GNU Make 3.7.7

An Architecture Extraction with the CIA Toolset

Mark G. Adams (mgadams@se.math.uwaterloo.ca)
James P. D. Keast (james@keiths.uwaterloo.ca)
`http://keiths.uwaterloo.ca/james/courses/746h/make/`

21 October 1998
Department of Computer Science
University of Waterloo

at
Waterloo, Ontario, N2L 3G1

ABSTRACT

This report contains the results of a software design extraction which was performed on *GNU Make 3.77* [3] using the **CIA** toolset[2]. First a naive decomposition is presented which was created by the authors using no special tools. This decomposition was created by looking at the source code and by using simple Unix* tools such as `grep`. This informal analysis was combined with the authors' own understanding and some guidance from the **Bunch** automated clustering tool[1] to form the *conceptual* architectural decomposition presented here. Next the real or *concrete* architecture was extracted using tools from the CIA toolset. This architecture was then visualized with the Rigiedit editor from the Rigi[4] toolset. The approach used in performing this extraction is discussed. Finally a discussion is presented as to how this sort of architecture extraction may be best performed.

Keywords: **Software Design Extraction, Architecture Extraction, CIA, Rigiedit.**

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

---

* "Unix" is a trademark of AT&T Laboratories and Unix Labs.

Contents

List of Figures

1 Introduction

1.1 Purpose

The purpose of this paper is to take a publicly available application which is well-used, develop a conceptual architecture based on its documentation and source code, and finally to produce the concrete architecture using CIA and identify any discrepancies.

1.2 Overview

GNU Make 3.77 was chosen to be examined as it is a very commonly used tool to manage the compilation of software systems. In addition, the source code is organized into thirty .c files plus header files. In addition, the size of each file is small enough that they can be examined, queried and visualized in a reasonable amount of time.

As a first step, a database of the interactions between the source files was built using CIA. These interactions were then extracted, reformatted and passed to *Bunch*[1] which produced a preliminary decomposition based on which files had direct interactions. Subsequently, the source files were examined and grouped into subsystems based upon their functionality, the interactions we expected, the suggested clustering from Bunch and our experience. This conceptual decomposition was then drawn graphically.

The concrete architecture was then constructed by querying the functional and data relationships between the various elements of the source program (functions and variables) from the CIA database, reformatting the results and passing them into Rigiedit. The elements were then collapsed into their respective files which were then grouped into subsystems based on their interconnections and the conceptual architecture.

Finally, the conceptual and concrete architectures were compared to locate any discrepancies. Our experiences with the various tools were summarized and potential improvements were discussed.

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

Two different "boxes and lines" notations are used in this paper.

The first notation is used in the presentation of the conceptual architecture in Section 2. The boxes represent subsystems of the system being discussed. The lines and arrows between the boxes represent expected call relationships between the subsystems. An arrow indicates that a function call was expected from one subsystem to another (in the direction of the arrow).

The concrete architecture is presented in Section 3. This architecture was visualized using Rigiedit. In this notation, boxes 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 that 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 Conceptual Subsystem Decomposition of GNU Make 3.77

The authors decided to use Bunch to produce a preliminary subsystem decomposition which could then be refined based on examination of the source code files. Thus, the first step was to produce a database of the interactions between the source files using CIA.

The script in figure 1 was created and used to create the database. The first two defines `-DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"'` were required by the source files, as they would normally have been provided from the makefile. The third define, `-U__GNUC__`, was required as one of the included Linux header files depended on a feature of the GNU C compiler which CIA could not handle, and thus the generic version of the header file had to be specifically enabled. Determining these defines was a non-trivial exercise.

```
#!/bin/ksh
set -x
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c main.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c commands.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c job.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c dir.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c file.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c misc.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c read.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c remake.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c rule.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c implicit.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c default.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c variable.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c expand.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c function.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c vpath.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c version.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c ar.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c arscan.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c remote-stub.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c signame.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c getopt.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c getopt1.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c getloadavg.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c glob/fnmatch.c
cia -I. -DLIBDIR='"/usr/local/lib"' -DINCLUDEDIR='"/usr/local/include"' -U__GNUC__ -c glob/glob.c
cia ar.A arscan.A commands.A default.A dir.A expand.A file.A fnmatch.A function.A getloadavg.A
       getopt.A getopt1.A glob.A implicit.A job.A main.A misc.A read.A remake.A remote-stub.A
       rule.A signame.A variable.A version.A vpath.A
set +x
```

Figure 1: Commands to create the CIA database for GNU Make.

Once the database had been created, it was queried to determine the inter-file function calls as follows:

```
cref -u fu - fu - file1=!(/usr/*) file2=!(/usr/*) | cut -d';' -f5,18 |
        uniq | sed -e 's/;/ /' > calls.txt
```

where the restrictions on the filenames were to filter out the system include files. `cut` was used to pick out the fields containing the files of the caller and callee functions respectively. `uniq` was used to remove the duplicates, since Bunch (apparently) does not recognize the concept of multiple links between files. Finally, `sed` was used to change the delimited to a space. The end result was a text file containing pairs of filenames representing a call from the first file to the second file. This text file was fed into Bunch which then produced the subsystem decomposition presented in figure 2.

With the Bunch decomposition in hand as a starting point, the source files for Make were then examined manually to determine their functionality. They were grouped into subsystems to produce the conceptual architecture presented in figure 3.

3 Extracting the Concrete Architecture

3.1 Approach

The following queries were run on the CIA database produced earlier:

```
cdef -u - - file=!(/usr/*) > defs.cdef
cref -u fu - fu - file1=!(/usr/*) file2=!(/usr/*) > functionCalls.cref
cref -u fu - var - file1=!(/usr/*) file2=!(/usr/*) > useVar.cref
```

The first query uses `cdef` to extract the definitions of all of the elements in the database (specifically the functions and variables) and locate in which files they are located. The second query uses `cref` to extract all of function calls, while the third query extracts all of the variable references.

3.2 Architectural Visualization

The authors chose to use Rigiedit[4] to visualize the extracted architecture. This tool was much more powerful than the Dot tool which comes with CIA, since Dot is little more than a graph description program. In order to visualize the extracted architecture, the output from the above queries had to be converted to Rigi Standard Form (RSF)[†]. This was easily accomplished using a simple `gawk` script.

Loading the extracted architecture into Rigi gave us the mess shown in figure 4. In this graph, the nodes are data items and functions, and the lines are call, data reference and include relationships. This level of granularity was too fine and does not help in understanding the system. The relationships can be easily brought up to the file level using Rigiedit's select, filter and collapse utilities; however, this is a very time consuming process and has to be repeated every time the queries were modified. For this reason, a script was written in the Rigi Command Language (RCL)[‡].

[†]RSF is a tuple notation ideal for representing these relationships.

[‡]RCL is based on TCL and has a very poorly documented API. The script in question was written by examining

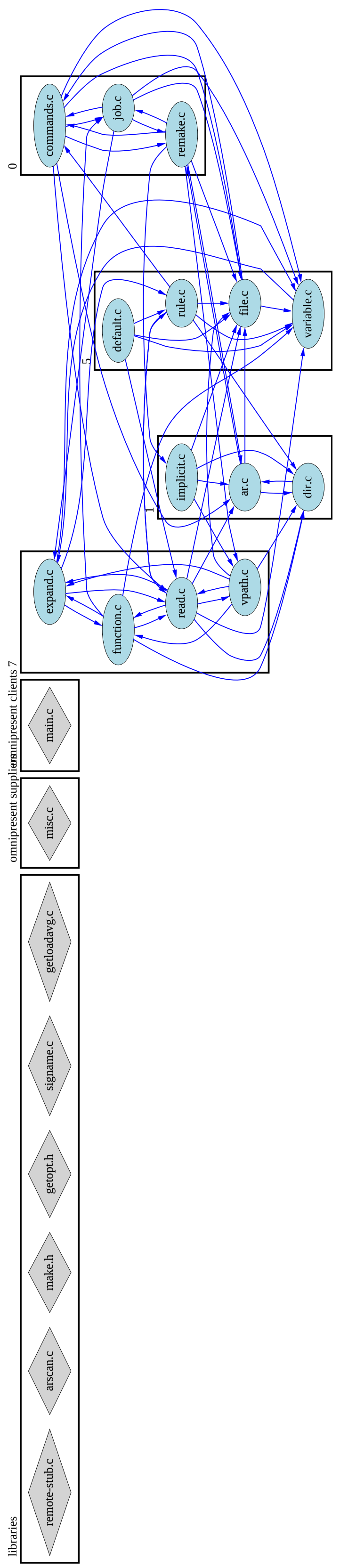

Figure 2: Bunch Subsystem Decomposition

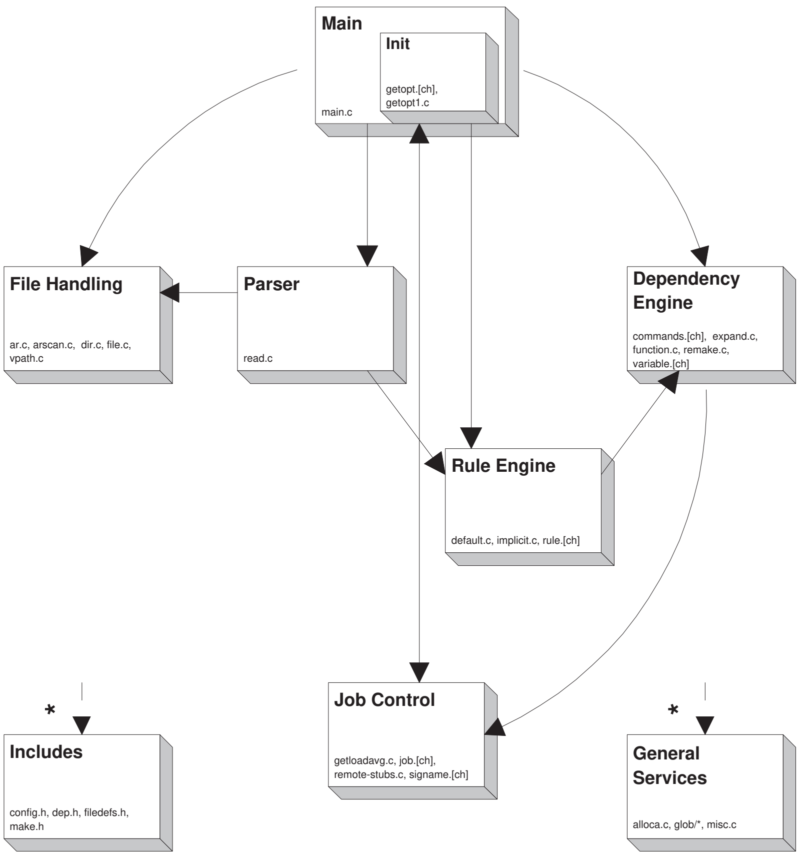

Figure 3: The conceptual architecture of GNU Make 3.77

.

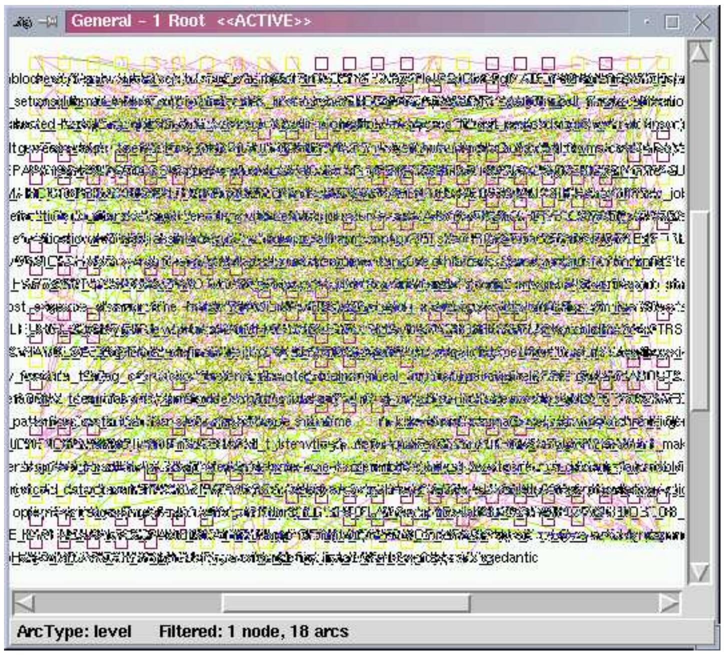

Figure 4: The extracted architecture of Make as first visualized with Rigi.

Bringing the relationships up to the file level by running the RCL script produces the figure shown in figure 5. The figure is now easier to understand, but it clearly needs to be better organized so that a better sense of the architecture can be developed. This organization is performed by identifying subsystems and grouping the files into the subsystems as described in Section 3.3. Rigiedit provides an ideal environment for performing this decomposition since it permits the nodes to be moved on the screen so that different configurations can be easily examined and evaluated while maintaining all relevant information about the relationships.

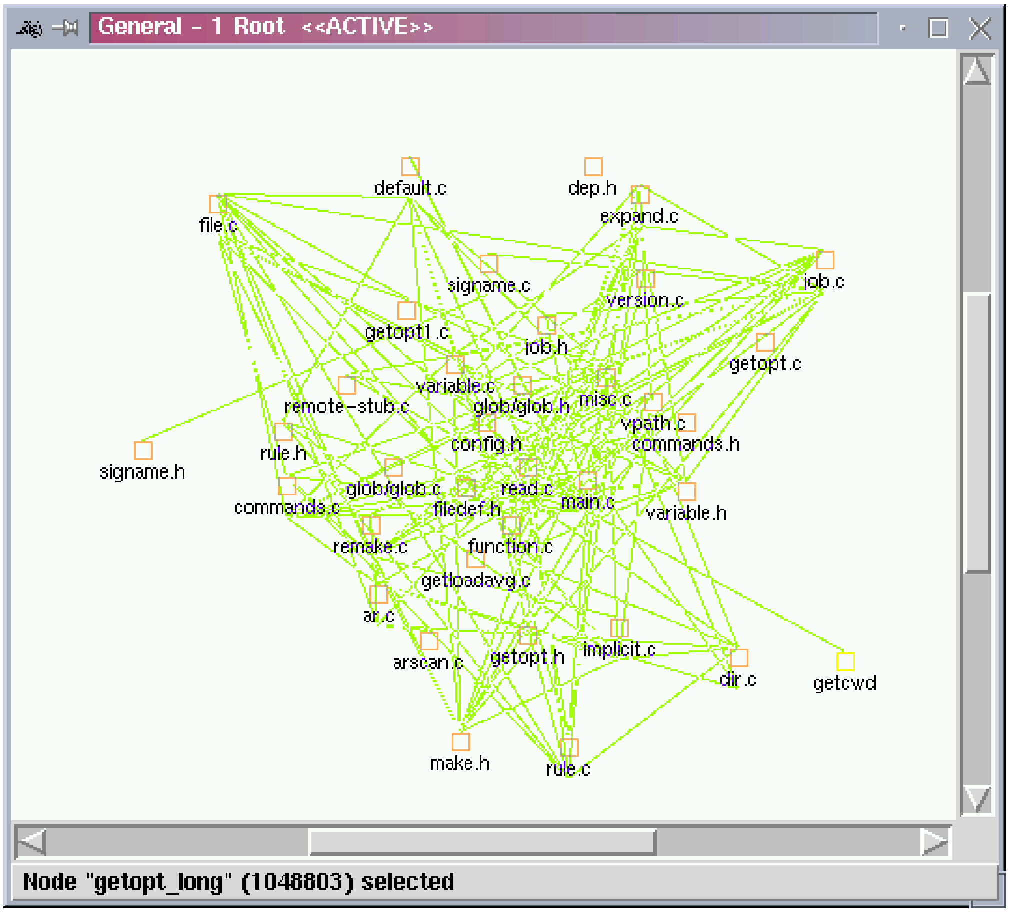

Figure 5: The extracted architecture of Make as visualized with Rigi at the file level.

3.3 Justification for the Proposed Subsystem Decomposition

3.3.1 The Overall Subsystem Decomposition

Figure 6 shows the subsystem decomposition in Rigiedit which was the result of the above grouping of files. This figure can be better understood by ignoring the Includes subsystem and the General Services subsystem. These subsystems contain header files and utility functions used by all of the other subsystems, and thus do not contribute to a better understanding of the system. Figure 7

---

undocumented RCL examples provided with the Rigi distribution.

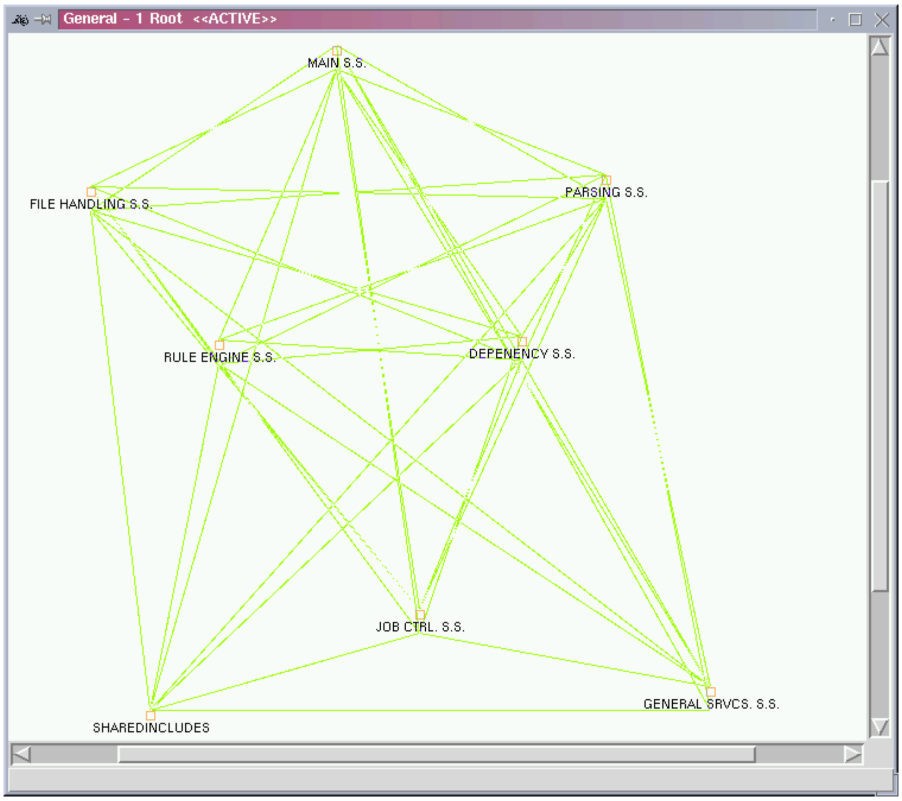

Figure 6: The subsystem decomposition.

shows the decomposition with those subsystems removed.

3.3.2 The Main Subsystem and its Init Subsystem

The Main subsystem in figure 8 consists of two units: the `main.c` source file and the Init. subsystem which is discussed below.

The Main subsystem contains the file `main.c` which contains functions that interface with virtually every other subsystem. This file is responsible for performing all of the initialization (calls to functions in the Init. subsystem) and also for starting all necessary processes. The responsibility of initiating all processes results in `main.c` calling functions in the File Handling, Rule Engine, Dependency Engine and Parsing subsystems as well as all of the support subsystems.

The Init. subsystem in figure 9 is responsible (as its name would suggest) for performing the various initializations as required. This subsystem contains the following files:

1. version.c
   This is a source file which contains no instructions; only declarations of the constants used to identify the version number of this Make system are in it.

2. getopt.h
   This is a header file which contains the prototypes of functions for command line argument

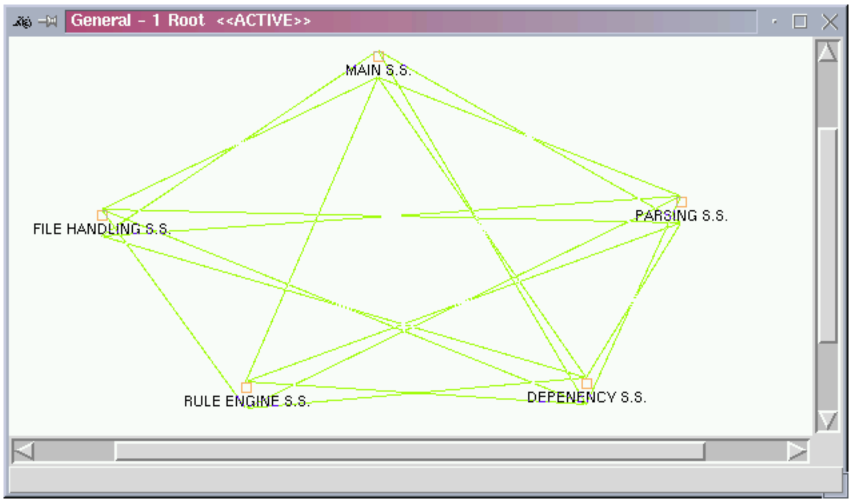

Figure 7: The subsystem decomposition with General Services and Includes removed.

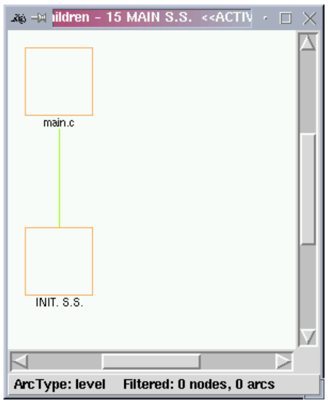

Figure 8: The Main subsystem.

manipulation.

3. getopt1.c
   This file implements the functions defined in `getopt.h`.

These files were all placed in the Init. subsystem because it was apparent that they were responsible for performing initializations which occurred only at startup. No CIA queries were used to support this position, and it disagrees with the layout suggested by Bunch in that Bunch wanted to place these files in libraries. However, this structure reflects the actual structure of the system as the authors understand it.

The Main subsystem contains only the `main.c` file because it is clearly responsible for starting all of the operations of the system. It performs the necessary initialization and then makes calls to all of the other subsystems as necessary.

3.3.3 The File Handling Subsystem

The File Handling subsystem is composed of the following files:

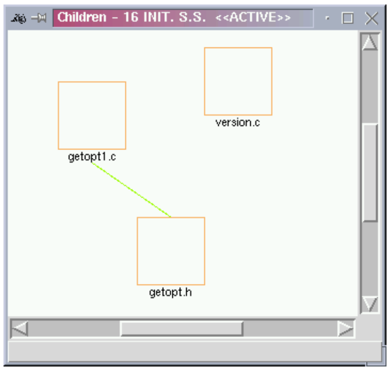

Figure 9: The Init. subsystem.

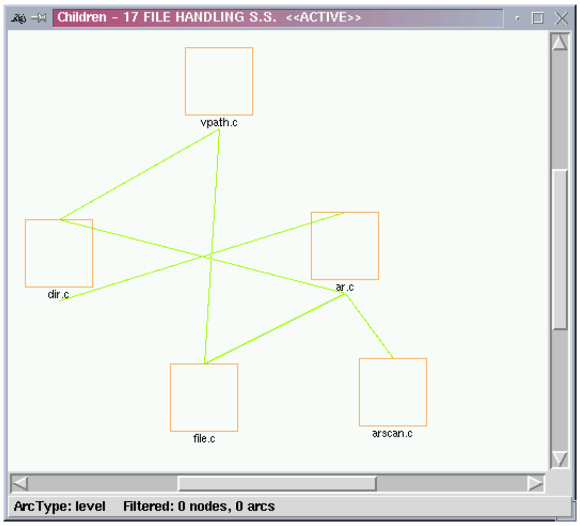

Figure 10: The File Handling subsystem.

1. ar.c

   This file contains functions for interfacing with `.ar` archives. It is referenced by many other files.

2. arscan.c

   This file contains a function for scanning an `.ar` archive. It is used exclusively by `ar.c`.

3. dir.c

   This file contains functions for creating, maintaining and querying hash tables of the files contained in directories. They are invoked from many other files.

4. file.c

   This file contains functions for creating, maintaining and querying hash tables of the target files for the make process. They are invoked from many other files.

5. vpath.c

   This file contains functions for supporting pattern matching of file search paths. It is called from `implicit.c` and `remake.c`.

These files were grouped together primarily because they all provide support for interacting with the file system, and thus they act as a library which provides a service to many of the other subsystems. The degree of interconnectedness as obtained from `cref` queries is not as high as some of the other subsystems, but given the obvious commonality of functionality provided by these files, it made sense to group them together into a single subsystem.

3.3.4 The Dependency Engine Subsystem

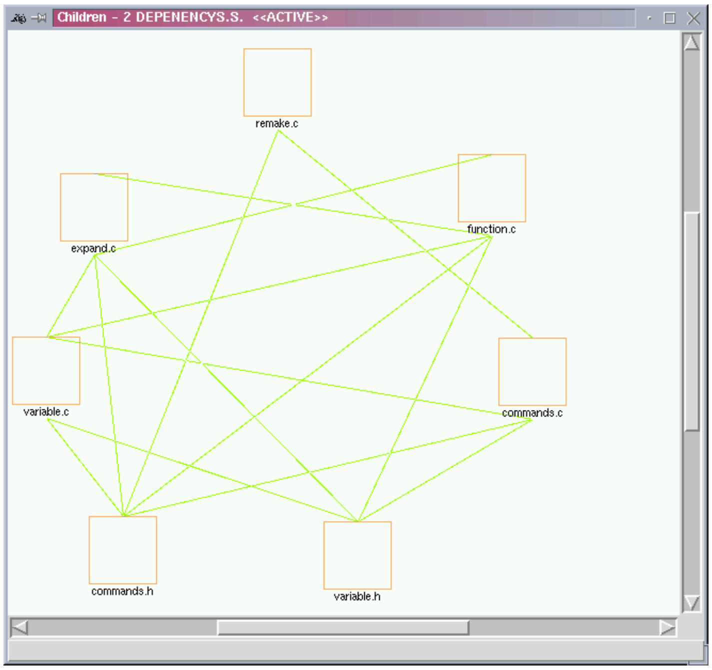

Figure 11: The Dependency Engine subsystem.

The Dependency Engine subsystem in figure 11 contains all of the files responsible for keeping track of dependencies and for actually executing the primary operations of the Make system. It contains the following files:

1. remake.c
   This file contains functions for parsing dependency rules and performing makes based on these rules. It also performs partial makes (or re-makes) based on what has already been made and on the parsed rules. It is called by `ar.c` and `main.c`.

2. expand.c
   This file contains functions for performing expansions of variables in a Makefile. It is used by many other files, but is included in this subsystem both due to similarity in functionality and due to the majority of the files which call it are also in this subsystem.

3. function.c
   This file contains functions for performing expansions of functions defined in a Makefile. It is used primarily by `expand.c`, although it is also referenced by `read.c` and `vpath.c`.

4. variable.[ch]
   This file contains functions for maintaining and manipulating Makefile variables. It is called by many other files, but was included in this subsystem due to similarity in functionality. The only non-library file it calls is `expand.c`.

5. commands.[ch]
   This file contains functions for processing makefile commands. It is called from outside this subsystem by `file.c`, `job.c`, `main.c` and `rule.c`. However, it is used primarily by `remake.c` which led to its inclusion in this subsystem.

The original basis for the formation of this subsystem was the arrangement suggested by Bunch, which was modified in places to take into account personal experience regarding how a system like Make would be structured. This process included looking at the basic functionality of the files as discussed above in order to ensure that files which appeared here belong together. This structure was then verified by using CIA `cref` queries together with the `wc` utility to determine how many references existed between the files inside and outside this subsystem. The queries provided results which supported the given subsystem decomposition in that most of the calls by files within this subsystem were to other files within this subsystem.

As can be seen in figure 11, this subsystem is nearly fully connected. These files are clearly very closely related.

3.3.5 The Parser Subsystem

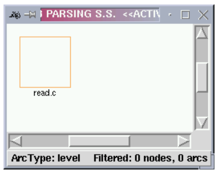

Figure 12: The Parser subsystem.

The parser subsystem is composed of the the single file `read.c` that contains the handling for reading and parsing makefiles which is called from the `main` function. This handling is quite self-contained and only makes use of library functions provided by other subsystems and returns the chain of makefile dependency structures to the main program.

The functionality provided by `read.c` is too specialized to combine into the `main` subsystem or the `General Services` subsystem and it makes use of functions from several other subsystems; thus, it the best choice is to place it in a subsystem by itself.

3.3.6 The Rule Engine Subsystem

The Rule Engine subsystem in figure 13 contains all the files which deal with Makefile rules. Specifically it includes the following files for parsing rules:

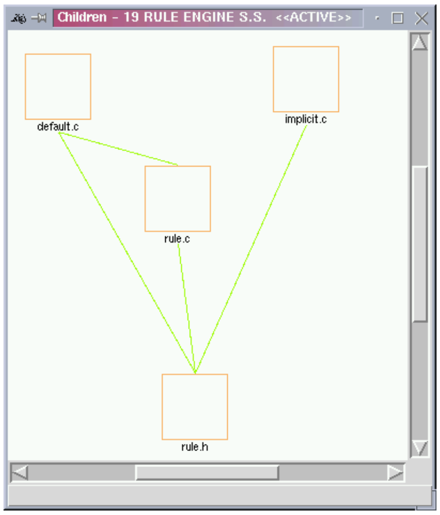

Figure 13: The Rule Engine subsystem.

1. rule.[ch]
   This file contains functions for parsing Makefile rules and for doing pattern matching of rules. It is called from outside this subsystem by a variety of other files, and from within this subsystem by only `default.c`.

2. default.c
   This file contains code to support the default rules common to all Makefiles. It is not referenced directly by the other files in this subsystem as it is called from `main.c` to initialize the internal rules. However, these rules are then used in `rule.c`.

3. implicit.c
   This file contains functions for handling implicit rules in Makefiles. It is called only from `remake.c`.

The primary motivation for grouping these files together was that they all provide functionality related to the handling of Makefile rules. Running various CIA queries provided support for this layout by showing that files which made use of the above files generally made use of all of them together. In addition, the files in this subsystem primarily make use of other files within this subsystem.

3.3.7 The General Services subsystem

The General Services subsystem in figure 14 contains the following files:

1. alloca.c
   This file provides a memory allocator which automatically reclaims memory when it is no longer in use.

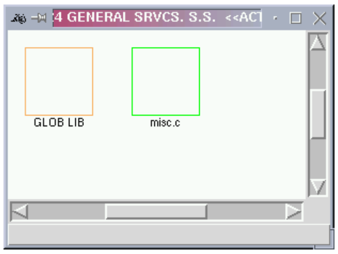

Figure 14: The General Services subsystem.

2. glob/*
   The files in this directory provide handling for wildcards.

3. misc.c
   This file provides a variety of general-purpose functions.

All of the files placed in this subsystem provide general-purpose functions which are used throughout the rest of the Make system.

3.3.8 The Job Control subsystem

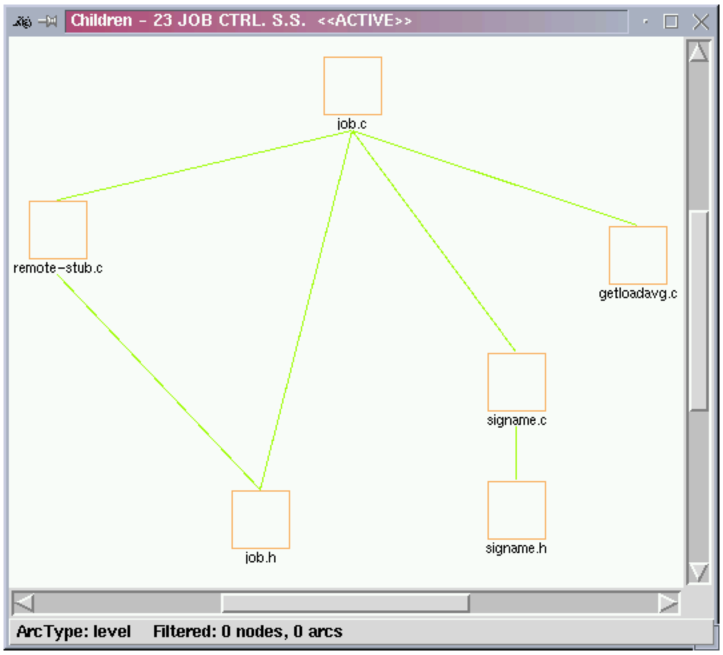

Figure 15: The Job Control subsystem.

The Job Control subsystem in figure 15 is composed of the following files:

1. getloadavg.c This file provides a function to return the current system load. It is invoked purely by `job.c`.

2. job.[ch] This file contains functions for creating and handling children processes. It is called from `commands.c`, `function.c` and `remake.c` which are all part of the dependency engine subsystem. They are also invoked from `main.c`.

3. remote-stub.c This file contains functions for supporting remote job execution and is tightly coupled to `job.c`.

4. signame.[ch] This file provides a function to convert signal names to signal numbers to handle inconsistencies between various Unix systems at run-time. It is invoked purely by `job.c`.

`job.c` provides the main job control functionality, and thus it makes sense to place it in a separate subsystem. Since `job.c` is the only file which calls `getloadavg.c`, `signame.c` and the only file which calls `remote-stubs.c` other than `main.c`, and those files are all job-control related, they have been placed in the same subsystem with `job.c`.

3.3.9 The Includes subsystem

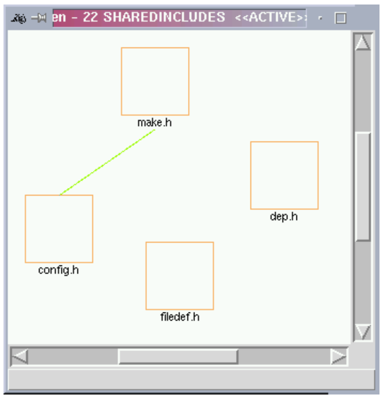

Figure 16: The Includes subsystem.

The Includes subsystem in figure 16 contains general-purpose include files which are used throughout the Make system.

4 Architectural Comparisons

This paper was begun by creating a conceptual architecture of the GNU Make system shown in Figure 3. This architecture was based on an examination of the source code, Bunch and our understanding of the Make system. The CIA and Rigi tool sets were then used to extract and visualize a concrete architecture. It is useful to examine the differences between these two architectures to identify where they differ. Figure 17 shows figure 3 with red lines indicating relationships which existed but were unexpected. All of the expected relationships existed.

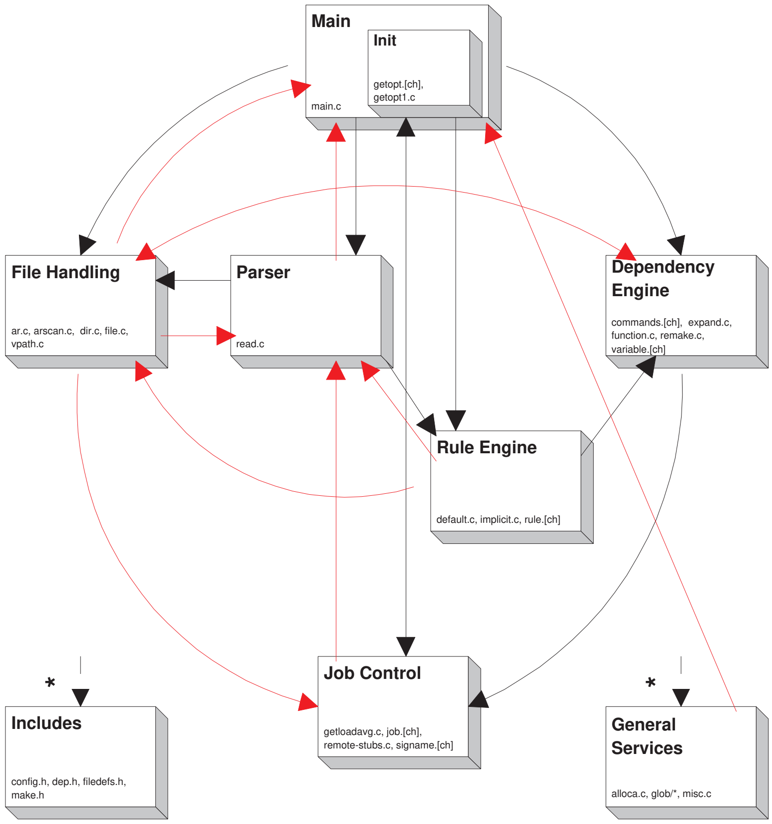

Figure 17: Comparison between conceptual and concrete architectures

.

4.1 Main Subsystem

It was very surprising to see that any subsystems other than Job Control made use of the Main subsystem. However, looking at figure 6, we can clearly see that many subsystems do in fact have relationships to the Main subsystem. This doesn't make sense at first glance since it was expected that the Main subsystem "starts" everything. Who calls Main?

By running the query `cref - - - - file1!=main.c file2=main.c`, all of the references to `main.c` from outside of it are obtained. Typical output is:

```
name1            file1            kind1 name2            file2            kind2
================ ================ ===== ================ ================ =====
set-default-suff default.c        funct no-builtin-rules main.c           varia
...
```

Here we see that a function in `default.c` is accessing a variable defined in `main.c`. This variable is one of many flags which are set in the Main subsystem. This is typical of most of the relationships coming into the Main subsystem. The rest are function calls related to initialization or error reporting, with the exception of Job Control.

4.2 File Handling Subsystem

There were two unanticipated relationships leading into the File Handling subsystem: one from the Rule Engine and one from the Dependency Engine. In both cases, those subsystems need to look up information on a file at one point or another and thus need access to the filesystem.

4.3 Parser Subsystem

There were several unexpected relationships leading into the Parser subsystem. Upon closer examination of read.c, it was noticed that that file provides some general purpose functions in which are used by other subsystems. Ideally, these library functions would be placed in a separate file so that the architecture could be clarified.

4.4 Job Control Subsystem

The only unexpected relationship leading into the Job Control subsystem was from the File Handling subsystem. This relationship turned out to be simply the fact that `file.c` includes `job.h`, even though it appears that `file.c` makes no use of anything in that header file. Thus, that include relationship can likely be removed from `file.c`.

4.5 Dependency Engine Subsystem

It was surprising to see that the File Handling subsystem has a relationship into the Dependency Engine subsystem. Using the Rigiedit navigator, information about this relationship was obtained and is shown in figure 18.

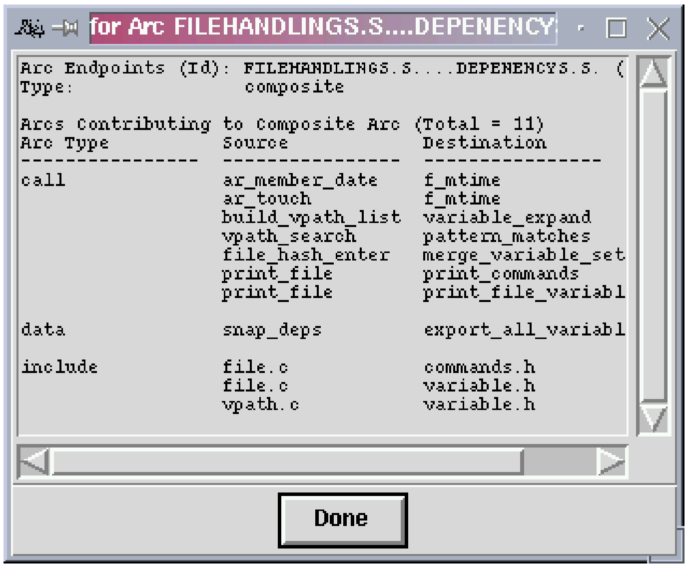

Figure 18: A Screen shot from Rigi showing information about a link.

This information states that the relationship link is an include link and that `file.c` includes `commands.h` and `variable.h`, and that `vpath.c` also includes `variable.h`. This can easily be confirmed with CIA as shown below.

```
cref - - - - file1=file.c file2=commands.h
name1            file1            kind1 name2            file2            kind2
================ ================ ===== ================ ================ =====
snap-deps        file.c           funct COMMANDS_NOERROR commands.h       macro
file.c           file.c           file  commands.h       commands.h       file
snap-deps        file.c           funct COMMANDS_SILENT  commands.h       macro
```

Here it can be seen that a function in `file.c` is using two macros defined in `commands.h`.

The same can be done for `variable.h`.

Finally, CIA can be used to acquire more information about the relationship between `vpath.c` and `variable.h`, where again we see the relationship is an include relationship.

```
cref - - - - file1=file.c file2=variable.h
name1            file1            kind1 name2            file2            kind2
================ ================ ===== ================ ================ =====
vpath.c          vpath.c          file  variable.h       variable.h       file
```

5 The Holy Grail of Software Architecture Extraction

The authors come to this project with varying experience in the area of software architecture extraction. One has extracted two large systems in the past (*GNU Gawk* and *The Linux Kernel*) and the other has been saved from such torture. Despite the differing background we agree on the following.

An ideal process for extracting a software architecture requires that the following, or similar, steps need to be followed:

1. Locate and interrogate the Software Architect of your system. If you can do this, you may be no further forward, since chances are his documentation is 2 years out of date anyway. Therefore it is probably best to start at step 2.

2. Become familiar with the system; this means reading the source code and whatever documentation you are lucky enough to find. The goal here is to gain a high level understanding about the relationships between entities in the system. The kinds of relationships you look at (calls, variable use, type use etc.) is up to you and depends on what you want to learn. You won't fully understand any reasonably sized system, so don't worry about learning everything.

3. Based on a tacit understanding of the system, form a high level conceptual architecture of how you view the system. This means grouping files into subsystems, and maybe subsystems into super subsystems. It is very important because it will help you to recognize and correct mistakes made by the tools you use.

4. Once you have a conceptual architecture to protect you from errors, you are ready to make use of the the ideal tools discussed below to build a concrete architecture. This will surely highlight errors in your conceptual view, but this doesn't make the conceptual view useless. Chances are, the conceptual view will be more useful as a day-to-day reference than the overly detailed concrete architecture - presuming you fix any glaring faults.

In order to extract the concrete architecture from the source code, you need excellent tool support. Without this, you will be forced to spend hours or even days writing scripts with `gawk, sed, wc, perl`, and other write-only Unix scripting languages. These scripts will be undocumented and impossible to use again. The characteristics of an ideal toolset are enumerated below.

1. **The Parser**.
   You need a parser to extract the facts from your source code. It is best if this parser doesn't make mistakes or introduce artifacts which aren't present in the source code (as Rigiparse has been known to do). When considering a parser, try and find one which doesn't rely too much on the ability of the code to actually be compiled on the platform it is being studied. For example, when looking at the *Linux Kernel*, the parsers from Rigi and PBS refused to work properly since the code we had was for an Intel architecture and the working environment was a SPARC Solaris. The CIA toolset seems to be very good at creating a database of facts. We didn't find that it introduced any errors.

2. **Database**.
   The parser should output the facts in a format which can be read by other tools. A database would be an ideal solution since it could be queried. CIA allowed us to use queries which would output text which could be further parsed and converted to a format viewable by our viewer, Rigiedit. A database which supports queries from both inside and outside a visualization tool would be ideal.

3. **The Buncher**.
   A tool like Bunch which could suggest some initial clustering based on internal intuition and user preferences is probably a good idea. However, a good tool will not be *too* configurable since this would simply replace the manual process of clustering by trying out all the combinations of preferences. Ideally this tool would work with the layout portion of the visualization tool and provide a first cut clustering which the user could move around.

4. **Visualization Tool**.
   Once we have the facts extracted, we want to be able to see them. For this we need a visualization tool. The tool not only needs to display the architecture, but it should also allow

the user to manipulate it. For example, *Dot* which is used with CIA does not fare well here. While it produces nice pictures, we can't touch them; that is, we can't move things around and navigate through them. The process of navigation and *feeling* the architecture is crucial since it supports understanding.

The tool must support querying from within. Ideally, the same sort of queries which can be run on the database outside the visualization tool should be able to be run from within the tool, except the results of the queries should be displayed graphically. Rigiedit fares well here. Not only can we move things around within this tool, but we can also perform queries (e.g. select all nodes with file attribute=`main.c`) and have the results displayed graphically. Being able to dynamically query information about a relationship link is also important.

The tool should support different views of the architecture. This is important so that we can hide the parts in which we are not interested at a given time, and so that different people can view the same system from different perspectives. Each person sees only what they need to see and what they are interested in seeing.

The tool must create aesthetically pleasing graphs (pretty pictures). Rigiedit fails here for many reasons: lines cross (no inflexion points) and it is sometimes hard to tell where lines start and end (no arrows.)

The visualization tool should make use of the latest fancy graph layout algorithms so that at least some level of automatic layout would be supported.

Finally, it would be nice if the visualization tool would support the creation of new nodes not actually present in the system. This would be useful for a architect who wants to do "What if tests" to see what things might look like if changes are made.

References

[1] **The Bunch Automated Clustering Tool**, `http://www.mcs.drexel.edu/~ serg/bunch/index.html`.

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

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

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