Browsing and Searching Software Architectures

Susan Elliott Sim[†] **Charles L.A. Clarke**[*] **Richard C. Holt**[‡] **Anthony M. Cox**[‡]
[†]Computer Science University of Toronto simsuz@cs.utoronto.ca
[*]Electrical and Computer Engineering University of Toronto clclarke@eecg.utoronto.ca
[‡]Computer Science University of Waterloo {holt, amcox}@plg.uwaterloo.ca

Abstract

Software architecture visualization tools tend to support browsing, that is, exploration by following concepts. If architectural diagrams are to be used during daily software maintenance tasks, these tools also need to support specific fact-finding through searching. Searching is essential to program comprehension and hypothesis testing. Furthermore, searching allows users to reverse the abstractions in architectural diagrams and access facts in the underlying program code. In this paper, we consider the problem of searching and browsing software architectures using perspectives from information retrieval and program comprehension. After analyzing our own user studies and results from the literature, we propose a solution: the Searchable Bookshelf, an architecture visualization tool that supports both navigation styles. We also present a prototype of our tool which is an extension of an existing architecture visualization tool.

Keywords

software architecture, information retrieval, program comprehension, software visualization

1. Introduction

A software architecture diagram is a high-level view of a software system [10, 25]. Common ways of representing software architectures include box-and-line drawings, hierarchical trees, and nested boxes [11]. They are structural abstractions of the underlying software; they itentionally leave out details, so that selected concepts can be depicted more clearly. In other words, they are visualizations of a large, complex *information space*—the program code.

There are two *navigation styles* for investigating an information spaces: searching and browsing [18]. *Searching*, sometimes called analytical searching, is a planned activity with a specific goal, such as to find a particular fact. It is often associated with who, what, when, and where questions. Searches involve formulating queries or looking in indices. In contrast, *browsing* is an exploratory strategy, with no fixed endpoint, and is relatively unstructured. The knowledge seeker relies on serendipity to uncover relevant information. Browsing is associated with why and how questions and exploratory investigations, and involves actions such as flipping through the pages in a book, or following links through hypertext.

When browsing a software architecture diagram, such as one shown in Figure 1, the user explores a software system via a visualization. The visualization is generated by abstracting details from the source code to show a conceptual representation of the system. Since browsing is suitable for exploring new domains, such an interface is appropriate for users who are unfamiliar with the software system. Browsing can be used to investigate the hierarchical composition of the software system by moving from subsystem to subsystem. However, if a software maintainer wanted to learn about the source code and not just the architecture, she would need to access the facts that were used to construct the abstraction. In software visualization, this process is called *reverse abstraction*, moving from representations of concepts to the underlying facts [31]. For instance, to find the lines of code that is represented by a single edge, the software maintainer needs to reverse abstract the architecture diagram. It is easier to reverse abstract using searching. This navigation strategy is commonly used by programmers through text editors and utilities, such as `grep`. Unfortunately, tools that work with text do not carry over to diagrams.

In this paper, we examine the problem of browsing and searching software architecture diagrams and the underlying program code. We use results from our own user studies, and the literature from various fields to guide the development of a tool to solve this problem. We propose a solution, adding search capabilities to an existing software architecture visualization tool, and construct a prototype, the Searchable Bookshelf.

The majority of software architecture visualization tools only support browsing. Tools such as Rigi [20], PUNS [16], Dali [14], and Software Bookshelf [8] display software architectures and allow users to explore them, but have only primitive query mechanisms. Other tools allow the user to query and build views, such as CIA [4], the Extensible Dependency Tool Set (EDATS) [33], Inter-Module Code Analysis system (IMCA) [7], LSME [21],

and ManSART [35]. However, these tools operate only on architectural level facts, they do not use the architecture as a means for organizing or accessing the information space underlying the abstraction. Indeed, we use similar tools for creating the Software Landscape diagrams.

Approaching the problem from the opposite direction, there are tools that facilitate searching on source code, for instance `grep` and its variants [13, 34], scruple [22] and `tksee` [30], but they operate separately from any available architectural or conceptual information. Our goal here is to allow the end-user to reverse abstract the information represented in the Software Landscapes, using both visual and textual data found in the source code and the diagrams.

The Searchable Bookshelf is an extension of the Software Bookshelf as constructed by Holt et al [12]. Searches are specified using GCL, a query language from information retrieval, designed for use with structured and semi-structured texts such as source code [5, 6]. GCL is distinguished by its support for queries that reference both structure and content, and by its uniform handling of structured data, so that diagrams, source code, and documentation can be searched using the same interface.

In Sections 2-5, we further discuss the idea of software as an information space. We use results from information retrieval, program comprehension, and software visualization literature, as well as our own user studies to motivate the design of the Searchable Bookshelf. The tool itself, along with an extended example, is described in Section 6, and additional details about the development of the Searchable Bookshelf are given in Section 7.

2. Navigation Styles

Information spaces can take many forms. They may be physical, such as libraries and card catalogues, city streets, or a region of wilderness. In such environments, browsing is analogous to taking a walk or looking at a map, while searching is analogous to asking a person, consulting a card catalogue, or looking up a street index. Examples of electronic information spaces are databases, document repositories, or the World Wide Web. When these electronic information spaces were first developed, the only way to navigate them was by searching. Users had to formulate queries using a command language or fill-in forms. With the advent of hypertext and visual displays, browsing, or surfing, of electronic information spaces has become feasible.

Consider the example of a physical library. If a user has a specific book in mind and knew something about that book such as its title or author, she can use a search strategy. She can look up this information in a catalogue, find the book's location on the shelves, and obtain the book directly. On the other hand, if the user has just read an interesting book and wants to know more about the topic, she could use a browsing strategy. She can go to where the book was shelved and examine others nearby volumes. This example illustrates two points. First, both searching and browsing are necessary to using the library effectively and the specific strategy chosen depends on the task. Second, browsing assumes that the information has been organized so that related elements can be found together.

These two navigation styles are present in many electronic spaces, for instance, in the Yahoo index at http://www.yahoo.com. Yahoo is an index of web pages organized hierarchically by categories. On its main page, there is a search field and a number of top level categories which supports browsing. Users can search for categories and sites of interest by typing keywords into the search field.

A software system can also be considered an information space. Instead of books or web pages, there are subsystems, modules, files, functions, variables, and lines of program code. The most common method of navigating program source is searching, using utilities like `grep`, or features in a text editor. However, once an organization is imposed on the code, such as class hierarchy, call graphs, or a software architecture, it is possible to browse based on structure. A visualization of the organization provides additional support for browsing. The diagram can serve as a map of the categories and the relationships between them. Also, by depicting the organization visually, additional information can be conveyed. For example, the importance of a concept or attribute can be shown using colour or size.

In the next three sections, these two navigation styles will be discussed, first separately and then together, within the context of information seeking within a software system.

3. Browsing Software Architectures

There are a number of tools for browsing software architectures, such as Rigi, Dali, the Software Bookshelf, and the Portable Bookshelf (PBS). Although there are differences between the tools, they are conceptually similar in that they all display software systems as graphs, and these graphs can be explored to view the architecture of the software system. We will use PBS to illustrate browsing of architecture diagrams, since we have studied users' interactions with this tool and the Searchable Bookshelf is an extension of this tool. PBS is actually a set of tools for generating software architectures from program source. It uses a Java-capable web browser as user interface, so users can uniformly access program source, documentation, and architectural diagrams called Software Landscapes [24].

Figure 1 is a Software Landscape of the FS (File System) subsystem of the Linux™ operating system kernel. Although this diagram is shown grayscale, colours are used to distinguish between boxes and lines. Modules or files are represented as blue rectangles with a corner

folded down. Subsystems are drawn as gray rectangles with tabs, like a file folder. Green edges represent variable references and red edges represent function calls between rectangles. The graph is drawn using a nested box formalism, meaning that subsystems can contain other subsystems or modules. In the diagram, the FS subsystem contains four subsystems and a module. In order to see the internal organization of one of these smaller boxes, the user can click on it with a mouse. A new landscape is displayed showing the subsystems and modules contained in the originally selected box. If the selected box is a file, the lines of code in the file are displayed.

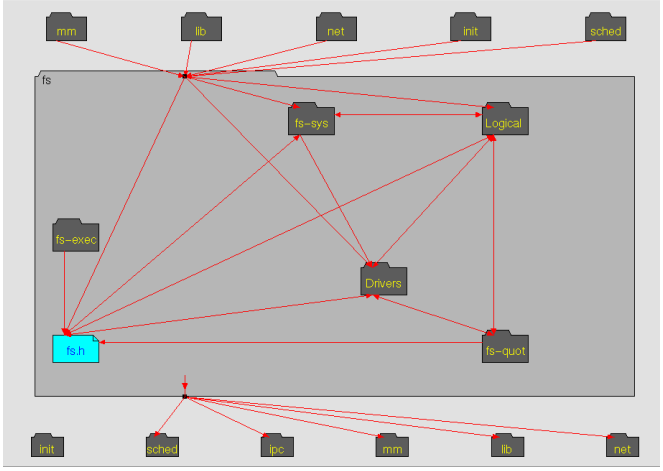

Figure 1: Software Landscape Diagram

The landscape also shows the clients and suppliers of the subsystem. The clients are the row of boxes shown at the top of the landscape; They use resources, such as variables and functions, provided by the central subsystem. Similarly, the suppliers are the row of boxes at the bottom of the landscape; They provide resources to the central subsystem. Clients and suppliers can be either subsystems or modules. In the diagram, the FS subsystem has 5 client subsystems and 6 supplier subsystems. By using this convention, we can show the central subsystem in context.

A Software Landscape is generated by a series of static analysis tools. A parser creates a factbase consisting of function calls and variable references. The hierarchical decomposition of the software is recovered by clustering files into subsystems using manual and automatic techniques. The resulting information is drawn and adjusted using a Java applet.

Landscapes and PBS are designed to be browsed. The basic mode of navigation is point-and-click, just as with other World Wide Web constructs. Searches are limited to a single Landscape at a time. It follows Shneiderman's mantra for visual information seeking: "Overview first, zoom and filter, then details on demand." [26], p. 523 The user is first presented with an overview of the software system and she can zoom in on subsystems of interest. Edges and nodes can be filtered out selectively and additional information is available through mouse clicks.

3.1 User studies

Portable Bookshelves have been constructed for a number of large systems at IBM Canada Ltd. Typically, these systems have several hundred thousand lines of code and are maintained by 10-20 people. The PBS usually contains the program code, some documentation, and Software Landscapes. We have studied PBS users both formally and informally over the past 3 years [3, 29, 32]. PBS was observed being used in a number of tasks, but there are four tasks in which it worked particularly well: familiarizing newcomers to the maintenance team with the documented software system, providing experienced team members with an overview of an unfamiliar subsystem, validating relations between subsystems, and verifying reengineering decisions.

Newcomers, or software immigrants, found Software Landscapes particularly useful during their first two weeks on the team [27]. The pictures gave them a good overview of the system and a sense of the relations between the parts. Both of these advantages are typical of tools with a browsing interface [18]. Project veterans would also use the landscapes before modifying an unfamiliar subsystem. They would use the diagrams to help them relate their knowledge of other parts of software system to the subsystem of interest. However, beyond providing an initial overview, landscapes were not used during actual maintenance tasks, such as defect repairs or feature addition, because there was a mismatch between the information provided by the landscapes and the information required by maintainers to perform these tasks. Landscapes provide abstract, high-level information, and this information tends to be conceptually distant from the concrete, low-level information provided by source code [27]. This gap is noteworthy because it underlines the fact that a maintainer's essential task is to modify the source code. Therefore, in order for landscapes to help with maintenance tasks, the users must be able to relate the concepts that they depict to source code. In other words, they need to be able to reverse abstract the diagrams.

Senior developers have been observed using landscapes to check for anomalous edges. These edges denote a relationship between two modules or subsystems, where there should not be one. These anomalies, which are found by browsing, serve as the basis for searches to identify the offending lines of code [3]. During reengineering, for example, re-implementing subsystems in a modern object-oriented programming language, landscapes are consulted to verify that they do not contradict any decisions made. For instance, a maintainer uses them to check that there are no edges to the re-designed subsystem that are not accounted for [27].

In using Landscapes, there were a number of situations in which browsing was not sufficient for the task at hand. Often when viewing a Software Landscape, a software maintainer wanted to relate the boxes and arrows to the source code they represented. In the case of boxes, this question could be answered by following branches of the hierarchy to its leaves, which were files. In the case of arrows, this question can only be answered with difficulty using search tools, such as `grep`, that are outside of PBS. In another situation, a software maintainer isolated a problem to a specific file and wanted to know what subsystem that file belonged to. This question could not be easily answered by browsing the landscapes. These examples illustrate situations requiring an architecture-guided search facility.

4. Searching Software Architectures

We now turn to the second navigational style, searching. It is a style that we commonly use with source code, databases, and indices. Searching is a powerful information seeking strategy. Its popularity is evident in the number of search tools and search specification, or query, languages available. Many studies have been performed in the field of information retrieval to characterize strategies used to query databases and textbases [18]. Searching is flexible and can be used to gather varying amounts of information from one or many sources. The main drawback of searching is that it is difficult to obtain an overview of the information space. Also, this mode of interaction is better suited to locating specific facts rather than gleaning concepts.

Software architecture visualization tools have limited search facilities. Since we could not study software developers and maintainers as they searched software architectures, we studied how they search source code. We conducted a survey using a questionnaire on a web page to collect information on the tools used to search, the strengths and weaknesses of these tools, and anecdotes of searches [28]. The results of the survey most germane to this discussion are the search targets and the tools used for searching.

The most common search targets were: function definitions (or bodies), all uses of a function, all uses of a variable, and variable definitions. These search targets were identified repeatedly by respondents, but most did not use specialized tools such as tagging utilities or cross reference generators that would simplify these searches. The three most common tools used for searching were the text editor, `grep` (a UNIX regular expression matching utility) and `find` (also a UNIX utility, alternatively File Find under Microsoft Windows). Although respondents were looking for semantically significant elements in the source code, they specified their searches using only strings or regular expressions.

A search facility for a software architecture must be able to specify searches for meaningful elements in the source code such as functions and variables. Such searches are difficult using only strings or regular expressions to specify the targets. Consider the problem of searching for references to the variable "i" in a set of C source files. The `grep` command "`grep i *.c`" prints all the lines on which the character "i" appears. This command may produce a considerable volume of output, most of which will not contain a reference to the variable. It is also difficult to express a query that may match across multiple lines, such as "find all functions that contain a call to both fork() and exec()". A more powerful query language, or a form-based interface with mouse actions will be necessary to support semantic searches.

5. Combining Browsing and Searching

In the previous two subsections, we discussed how browsing and searching are both necessary to navigate an information space. The navigation style chosen depends on the task at hand. Users browse to explore the information and to understand concepts. They search to find particular facts and to answer specific questions. Furthermore, users often switch between these strategies to accomplish a single task. For example, a user may perform a search to find a starting point for browsing. Or during browsing, a user may find an appropriate keyword to use for a search.

These styles of navigation support different program comprehension strategies. Currently, the dominant model of program comprehension is the integrated model which states that programmers use top-down and bottom-up comprehension strategies as dictated by the available information and frequently switch between them[15, 19].

A programmer uses a bottom-up strategy by reading the source code and building abstract concepts by *chunking* together low-level information [23]. A software maintainer using bottom-up program comprehension would require searching. Bottom-up comprehension relies on finding facts in the code and building concepts with them. Both strategies move from lines of source code to abstractions.

A programmer uses a top-down strategy by employing domain knowledge to build a set of expectations about the program. These expectations are mapped onto features or *beacons* in the source code [2, 17]. A software maintainer using a top-down strategy would require browsing. Top-down moves from concepts to specific code elements. Both strategies involve the user moving from high-level concepts to program source.

Just as navigating an information space requires both browsing and searching, the integrated program comprehension model states that programmers use both top-down and bottom-up strategies. Since software maintainers use both bottom-up and top-down strategies to comprehend code, there should be a unified interface for searching and browsing the software architecture. Because the programmer frequently switches between

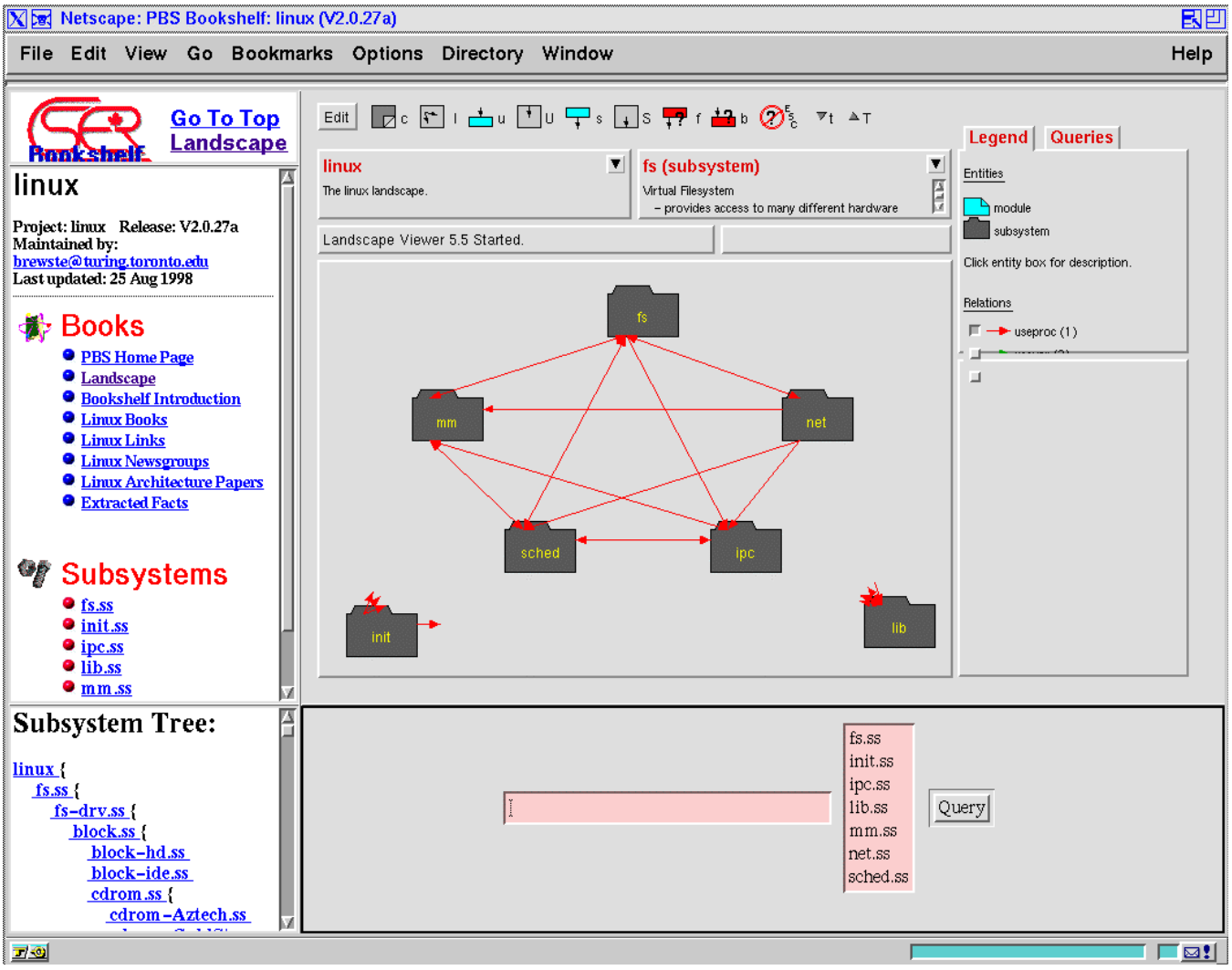

Figure 2: The Searchable Bookshelf

code comprehension strategies, she should not have to change tools or views to switch navigation modes.

In the next section, we describe the Searchable Bookshelf and how it supports both browsing and searching. In the following section, we present the technical details of the design and construction of our tool.

6. The Searchable Bookshelf

The interface to the Searchable Bookshelf of the Linux kernel is shown in Figure 2. The column along the left side contains the table of contents of the Bookshelf and indicates what information is available. The landscape diagram is found the large window in the right. The interface to the search tool is the HTML form found in the small window on the bottom. It consists of a text box to enter the query, a scrolling selection box from which to specify search targets, and a button to activate the search.

The contents of the list of search targets is generated from the same information as the currently displayed landscape. It can contain subsystems, modules, or files. In Figure 2, the seven choices in the target list correspond to the seven subsystems in the Software Landscape. These choices are used to restrict the search to specific parts of the software system. If no target is specified, the entire system is searched. Queries made in the form are passed to the `grug` tool using a Perl script via the CGI. Information returned by the query is also displayed within the frame.

The `grug` tool is an extension of `grep`, and combines regular expression search with semantic and structural search capabilities taken from the GCL query language. In addition to regular expression and literal string matching, `grug` can search for more meaningful information such as declaration, definition, and use of variables, functions, modules and subsystems.

Recall our discussion earlier of searching for all references to the variable "i". This search can be performed in `grug` with the query:

```
VARREF > "i"
```

The ">" symbol should be read as "containing". Expressions enclosed in double quotes (") match on complete tokens such as variable or function names. Using grug, it is possible to distinguish between variable declarations and references. Declarations of the variable "i" may be identified using the `grug` query:

```
VARDEF > "i"
```

Both queries return the lines where the variable "i" appears. Character positions may be used to identify other structural elements, such as searches for functions containing the reference to the variable i.

```
FUNDEF > (VARREF > "i")
```

More complex combinations are also possible, for instance:

```
FUNDEF > ((FUNREF > "fork") AND
        (FUNREF > "exec"))
```

which finds all functions that contain a call to both fork() and exec().

By combining `grug` with Software Landscapes, we create a software architecture visualization tool that supports both navigation styles, browsing and searching. Transitions between different styles can occur with each new mouse click issued by the user.

6.1 Extended Example

In this subsection, we demonstrate how the Searchable Bookshelf is used in a realistic task to illustrate a selection of its functionality. The example shows both the browsing and searching capabilities of the Searchable Bookshelf and how they can be used together.

In the course of building a PBS for the Linux kernel, we observed that there was some architectural erosion in the file system subsystem. The device drivers use a Facade design pattern, meaning that all drivers are meant to be accessed through single interface.[9] This interface allows higher level file system functionality, such as logical and virtual file systems, to be constructed independently of the implementation details of any particular device. In other words, the same file structure can then be used across different storage devices, such as hard disk drives and floppy disk drives. However, the ISO File System (ISOFS) subsystem of the logical file system that does not follow this convention. The ISOFS subsystem implements the ISO 9660 file system for CD-ROM files and accesses resources from the CD-ROM device driver directly, instead of using the Facade interface [1].

Lucy the programmer has been given the task of carrying out this architectural repair. The purpose of an architectural repair is to restore the original design of the system. In this case, the task is to repair the Facade design pattern, so that the ISOFS does not access the CD-ROM driver directly. Lucy has previous experience with a UNIX operating system, is familiar with Linux, and has used the Searchable Bookshelf extensively. Lucy begins by browsing the landscapes to find the ISOFS subsystem. In the diagram of the ISOFS subsystem, she sees that the devices subsystem is indeed a supplier, as it is located beneath the central system. At this point, she can perform a visual query, that is, a manipulation of a visual elements to see relationships more clearly. Lucy clicks on the devices subsystem to select it, then she clicks on the backtrace button to find out what modules in the ISOFS subsystem use the CDROM subsystem. The result is a single green edge from the `inode.c` module to the Driver subsystem. Lucy now knows that she needs to find a variable reference between two files, so she can begin searching.

She needs to find a variable that is declared in the driver subsystem and is used in the `inode.c` module, so she types the following query:

```
(VARNAM < (VARDEF < SUBSYS ("driver")))
 > (def @ (VARREF < (MODULE ("inode.c")
        < SUBSYS ("isofs"))))
```

In English the query would be “find all the variable names that can be found in a variable definition in the driver subsystem that is also referenced by the `inode.c` module in the isofs subsystem.”

In the preceding query, the majority of terms, VARNAM, VARDEF, SUBSYS, VARREF, and MODULE, are GCL macros to access indexed information in the factbase. Two of the macros, SUBSYS and MODULE, take parameters. The “>” symbol should be read as “contained in” and the “<” symbol as “containing.” The “`def @`” expression is used follow cross references between the variables and their definitions. The factbase, GCL query language, and macros are explained in greater detail in Section 7.

Lucy wants to search the entire system, so she does not select any targets before clicking the query button. This query does not return any matches, so Lucy thinks for a moment before realizing that the edge could also represent use of a variable type or macro. She proceeds to make some more queries based on this idea. First, she searches for the type usage:

```
(TYPNAM < (TYPDEF < SUBSYS
("driver"))) > (def @ (TYPREF < (MODULE
   ("inode.c") < SUBSYS ("isofs"))))
```

An English translation of the query is “find the names of all the types that are defined in the driver subsystem and is also used in a variable definition in the `inode.c` module of the isofs subsystem.”

This search returns the type name “cdrom_multisession” which is found in the file cdrom.h. Although she has found a match, Lucy will also search for any uses of macros as well. This search is similar in to the ones above and yields the match “CDROM_LBA”, also from the file `cdrom.h`.

Although the architecture can be repaired in a number of ways, Lucy accomplish the task by creating a new header file for the ISOFS subsystem. Armed with her search results, Lucy copies these definitions into a new header file that is part of the in the ISOFS subsystem, and

modifies `inode.c` to use this new header. To keep the code maintainable, she puts a comment into both the old and the new header files indicating that portions of the code have been copied, and that changes should be propagated.

7. Adding Search to an Architecture Browsing tool

In this section, we present some of the technical details of the search mechanism in the Searchable Bookshelf. We begin by providing a rationale for selecting GCL as the query language and an overview of the language itself. This section also includes a description of the tool to generate the factbase for the Searchable Bookshelf and the syntax of GCL.

The two most common search specifications in text editors and simple search tools such as `grep`, are literal string matching and regular expression matching. Both regular expression and string matching operate by comparing a specified target to a file or set of files and returning matching records. Often, a record is defined as a line of text, but it can also be defined as a word, a data record, or file.

Although these two search specification mechanisms are commonly used, we found that they had two main shortcomings when used to search software architectures. The first drawback is they cannot be used to search for semantic elements. These mechanisms are agnostic about the structure of text being searched. Consequently, there is no syntax to restrict searches to a particular structural or syntactic element. The second drawback of these two mechanisms is that the matches they return come in fixed units or records. Depending on the task, the size of the match can vary. Sometimes it is the name of a function, at other times, it is statement block, and at still others, it can be an entire function definition.

GCL overcomes the two drawbacks given above. Its syntax can be used to search both structure and text. Since GCL is programming language- and schema-independent, it can be used with different software systems and the various documents associated with them, such as source code, documentation, and landscapes.

This query language was designed for use with structured and semi-structured text, such as web data, bibliographies, and email [5, 6]. The query language requires character-level markup of the text to indicate the boundaries of structural elements. For example, HTML tags can serve as the markup for a web document. Alternatively, a document can be marked up implicitly by building an index. We use this last approach with the Searchable Bookshelf, since PBS also has an underlying factbase. In this tool a factbase, or index, is constructed using an extended version of the GNU C Compiler. Some of the facts that are extracted are definitions and uses of macros, variables, functions Included with each fact, are the file positions at which each element occurs.

In section 7.1, the method we use to generate the factbase is described. In section 7.2, we describe the syntax of the GCL query language itself.

7.1 Factbase Generation

We extended the GNU C compiler so that a set of factbase files (one for each source file) is generated during compilation if the "-FB" flag is used. This usage is analogous to the “-d” flag which is used to generate output for use with a debugger. While making this change, we found that it was necessary to modify the C pre-processor to emit both a character map and a set of facts describing its activities, such as macro expansion, file inclusion, locations of comments. The character map describes the actual source of every character in the pre-processed file so that facts generated in the compiler proper can be mapped back to their source files. Facts are generated only for the compiled portions of the code.

Often, more than one fact is collected for each statement of interest and in addition to the facts themselves, the factbase also contains information about their locations. Consider the following variable declaration:

```
0 1 2 3 4 5 6 7 8 9
i n t   c o u n t ;
```

The factbase would contain facts about the variable definition, type of the variable, and the name of the variable. Each of the facts would have a start and end locations in terms of file positions, and these would be denoted in the factbase using SGML-style tags. The start of the variable definitions would be denoted with the tag <vardef> and associated with file position 0. Correspondingly, the end of the variable definition would be denoted as </vardef> and associated with the file position 9. Thus, the complete list of facts for the above declaration would be:

Fact	File Position	Fact	File Position
<vardef>	0	</vardef>	9
<vartyp>	0	</vartyp>	2
<varnam>	4	</varnam>	8

More facts would be collected about other statements, such as a function definition, which would includes facts on function name, return type, arguments, types of the arguments. The parser generates over 60 different types of facts, about macros, macro calls, function declarations, definitions, and calls, variable definitions, declarations, and calls, type definitions and uses, and blocks.

Locations are recorded as file positions because `grug` performs matches at the character level. The file positions can be used to retrieve the matches from a file quickly and easily. These locations are used to enforce a rule for matches. This rule will be discussed in the next subsection.

7.2 The `grug` Tool

There are four aspects of the `grug` tool that need to be understood: basic queries, the matching rules for

solutions, the operators to form complex queries, and the macros. These concepts will be explained with the aid of the code sample shown in Figures 3 and 4. The program prints command line arguments to standard output. Figure 3 shows the original program code, while Figure 4 shows the same program code with the file positions labeled.

A basic `grug` query is a literal string or regular expression to be matched. Searches can be performed on the source code or on the tags in the factbase. These searches function in the same way as commonly found tools and utilities, but it should be noted that only the match itself is returned and rather than the line, or record, containing the match. In technical terms, the *solution* to a query is a set of *extents* or ranges in the text of the leftmost shortest matches. For example, the query `"main"` on the code sample has the solution "main" at file positions 5-8. Double quoted strings match complete tokens. Regular expressions need to be enclosed in back quotation marks and match anywhere. The query `` `arg.*` `` would return matches to argc from file positions 14-17, and 50-63, and matches to argv from file positions 27-30, 93-96, and 123-126.

```
 1  void
 2  main(int argc, char * argv[])
 3  {
 4    int i;
 5
 6    for(i = 1; i < argc – 1; i++)
 7    {
 8      printf("%s ", argv[i]);
 9    }
10
11    printf("%s\n", argv[i]);
12  }
```

Figure 3: Program Source for echo.c

	0	1	2	3	4	5	6	7	8	9
0	v	o	i	d	\n	m	a	i	n	(
1	i	n	t		a	r	g	c	,	
2	c	h	a	r		*		a	r	g
3	v	[	]	)	\n	{	\t	i	n	t
4		i	;	\n	\n	\t	f	o	r	(
5	i		=		1	;	i		<	
6	a	r	g	c		-		1	;	i
7	+	+	)	\n	\t	{	\n	\t	\t	p
8	r	i	n	t	f	(	"	%	s	
9	"	,		a	r	g	v	[	i	]
10	)	;	\n	\t	}	\n	\n	\t	p	r
11	i	n	t	f	(	"	%	s	\	n
12	"	,		a	r	g	v	[	i	]
13	)	;	\n	}	\n	^Z				

Figure 4: echo.c with File Positions Labeled

An important rule about solutions is that extents may overlap, but they cannot nest. In other words, a solution cannot itself contain an entire solution. However, a solution can begin within the extent of another solution, so long as it ends outside of that extent. This rule is the reason that both start and end positions must be stored in the factbase. This distinction becomes important with complex queries that return multiple solutions.

The next significant feature of `grug` are the operators which are used to combine queries we can obtain solutions other than simple strings. These longer solutions must begin and end with solutions to basic queries, i.e. literal strings and regular expressions. The operators fall into three categories: ordering, combination, and containment.

There is one ordering operating, "`...`". It can be used to search for an extent that begins and ends with a query. For instance, the query `` `printf(`...`);` `` would return every call to printf, that is, file positions 79-102 and 108-124, in the example. The search

```
"<vardef>" ... "</vardef>"
```

would match every variable definition in the factbase. There are two combination operators, `AND` and `OR`. Again, solutions must begin and end with a match to a query. For example, the query `"argv" AND "argc"` match file positions 14-30, and return "`argc, char * argv`". Finally, there are four containment operators: containing, contained in, not containing, and not contained in, represented by `>`, `<`, `/>`, and `/<` respectively. The query `` (`printf(`...`);\n`) > '\\n` `` would return all calls to printf containing "\n". In the example, only the second call to printf would be returned.

Macros in `grug` allow users to chunk frequently used, but complex, searches to simplify their use. The utility can retrieve macros from a file. Different sets of macros could be defined for different purposes, such as email, bibliographies, and programming languages. For example, `` LINE = `^` ... `$` `` defines a macro called LINE that returns each line in the file. This macro can be used for grep-like searches, such as

```
LINE < `arg.*`
```

which matches every line containing the specified regular expression. Macros are also used to simplify access to information in the factbase. Recall the variable definition from the previous subsection. There were three different facts that were denoted by pairs of tags. Using only the tags, a search for the declaration of the variable "count" would be written as:

```
      ("<vardef>"..."</vardef>") >
(("<varnam>"..."</varnam>") > "count")
```

But if we introduce the following macros:

```
VARDEF = "<vardef>" ... "</vardef>"
VARNAM = "<varnam>" ... "</varnam>"
```

the query would be simplified to:

```
VARDEF > (VARNAM > "count")
```

As illustrated in the extended example, pre-defined macros were the primary way to access structural and semantic information. For each type of fact in the

factbase, a corresponding macro is defined. The same macro names could be used in different programming languages, so queries for the same facts would remain consistent, thus making `grug` programming language independent. The `grug` tool can also be used to search documentation in addition to source code and landscapes. We plan to use this ability to extend the information space accessed by future versions of the Searchable Bookshelf.

8. Summary and Future Work

We developed the Searchable Bookshelf by taking a number of concepts from information retrieval and applying them to software. The first notion is the information space. The written data of a software system, source code, documentation, Software Landscapes, etc., can be thought of as an information space. The second notion is that an information space can only be fully utilized when both navigation styles, browsing and searching, are available. Browsing is used to explore the information space and to understand high-level concepts. Searching is used to find specific facts and to identify low-level details. Browsing is most effective when some conceptual organization has been imposed on the data, which allows the users to follow relationships between points in the information space. Software architecture diagrams, such as Software Landscape, impose such an organization, and browsing is a commonly available navigation style in software architecture visualization tools. In contrast, searching is a navigation style that has long been available on program code, but is largely absent in architecture visualization tools. The Searchable Bookshelf is an architecture visualization tool that supports both navigation styles.

The Searchable Bookshelf was constructed by adding a search tool to PBS, an existing architecture visualization tool that supported browsing, but had limited search capabilities. Searches are specified using the GCL query language, also taken from information retrieval. The main feature of this language is it can be used to search both structure and text. There are two key reasons for adding searching to PBS. The first reason is that the architecture diagrams become more germane to daily maintenance tasks, such as defect repairs and feature additions, because users can reverse the abstractions in the diagrams. By reverse abstracting, software maintainers uncover the facts that were clustered together to make a concept, and it is these facts that they require to modify the source code. In other words, browsing is needed to understand the concepts, but searching is needed to uncover the facts underlying the concepts. The second reason is the two navigation styles, browsing and searching, parallel the two primary program comprehension strategies, top-down and bottom-up, respectively. The dominant model of program comprehension, the integrated model, states that programmers use the strategy that fits the available information and switch freely between them. Hence, the navigation styles supported by the Searchable Bookshelf are consistent with program comprehension strategies.

Our first prototype of the Searchable Bookshelf was completed in the Spring of 1998.[22] Experience with the prototype led to several design changes (notably the addressing of elements at the character level) that are included in the description above and are being incorporated into the current version.

The Searchable Bookshelf and its underlying technology can be improved in a number of ways. First and foremost, the usability of the Searchable Bookshelf has not been validated. Although, we relied on our own user studies and similar studies in the literature to design the Searchable Bookshelf, we have not tested our tool with professional software maintainers. The second issue is that there is no mechanism in our tool to support learning of the GCL query language. Currently, there is a steep learning curve between simple queries for strings and regular expressions and complex ones, like those in the extended example. Third, we are working to improve the query language to deal with source code features, such as blocks and recursive patterns.

Acknowledgments

This work is supported by NSERC and sponsored by the Consortium for Software Engineering Research (CSER). Thanks to Ivan Bowman for his advice on the Linux kernel.

9. References

[1] I. T. Bowman, R. C. Holt, and N. Brewster, “Linux as a Case Study: Its Extracted Software Architecture,” presented at 21st International Conference on Software Engineering, Los Angeles, CA, 1999.

[2] R. Brooks, “Towards a theory of the comprehension of computer programs,” *International Journal of Man-Machine Studies*, vol. 18, pp. 543-554, 1983.

[3] I. Carmichael, V. Tzerpos, and R. C. Holt, “Design Maintenance: Unexpected Architectural Interactions,” presented at International Conference on Software Maintenance, Nice, France, 1995.

[4] Y.-F. Chen, M. Y. Nishimoto, and C. V. Ramamoorthy, “The C Information Abstraction System,” *IEEE Transactions on Software Engineering*, vol. 16, pp. 325-334, 1990.

[5] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski, “An Algebra for Structured Text Search and a Framework for its Implementation,” *The Computer Journal*, vol. 38, pp. 43-56, 1995.

[6] C. L. A. Clarke, G. V. Cormack, and F. J. Burkowski, “Schema-Independent Retrieval from Heterogeneous Structured Text,” presented at Fourth Annual Symposium on Document Analysis and Retrieval, Las Vegas, NV, 1995.

[7] S. W. Dietrich and F. W. Calliss, “A Conceptual Design for a Code Analysis Knowledge Base,” *Journal of Software Maintenance: Research and Practice*, vol. 4, pp. 16-36, 1995.

[8] P. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong, "The Software Bookshelf," *IBM Systems Journal*, vol. 36, pp. 564-593, 1997.

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[10] D. Garlan and M. Shaw, *Software Architecture: Perspectives on an Emerging Discipline*: Prentice-Hall Press, 1996.

[11] D. Harel, "On Visual Formalisms," *Communications of the ACM*, vol. 31, pp. 514-530, 1988.

[12] R. C. Holt, "Software Bookshelf: Overview and Construction," , 1997.

[13] J. Jaakkola and P. Kilpeläinen, "The Sgrep Home Page," Department of Computer Science, University of Helsinki, Helsinki, Finland http://www.helsinki.fi/~jjaakkol/sgrep.html, 1999.

[14] R. Kazman and S. J. Carriere, "View Extraction and View Fusion in Architectural Understanding," presented at 5th International Conference of Software Reuse, Victoria, BC, 1998.

[15] S. Letovsky, "Cognitive Processes in Program Comprehension," presented at Empirical Studies of Programmers, First Workshop, 1986.

[16] Linore and Cleveland, "A program understanding support environment," *IBM Systems Journal*, vol. 28, pp. 324-344, 1989.

[17] D. C. Littman, J. Pinto, S. Letovsky, and E. Soloway, *Mental Models and Software Maintenance*. Norwood, New Jersey: Ablex Publishing, 1986.

[18] G. Marchionini, *Information Seeking in Electronic Environments*: Cambridge University Press, 1995.

[19] A. v. Mayrhauser and A.-M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, pp. 44-45, 1995.

[20] H. A. Muller, M. A. Orgun, S. R. Tilley, and J. S. Uhl, "A Reverse-Engineering Approach to Subsystem Structure Identification," *Software Maintenance: Research and Practice*, vol. 5, pp. 181-204, 1993.

[21] G. C. Murphy and D. Notkin, "Lightweight Lexical Source Model Extraction," *ACM Transactions of Software Engineering and Methodology*, vol. 5, pp. 262-292, 1996.

[22] S. Paul and A. Prakesh, "A Framework for Source Code Search Using Program Patterns," *IEEE Transactions on Software Engineering*, vol. 20, pp. 463-475, 1994.

[23] N. Pennington, "Stimulus Strutures and Mental Representations in Expert Comprehension of Computer Programs," *Cognitive Psychology*, vol. 19, pp. 295-341, 1987.

[24] D. A. Penny, "The Software Landscape: A Visual Formalism for Programming-in-the-Large," Ph.D. Thesis in Department of Computer Science. Toronto: University of Toronto, 1992.

[25] D. E. Perry and A. L. Wolf, "Foundations for the Study of Software Architecture," *ACM SIGSOFT*, vol. 17, pp. 40-52, 1992.

[26] B. Shneiderman, *Designing the User Interface for Effective Human-Computer Interaction*, 3rd ed: Addison-Wesley, 1998.

[27] S. E. Sim, "Supporting Multiple Program Comprehension Strategies During Software Maintenance," Master's Thesis in Department of Computer Science. Toronto: University of Toronto, 1998.

[28] S. E. Sim, C. L. A. Clarke, and R. C. Holt, "Archetypal Source Code Searches: A Survey of Software Developers and Maintainers," presented at 6th International Workshop on Program Comprehension, Ischia, Italy, 1998.

[29] S. E. Sim and R. C. Holt, "The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize," presented at 20th International Conference on Software Engineering, Kyoto, Japan, 1998.

[30] J. Singer, T. Lethbridge, N. Vinson, and N. Anquetil, "An Examination of Software Engineering Work Practices," presented at CASCON97, Toronto, Canada, 1997.

[31] J. Stasko, J. Domingue, and M. H. Brown, "Software Visualization: Programming as a Multimedia Experience," : The MIT Press, 1998.

[32] V. Tzerpos, R. C. Holt, and G. Farmaner, "Web-Based Presentation of Hierarchic Software Architecture," presented at Workshop on Software Engineering on the World Wide Web, Boston, MA, 1997.

[33] N. Wilde, A. Chapman, and R. Richardson, "The Extensible Dependency Analysis Tool Set: A Knowledge Base for Understanding Industrial Software," *International Journal of Software Engineering and Knowledge Engineering*, vol. 4, pp. 521-534, 1994.

[34] S. Wu and U. Manber, "Agrep-- A Fast Approximately Pattern-Matching Tool," presented at USENIX Winter 1992 Technical Conference, San Francisco, U.S.A., 1992.

[35] A. Yeh, D. Harris, and M. Chase, "Manipulated Recovered Software Architecture Views," presented at 19th International Conference on Software Engineering, Boston, MA, 1997.