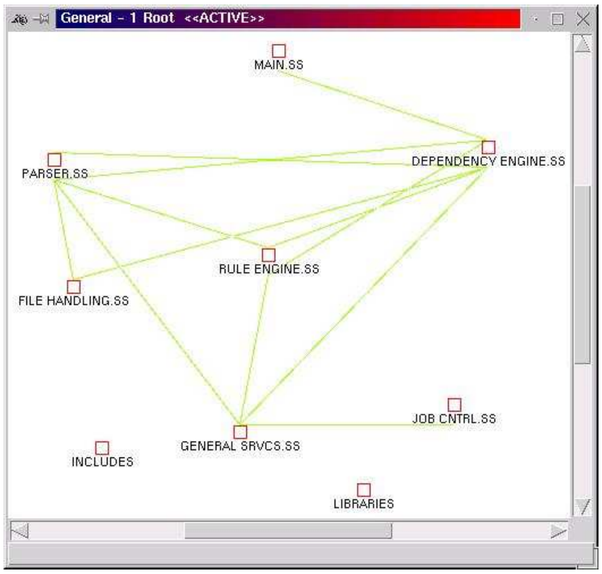

Figure 15: The **newCall** relationships between versions 3.75 and 3.77 of Make.

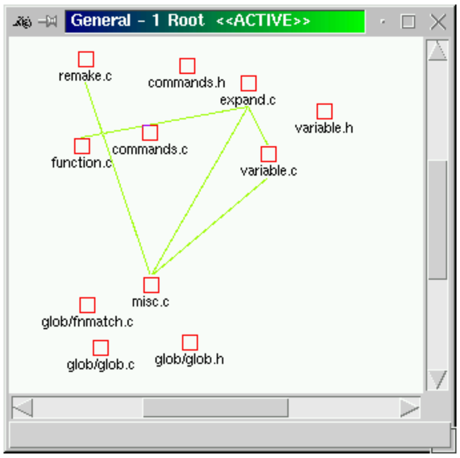

Figure 16: Some of the **newCall** relationships between versions 3.75 and 3.77 of Make shown at the file level.

5.1.4 Conclusions Case Study One: Experiences with KAC and Make

The results of this case study show that both our approach and tool extensions have been reasonably successful. The approach of comparing the RSF of two versions of Make was useful in that it clearly identified how the software had evolved. In particular the removal and addition of function calls between versions was quickly apparent.

The extensions to *Rigiedit* made it possible to easily lay out and visualize the evolution architecture of Make. Specifically we were able to raise facts to the file level and creates subsystem decomposition without much effort. Being able to expand selected subsystems was extremely important when trying to determine which files within a group of subsystem had changed through the evolution.

5.2 Case Study Two: The Wget System

GNU Wget[9] is a free utility to retrieve files from the World Wide Web using HTTP and FTP. Wget is very useful for mirroring HTML pages as well as FTP sites. Wget works non-interactively and can thus be run automatically using scripts.

Several versions of Wget are available. Versions 1.5.0 and 1.5.1 are examined and compared in this case study to determine in which ways Wget evolved.

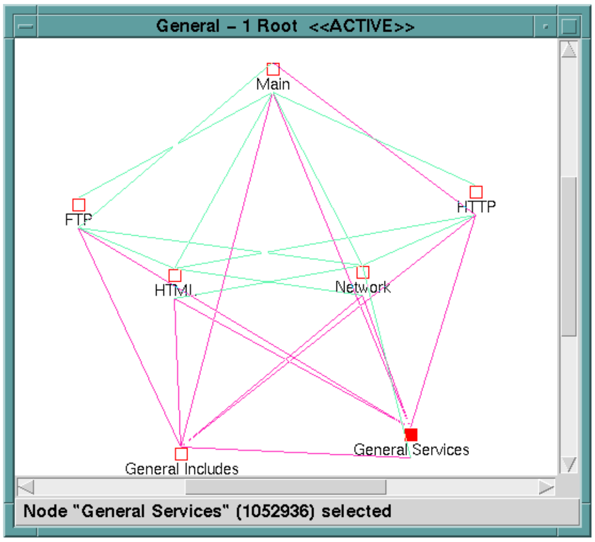

Figure 17: Subsystem decomposition for Wget

5.2.1 Subsystem Decomposition

As can be seen in Figure 17, Wget is composed of the following five primary subsystems:

1. **Main.** This subsystem contains the initialization codes including the reading and parsing of configuration files. It also controls the flow of execution.

2. **Network.** This subsystem contains general low-level network utilities including resolving host names.

3. **HTML.** This subsystem contains code for handling and parsing HTML pages and their headers as well as general URL handling.
4. **FTP.** This subsystem contains the driver for using the FTP protocol to interact with FTP servers and retrieve files.
5. **HTTP.** This subsystem contains the driver for using the HTTP protocol to interact with web servers to retrieve web pages.

There are also the General Services and General Includes subsystems which contain general utilities and include files respectively.

5.3 System Evolution from Wget 1.5.0 to 1.5.1

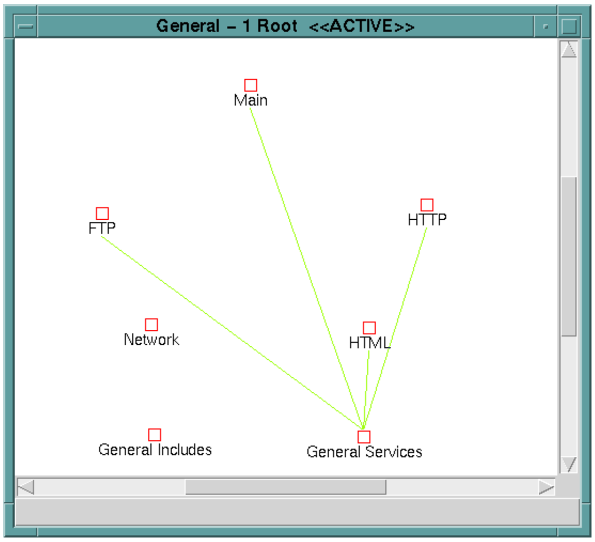

Figure 18: New calls in version 1.5.1

Upon examining Figures 18 and 19, it is apparent that there is a strong correlation between the missing calls and the new calls. By examining these calls in further detail, it can been seen that the majority of the changes are due to functions being renamed. For example, `has_wildcards` has been replaced by `has_wildcards_p` as can be seen in Figure 20.

Much the same observation can be made when examining the new and missing data items. As can be seen in Figure 21, several data items have been renamed in version 1.5.1.

A different type of dependency change can be observed when examining the changes to the include relationships in Wget. As can be seen in Figure 22, all references to the `src/ftp-basic.h` include file have been removed in version 1.5.1. In fact, the include file itself has been entirely removed and no longer exists. Thus, under some circumstances the KAC tool extensions can be used to identify files which are no longer used or no longer exist.

5.3.1 Case Study Two Conclusions: Experiences with KAC and Wget

The use of the KAC tool assisted in readily identifying the relationships which had been changed between versions 1.5.0 and 1.5.1 of GNU Wget. It was particularly effective in indi-

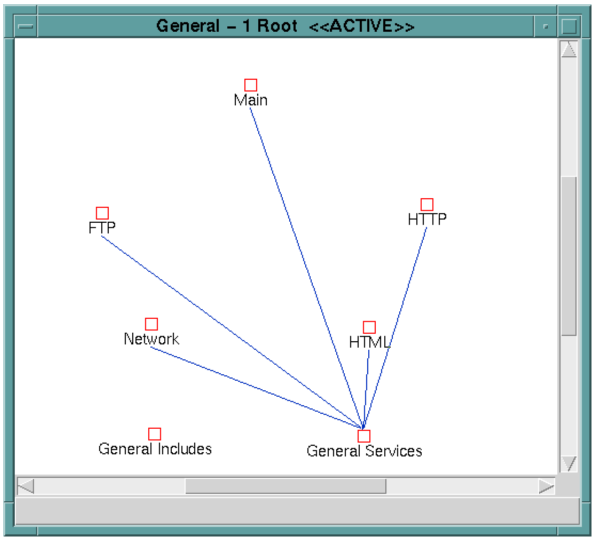

Figure 19: Missing calls in version 1.5.1

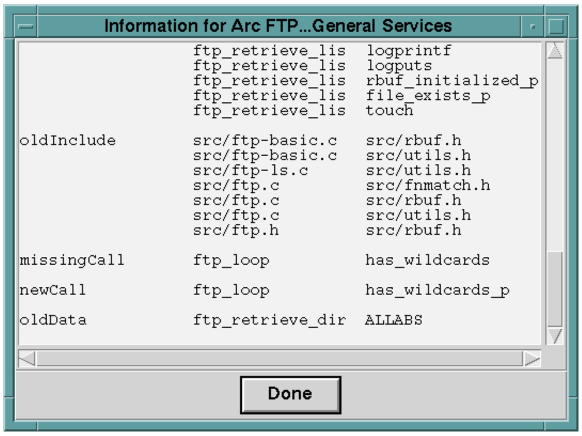

Figure 20: A missing call dialog in version 1.5.1

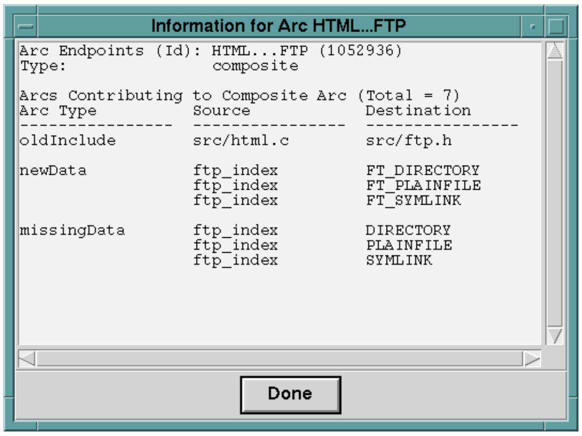

Figure 21: Data item renaming in version 1.5.1

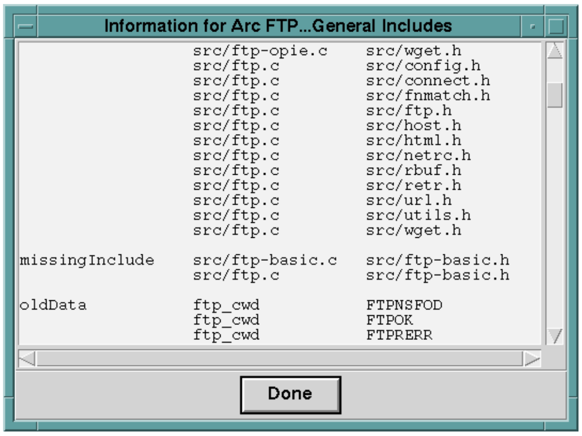

Figure 22: Include file removal in version 1.5.1

cating cases where a function had been renamed or replaced. An additional and unexpected benefit was that the removal of some of the header files was clearly indicated.

5.4 Case Study Three: The Grep System

`Grep`[9] is a well known text parsing tool and regular expression engine which is available for many different platforms. In this case study we examine versions 2.0, 2.1 and 2.2 following our approach and tool extensions.

The evolution process often results in new source files being added, sometimes erroneously. In this final case study we explore the addition of new source files to an evolving system. In particular, we will be using the `KAC` tool extensions to quickly identify a new source file `btowc.c` introduced in version 2.2 which is not even used!

5.4.1 Subsystem Decomposition

The `Grep` system is divided into three primary subsystems and a `Libraries` subsystem. These are: `Memory Management.SS`, `Parser.SS`, and `Main.SS`.

- The `Memory Management` subsystem contains functions and files related to memory allocation and garbage collection.
- The `Parser` subsystem contains functions and files used in parsing and searching a given stream of text.
- The `Main` subsystem contains functions and files that provide the main functionality of `Grep`.

The subsystem decomposition as viewed with *Rigiedit* is shown in Figure 23.

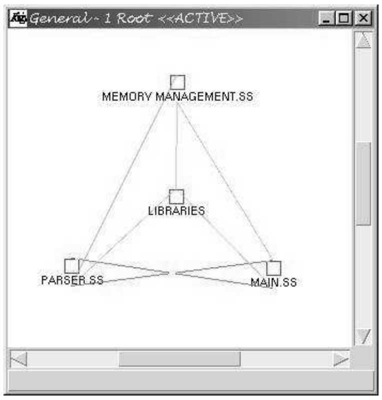

Figure 23: The subsystem decomposition for `Grep`.

5.4.2 System Evolution from Version 2.0 to 2.1

Close examination of the source files of these two versions results in the identification of several new source files in version 2.1. In particular, the files `getopt1.c` and `memchr.c` are introduced to the `Grep` system in version 2.1.

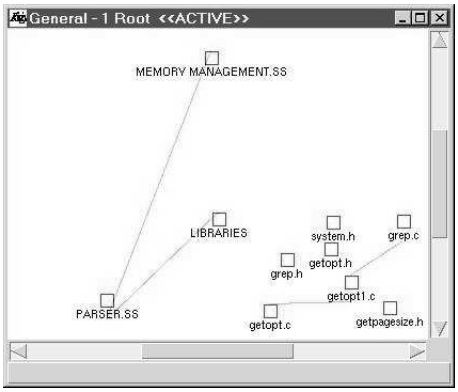

Figure 24: The **newCall** relationships between versions 2.0 and 2.1 with the `Main.SS` subsystem expanded.

We use the *Rigiedit* tool to navigate the compare-RSF resulting from running the `versioner` tool extension. The tool allows us to go directly to the `Main.ss` subsystem which is of interest to us since this is the subsystem to which `getop1.c` is allocated. The source file `getopt1.c` logically belongs to the `Main.SS` subsystem because it extends the existing command-line option parser module `getopt.c`.

Once we have expanded the `Main.SS` subsystem, we are able to examine the various new relationships such as **newCall** or **missingData**. We place the file `getopt1.c` in the `Main.SS` subsystem and confirm that it introduces some **newCalls** (from `getopt1.c` to `getopt.c`) to the system as well as receives some **newCalls** (from `grep.c` to `getopt1.c`). It is fairly straight forward to use the `KAC` tool extensions to find out more information about the **newCall** relationship that goes from `grep.c` to `getopt1.c` by selecting the two nodes and clicking on the `Expand Selection` button as seen in Figure 24. Doing so provides us with the additional detailed information that the function `main` in `grep.c` made a **newCall** to the function `getopt_long` in `getopt1.c`.

5.4.3 System Evolution From Grep 2.1 to 2.2

When comparing versions 2.1 and 2.2 we note that the evolutionary process had resulted in a new source file being introduced to the system. This source file is `btowc.c`.

We have determined that `btowc.c` logically belongs to the `Parser.SS` subsystem. According to the description provided, `btowc.c` provides the multi-char support in parsing. Figure 25 demonstrates the `Parser` subsystem (in its expanded state) in relation to the other subsystems in version 2.2. The figure shows all of the possible relationships in this system extracted by the `KAC` tool.

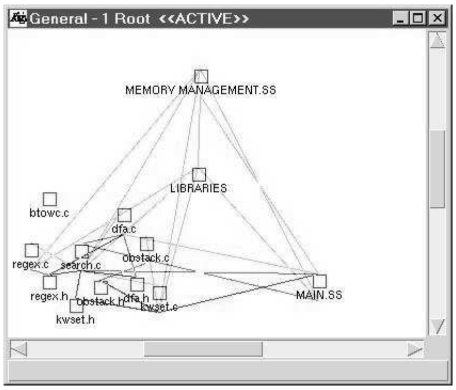

Figure 25: All relationships between versions 2.1 and 2.2 in the `Grep` system.

It turns out that the new file, `btowc.c`, is not used in version 2.2 at all[‡]. This information becomes apparent when we try to view all of the relationships associated with this file. Figure 25 shows that the source file `btowc.c` does not relate to any other part of the system. This assertion is confirmed by removing the source file and successfully building the system. The resulting executable is identical to the one built with this file included.

5.4.4 Conclusions Case Study Three: Experiences with KAC and Grep

Using the `KAC` tool we were able to quickly set up the three versions of the `Grep` system for evolutionary analysis. In fact, it is straight forward and natural to add future versions into the analysis by adjusting the overall layout of the subsystems.

Once the compare-RSF file was loaded we were able to use *Rigiedit* and the `KAC` tool extensions to find out very interesting information about the evolution of the *Grep* system. Specifically we were able to identify source files which were added but never used. This sort of information would be very important to a software engineer responsible for maintaining this system.

[‡]On re-examination, it was determined that this particular source file is only used on Solaris platforms to provide specific functionality left out of the Solaris version of the *libc* library. The case studies presented in this paper were executed on the Linux 2.0 operating system where this file is not required. It is therefore not surprising that this source file was in fact not used.

6 Limitations and Future Work

6.1 Limitations

Our approach has several limitations which need to be further examined. The most significant of these is that we assume that no files will be added or removed between versions of a software system. The current tool extensions make little attempt to handle this situation. As was seen in case studies two and three, the extensions will support this situation in some cases but not in general. In our approach we also assume that the subsystem decomposition does not change significantly between versions. This assumption is made in order to simplify the visualization process and is not really necessary. However, removing this assumption requires that we decide how several things would be represented. In particular, how would we represent new subsystems, deleted subsystems, non deleted files within a deleted subsystem, files which have been moved between subsystems, and new files which are not assigned to any subsystem.

A second major limitation of our approach is that it is little help if there have been no changes at any interface. That is, changes which are limited to the code within a function are not detectable by our approach.

6.2 Future Work

There are several areas for future investigations. The limitations discussed above require detailed consideration in order to determine how our approach should be improved. In addition to this work, there are possible areas of interest regarding the tool extensions. In particular, we did not take advantage of two areas we had originally considered.

The RSF which was created by converting the results of *CIA* queries contains versioning information about every node. It is therefore possible to load any number of versions into *Rigiedit* at the same time. Once this had been done we would be able to filter based on relationship types and also version information. For example: "Show me all the new calls and old calls in versions 3.75 and 3.77". Further tool extensions could support questions like: "Show me old calls which appear in all of the currently displayed versions." Time restrictions prevented us from investigating this area to determine how useful it is.

A second area which could be investigated involves using current *Rigi* annotations to aid in the redocumentation of the evolved systems. Every node could be automatically annotated to provide information about its evolution over several versions. This information could easily be extracted from the RSF of the versions in question. This would allow a software engineer to visualize the current architecture while being able to easily access evolution information which would be automatically documented in each node.

7 Conclusions

We have presented an approach for comparing two versions of a software system in order to visualize how the system has evolved. In order to investigate this approach we extended *Rigiedit*, the visualization tool which is part of the *Rigi* systems. These extensions proved to be very useful in exercising our approach.

We illustrated our approach through the use of three separate case studies which demonstrated both the technique and the tool extensions. The case studies provided positive support for our approach and tool extensions in that we were able to find out very useful information about the evolution of the systems examined. In particular we showed that we were able to detect the replacement of functions by new functions, and the addition and removal of function call and data dependencies. Finally, the `KAC` extensions surprised us by being able to detect and represent the addition and removal of source and header files. We had not expected that our tool extensions would handle these last two changes without modification; this is why we have listed these cases as limitations above.

In practice large software systems either evolve or become obsolete. Those that evolve pose a difficult challenge for the software engineers who are responsible for maintaining them. Being able to monitor the evolutionary process in a more or less automated fashion makes this otherwise daunting task manageable. We were able to quickly make use of our approach and tool extensions to visualize the evolution of three small software systems. The tool extensions enable the approach to scale to larger systems with minimal effort.

Any effort to monitor the evolution of a software system will have to provide the abilities to query and navigate the architecture within a visual environment. *Rigiedit* provides an excellent base which, when extended with `KAC`, will allow our approach to scale to larger software systems.

References

[1] Adams, M., Keast J., "Make 3.77 An Architecture Extraction with the CIA Toolset", `http://keiths.uwaterloo.ca/james/courses/746h/make`.

[2] Ernst, M., Cockrell, J., Griswold, W. G., Notkin, D., "Dynamically Discovering Likely Program Invariants to Support Program Evolution", Technical Report UW-CSE-98-09-03, University of Washington, Seattle WA, 1998.

[3] Garlan, D., Shaw, M., "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, Volume 1, World Scientific Publishing Co., 1993.

[4] Murphy, G., C., Notkin, D., Sullivan, K., "Software Relflection Models: Bridging the Gap between Source and High Level Models", *Proceedings of the Third ACM Symposium on the Foundations of Software Engineering*, 1995.

[5] Murphy, C., Notkin, D., "Lightweight Lexical Source Model Extraction", *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, July 1996, p.262-292.

[6] Shaw, M., Clements, P., "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", C.S. Dept and Software Engineering Institute, Carnegie Mellon Univ, Pittsburgh, PA 15213, 1996.

[7] Schwanke. "An Intelligent Tool for Re-engineering Software Modularity", *In proceedings of the 13th International Conference on Software Engineering*, pages 83-92.

[8] **The CIA homepage**, *AT&T*, `http://www.research.att.com/ciao/`.

[9] **The Free Software Foundation**, `http://www.gnu.org`.

[10] **The PBS Project**, `http://turing.toronto.edu/holt/pbs/Welcome.html`

[11] **The Rigi Project**, `http://tara.uvic.ca`.