Semantic Grep: Regular Expressions + Relational Abstraction

R. Ian Bull, Andrew Trevors, Andrew J. Malton, and Michael W. Godfrey
Software Architecture Group
University of Waterloo
Ontario, Canada, N2L 3G1
{irbull,adtrevors,ajmalton,migod}@uwaterloo.ca

Abstract

Searching source code is one of the most common activities of software engineers. Text editors and other support tools normally provide searching based on lexical expressions (regular expressions). Some more advanced editors provide a way to add semantic direction to some of the searches. Recent research has focused on advancing the semantic options available to text-based queries. Most of these results make use of heavy weight relational database management technology.

In this paper we explore the extension of lexical pattern matching by means of light weight relational queries, implemented using a tool called `grok`. A "semantic grep" (`sgrep`) command was implemented, which translates queries in a mixed algebraic and lexical language into a combination of `grok` queries and `grep` commands. This paper presents the design decisions behind `sgrep`, and example queries that can be posed. The paper concludes with a case study in which `sgrep` was used to identify architectural anomalies in PostgreSQL, an open source Database Management System.

1 Introduction

Lethbridge and Anquetil have reported that software engineers spend a considerable portion of their time exploring source code [15]. Exploring source can be classified into two categories: *searching* and *browsing* [16]. Sim et. al. have shown how these two separate navigation styles can be used to explore software architecture diagrams [25]. This paper shows how binary relational calculus and textual pattern matching can be used together as a lightweight means of exploring source code.

Lethbridge et. al. have studied in depth the work practices of professional software engineers in order to facilitate the design of a tool that would enhance their day-to-day activities [27]. Through their studies they have devised a list of requirements that a program understanding tools should contain. This list includes includes functional requirements such as:

F1 *search capabilities*
F2 *capabilities to display all relevant attributes of the items retrieved*

The list also includes non-functional requirements such as:

NF1 *ability to handle large amounts of code*
NF2 *respond to most queries without perceptible delay*
NF3 *process source code in a variety of programming languages*
NF4 *interoperate with other software engineering tools*
NF5 *permit the independent development of user interfaces*
NF6 *integrate facilities that Software Engineers already use*
NF7 *present the user with complete information*

We have used this list of requirements to develop a tool called `sgrep` which mixes regular expression matching and semantic querying.

`Sgrep` has been integrated with `grok` [13], a relational calculator, to allow the user to issue both semantic and syntactic queries and receive the results in a timely fashion.

1.1 Overview

The rest of this paper is structured as follows. Section 2 explores the motivation for the semantic grep tool. Section 3 discusses some related work in the field of source code searching and browsing. Section 4 outlines implementation considerations for a source exploration tool. Section 5 explains the `sgrep` tool itself and shows some sample queries. Section 6 shows how `sgrep` was used in practice to analyze PostgreSQL, an open source Database Management System [22]. Finally, section 7 summarizes our results and outlines our future work.

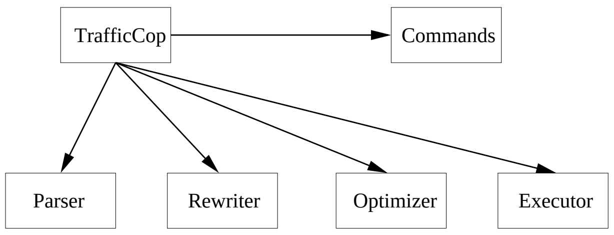

Figure 1. Conceptual Architecture of PostgreSQL

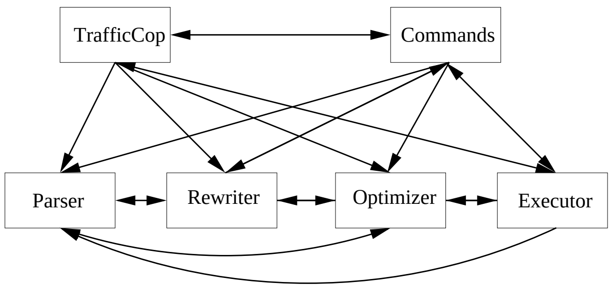

Figure 2. Concrete Architecture of PostgreSQL

2 Motivation

High level software documentation comes in many forms such as software bookshelves [10], architectural diagrams, module interconnection diagrams, and call graphs. In all cases, higher levels of abstraction are used to convey as much about the software system as possible while filtering out the low level details. By doing this the diagrams are less cluttered and easier to understand. In our work, we often look at two kinds of architectural diagrams: the “As Designed” model and the “As Built” model.

The “As Designed” or *conceptual* model shows what the developers think the system looks like or how it was designed. This model can be determined through interviews with developers, sifting through documentation and studying reference architectures. This may not give an accurate view of the system details, but it illustrates the subsystem interactions that the software engineers feel are important. An example of a conceptual architecture for PostgreSQL can be seen in figure 1 [31].

The “As Built” or *concrete* model of a system shows the relations that exist in the implemented system [2]. This model can be extracted through reverse engineering tools such as PBS, Dali, Rigi, CIA or SWAGKit [21, 29, 17, 11, 28]. Each of these tools produces different output; however, they all have the common goal of producing a higher level of abstraction of the system under examination based on low level “facts” extracted from the source code. Figure 2 shows the concrete architecture for PostgreSQL as extracted by SWAGKit [28].

Both of these models can help a software immigrant “Ramp-Up” on a new software project, [26] because these models are easy to navigate and provide a shared mental model of the code base [12]. Because of the high level of abstraction, diagrams such as these become less and less useful as the software engineer becomes more familiar with the code base and require detailed information about the source code. These diagrams are almost never used for maintenance activities and code migration. As the level of abstraction gets higher, it becomes more difficult to tie the types of questions software engineers have back to the source code where the experienced software engineer needs the results. Each of the subsystems in figure 2 contains additional subsystems, dozens of files and tens of thousands of lines of code. Due to the large complexity of each subsystem, navigating this diagrams often becomes harder than navigating the source using a regular expression matcher. Questions such as *why do these unexpected dependencies exist and where do they originate* are hard to answer without returning to the source code. The motivation for `sgrep` came from the need to answer detailed questions about the source code of PostgreSQL while navigating the architectural model. Section 6 returns to the PostgreSQL example and reviews the following questions in depth by using `sgrep`:

1. How can the differences between the conceptual architecture and the concrete one be reconciled.
2. How can the concrete architecture be repaired to fix the unexpected dependencies?

3 Related Work

The present ”semantic `grep`” proposal combines two basic ideas: regular pattern matching in text, and relational modeling of software structure.

3.1 Pattern Matching in Source Code

Browsing and searching texts or structures by specifying patterns is, of course, a facility provided in every text editor and development environment. In order to be flexible and fast, such facilities are almost without exception based on lexical structure and restricted to small predictable chunks of text, especially lines. (According to Dennis Ritchie [23], regular expression searching was introduced by Ken Thompson’s 1968 version of the QED editor. In QED and its successor, `ed`, the command `g/`*re*`/p`, where *re* is a regular expression, meant ”throughout the current document, for every line containing a match of *re*, print the line". That editor command was used to name the program `grep`, probably the most useful single shell program from the Unix tradition.)

Pattern languages usually (not always) belong to one of two classes, "wildcard patterns" and "regular expressions". In both classes, a pattern is a string of characters, some of which are intended to match themselves, and others to match one or more of a set or range of characters. They range in expressiveness from the simplest wildcard pattern language with two match facilities (one-character and many-character match) through full regular expressions with alternation, sequence, grouping, and iteration. Wildcard patterns are more commonly used for browsing short words, phrases, and names, such as file names; general regular expressions are more commonly used for browsing text.

Generally speaking, of course, source code texts are more structurally complex than what can be captured by regular languages. Paul and Prakash [20] discussed the many limitations of even full regular expression matching:

1. writing a pattern to distinguish nesting structure of programming language statements is difficult or impossible;
2. writing a pattern to distinguish variable declarations, characterized by order-independent strings of attributes (e.g. `extern long int x;`) is difficult and unwieldy;
3. many implementations of regular-expression matching do not allow patterns to match across line boundaries.

Such problems are familiar to any programmer trying to find (and be sure of finding) patterns of use in source files. In the case of searching source code, the solution is to extend the pattern language. This may be done by including some or all of the syntactic categories of the target programming languages, (SCRUPLE [20], GENOA [7]) by enriching the source with markup (Hypersoft [18], GRASP-ML [6], CHIME [8], Jupiter [5], HSML [4]) or by incorporating reference to a semantic model (Searchable Bookshelf [25, 21]), tksee [15, 27], Rigi [17], CIA [3], Dali [29], SWAGKit [28]. In the next subsection we discuss related work in software modeling.

Commercial-origin tools such as SNiFF+ (Wind River) and IDE's from Borland, Microsoft, IBM, Sun, etc. often combine these strategies with successful results: we applaud the trend which Sun and IBM have lead in making some of these tools open and free. However even though many available tools and frameworks allow both browsing and searching of semantic information, often they are "heavy weight": we must work from within them and their design decisions and accept their straightening limitations. For this reason `grep` is still the choice for many software engineers just because it it so simple. This was recently illustrated in a structured demonstration organized by Sim and Storey [24]. We hope that tools like `sgrep` can serve as a lightweight option for more advanced source queries.

Computes	**Operation**	**Purpose**
Relation	R1 **o** R2	Relational Composition
Relation	**id** (s)	Identity relation on set **s**
Relation	R **+**	Transitive closure of **R**
Set	s **.** R	Project set **s** through relation **R**
Set	R **.** s	Project set **s** through relation **R** backwards
Set	**rng** (R)	Range of **R**
Set	**dom** (R)	Domain of **R**

Table 1. Standard Grok Commands

3.2 Relational Modeling of Software

Software is modeled in reverse- and re-engineering systems to facilitate analysis and transformation, as well as browsing and searching. The technique is invariably to construct a semantic (E/R) graph, whose nodes represent entities (from expressions to subsystems) and whose edges represent relationships (implicit or explicit) found between them in the code. What varies in detail is (a) the degree of completeness, and (b) the strictness of adherence to a previously determined or stated schema.

In reverse- and re-enginering systems, an E/R model "extracted" or derived from code is then made available to users and other algorithms. This may be done by means of a standard relational database and SQL query system – although it appears that RDBMS technology is not a perfect fit (see below). Alternatively, the "purely relational" aspect of the extracted "fact base" can be processed using a relational-algebra engine. Several such engines have been described or mentioned in the reverse engineering and related literature, including RPA [9], RELVIEW [1], SCA [19], and `grok` [13, 14].

In what follows, we use `grok` to provide the relational computations required in the design and implementation of `sgrep`. The `grok` program implements a command and expression language: the commands are used for reading and writing fact (data) bases, and the expressions are relational and set expressions. The values of expressions are *identifiers*, *sets* (of identifiers), and *relations*. Identifiers are uninterpreted strings. Sets are finite. Relations are finite sets of pairs of identifiers. Expressions are constructed from operators and subexpressions: Table 1 contains a summary of the operators we use.

The range, domain, and composition of relations are standard. The identity relation on a set S consists of all the pairs (x, x) when $x \in S$. The transitive closure of a relation R consists of all the pairs (x_0, x_n) where there is a finite sequence or path $x_0, x_1, ..., x_n$, where $n > 1$ and each (x_i, x_{i+1}) is in R. The transitive closure is thus used to summarize hierarchical and other path data in the fact base.

The ”projection” operations of `grok` result from applying the relation or its converse to all the elements of a set. The left projection $S.R$ of a set S and a relation R is the set $\{t|s \in S\&(s,t) \in R\}$, that is, the set of identifiers which R relates to a member of S. The right (backwards) projection $R.S$ is the dual: $R.S$ is the set $\{t|s \in S\&(t,s) \in R\}$.

4 Implementation Considerations

Many of the research tools developed to address the problem of searching and browsing source code are not widely accepted as industrial tools. Often this is because they do not satisfy both the functional and non-functional requirements a software engineer has. In section 1, 2 functional and 7 non-functional requirements were listed for program understanding tools. NF3 is not considered relevant for this tool since `sgrep` just processes data models and these models can represent any program written in any language. We combine F1 and F2 into section 4.1 (Searching), NF1 and NF2 into section 4.2 (Performance) and NF5-NF7 into 4.3 (Interface). We also introduce the importance of *program complexity* in section 4.4, as a non-functional requirement.

4.1 Searching

Searching requires both an easy to specify pattern and the an accurate presentation of all relevant results. In the case of a program comprehension tool, designed to assist programmers better understand the low level details, it is important to tie the queries back to the source code. For this reason `sgrep` returns not only the query results, but allows the user to request entity attributes such as the file and line number the results appear on.

4.2 Performance

Performance is arguably the most important non-functional requirement when designing a navigation tool. It is important for the tool not only to handle large amounts of data, but the tool must respond to queries on this data without perceptible delay. If a tool does not respond in this manner, the user will find a tool that does, even if the tool may not provide as accurate a search. `Sgrep` was built on top of `grok`, a relational calculator which is extremely fast for navigating relational models. `Grok's` performance boost comes from the fact that the entire model is packed into memory, so no disk caching is required. The only bottleneck of this approach was the initial load time for the model.

There are several alternatives that were explored in order to retain the data model in core memory:

1. *A monolithic system* that starts up and allows the user to perform as many queries as they wish. This limits the performance hit to start-up only.

2. *A distributed system* that keeps the information in core and the client can connect to the server at run time.

Since the need for a command line interface was deemed important (see section 4.3) `sgrep` was designed as a distributed system. This allows the client to act as a small query tool (much like `grep, sed` or `awk`) without the overhead cost of loading the model each time.

4.3 Interface

Although graphical user interfaces are often easier to use for novice users, we believe that familiarity is more important than ease of use, for the kinds of tasks we envision for `sgrep`. `Sgrep` was designed as a command line tool for this reason. `Sgrep` is designed to be used in place of `grep`, so it is important that many of the design decisions found in `grep`, transfer over to `sgrep`.

Since `sgrep` is built in a distributed fashion is allows developers to independently develop user interfaces. The UI could be a graphical swing application or integrated into another development environment.

4.4 Program Complexity

In any field, ease of use and adaptability to the tasks at hand are what causes a tool to be adopted. The familiar Unix shell tools, such as `grep`, `sed`, `awk`, and `vi` or `emacs`, have been tested, proved, and entered the necessary toolkit of countless users. If a tool is hard to setup and administer, the value the tool adds to the development environment doesn't outweigh its administration cost. When the task is program comprehension, including text searching, many of the tools currently available depend on an industrial strength database management system. While these DBMSs provide many advantages such as:

1. a standardized query language
2. a published API and a rich set of tools
3. industrial-strength implementations (e.g. with concurrent access, disk c aching, security, etc.)

they require a large overhead cost of setup and configuration.

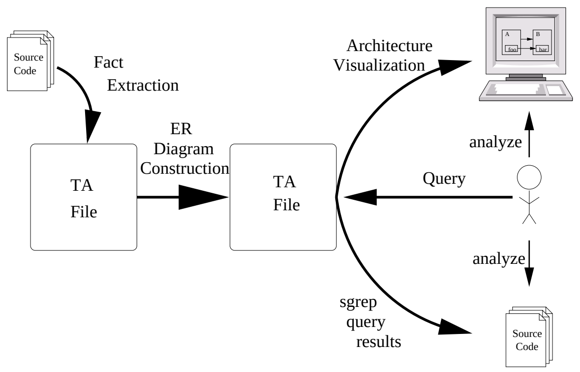

Figure 3. SWAGKit Pipeline

Other than the first of these, the advantages presented above are usually not required for program comprehension tools. A published API and a rich set of tools are not necessary since we are developing our own tool. Concurrent access is not needed, and usually the information can fit into memory so the need for good disk usage is lessened.

Even the fact that SQL is a universal standard does not play a large role, since SQL is not a universally adopted standard for source code exploration.

5 The Sgrep Tool

5.1 Design of Sgrep

`Sgrep` was designed to be used as part of the SWAGKit pipeline. SWAGKit provides a set of tools that can be used to extract information from the source code, and manipulate that information into a form that can be meaningfully analyzed. A tool, called `lsedit`, is then used to visually navigate this information as a software landscape. The intention of `sgrep` was to provide an alternate form of analysis where the user can query this information, and map it back to the source code. This is illustrated in Figure 3.

The tool itself, as mentioned in section 4, is designed in a client/server fashion. In this configuration, the client offers a command line interface by which the user can construct their queries. The server uses `grok` to load information about the system into memory and perform queries on it. The interaction between the client and server programs is illustrated in Figure 4 and described in the numbered list below.

1. SgrepServer is started by a user, and it creates a socket used for listening for client connections.
2. The information about the system is loaded into memory. SgrepServer then waits for client connection requests.

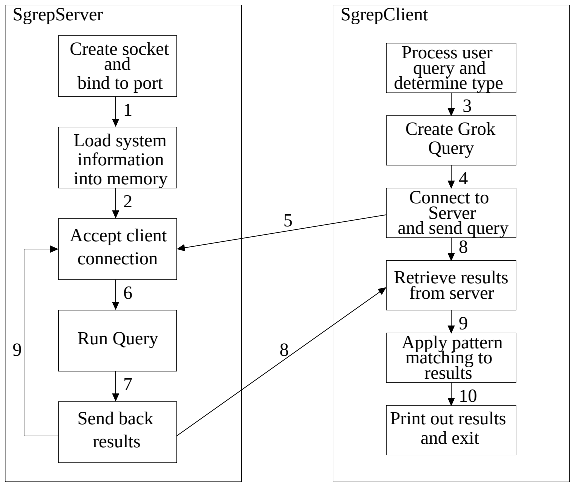
SgrepServer
Create socket and bind to port
1
Load system information into memory
2
Accept client connection
6
Run Query
7
Send back results
9
SgrepClient
Process user query and determine type
3
Create Grok Query
4
Connect to Server and send query
8
Retrieve results from server
9
Apply pattern matching to results
10
Print out results and exit
5
8

Figure 4. Client/Server Interaction

3. SgrepClient is started by a user, and the user's query is parsed to determine the query type.
4. A `grok` query is created based on the type of query to be performed.
5. SgrepClient connects to SgrepServer, and sends the query to run.
6. SgrepServer accepts a connect request from SgrepClient, and receives the query.
7. The script is run against the information about the system.
8. The results from the query are sent back to SgrepClient.
9. SgrepClient receives results from SgrepServer and terminates connection. SgrepServer then waits for the next request.
10. Pattern matching is applied to the results of the `grok` query. Those results that satisfy the pattern are printed.

5.2 Sgrep Queries

`Sgrep` allows the creation of both entity and relation queries. In addition to entity and relation queries on the entire system, `sgrep` also allows for entity and relation queries that are restricted to a particular subtree of the system. Relational transitive closure queries are also available so such information as function control flow can be determined. Each of these types of queries are briefly explained in the following subsections. In addition to the explanation an example is given, which includes the syntax of the `sgrep` query, its translation into a `grok` query, as well as the results of the query once pattern matching has been applied.

All query results are based on the example in figure 5. This figure was derived from the *Abstract Semantic Graph*

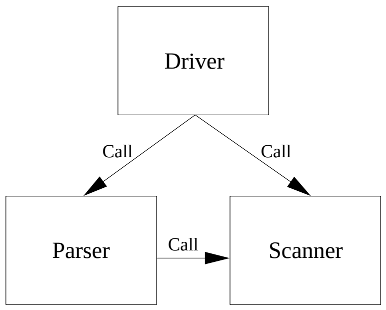

Figure 5. Landscape of a Compiler's Front End

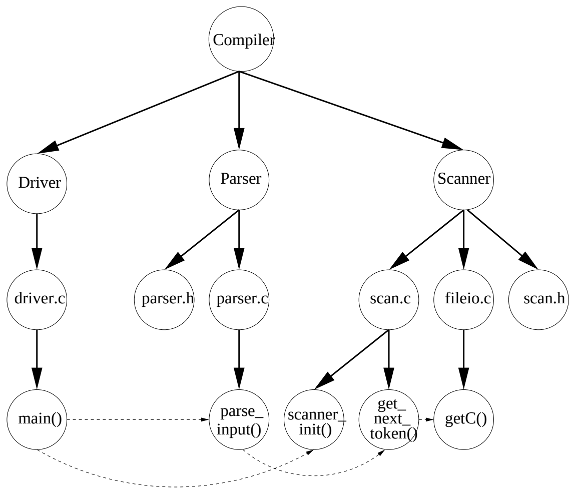

Figure 6. ASG of a Compiler's Front End

(ASG) presented in figure 6. This ASG could have been computed using a tool such as SWAGKit, and represents the interaction of program entities in the front end of a small compiler. The heavy black edges are the *structural edges* and show the containment hierarchy. The dashed edges are *non-structural* and in this particular diagram represent functions calls. The circles are program entities. The top two levels of the graph represent the subsystems that exist in the compiler. The third level represents source and header files, while the bottom level represents functions.

5.2.1 Entity Query

Entity queries are used to locate instances of a particular entity based on its type. For example, locate all functions in the system who's name matches the pattern "`get*`".

```
sgrep 'get* is function'
```

This `sgrep` query would translate into the following `grok` relational query:

```
$INSTANCE . { "function" }
```

This relational projects the set `function` backwards through the relation `$INSTANCE`. This results in the set of all entities that are instances of the type function. This set is then matched against the wild-card pattern "`get*`", resulting in:

```
get_next_token
getC
```

5.2.2 Restricted Entity Query

Restricted entity queries restrict the query to a particular *subtree*. A *subtree* is defined as any entity and all of its descendents. The example below searches for all functions in `parser.c`.

```
sgrep '∗ is function in parser.c'
```

This `sgrep` query would translate into the following `grok` relational query:

```
1   Desc := contain+
2   isParserDotC := id { "parser.c" }
3   isFunc := id( $INSTANCE . {"function"})
4   rng ( isParserDotC o Desc o isFunc )
```

This query first computes all the descendants in the system (line 1). A identity relation is then computed for both `parser.c` and all the functions in the system (line 2 & 3). [1] Finally, the relation from `parser.c` to all functions contained within the `parser.c` subtree is computed and the range of this relation returned. The pattern "`*`" then applied to this set resulting in:

```
parse_input
```

5.2.3 Relation Query

The purpose of a relation query is to find a particular type of relation that exists between two types of entities. For example, searching for all functions that `call getC`.

```
sgrep '∗ is function <calls> getC is *'
```

This `sgrep` query would translate into the following `grok` relational query:

```
isFunc o calls
```

[1]The variables `Desc`, and `isFunc` are used throughout the rest of the examples.

Query Type	**Syntax**
Entity Query	*pattern* **is** *entity*
Restricted Entity Query	*pattern* **is** *entity* **in** *pattern*
Relation Query	*pattern* **is** *entity* <*relation*> *pattern* **is** *entity*
Restricted Relation Query	*pattern* **is** *entity* **in** *pattern* <*relation*> *pattern* **is** *entity* **in** *pattern*
Relational Transitive Closure Query	*pattern* **is** *entity* <*relation+*> *pattern* **is** *entity*

Table 2. Sgrep Query Syntax

This `grok` query computes the relational composition from all functions (`isFunc` is defined in section 5.2.2) to everything else that can be reached through a `calls` relation. The pattern "`*`" is applied to the domain of the relation, and the pattern "`getC`" is applied to the range resulting in the following relations:

```
get_next_token calls getC
```

5.2.4 Restricted Relation Query

A restricted relation query is similar to a regular relation query except that the each of the two entities can be restricted to a subtree. For example, searching for functions in Parser which call functions in Scanner.

```
sgrep '* is function in Parser <calls>
       * is function in Scanner'
```

This `sgrep` query would translate into the following `grok` relational query:

```
1   isScanner := id { "Scanner" }
2   isParser := id { "Parser" }
3   ParserFunc := isParser o Desc o isFunc
4   ScanFunc := isScanner o Desc o isFunc
5   ParserFunc o calls o ScanFunc
```

An identity relation is first computed for the `Scanner` and `Parser` subsystems (line 1 & 2). All the functions in the `Scanner` and `Parser` are then computed (line 3 & 4). A relation is then constructed for each function in `Parser` which has a `call` relation to each function in `Scanner`. The pattern "`*`" is applied to both the domain and range of this relation resulting in the following:

```
parse_input calls get_next_token
```

5.2.5 Relational Transitive Closure Query

Relational transitive closure queries involve starting with an entity, and then following a particular type of relation until there are no more relations to follow. The example below searches for all functions that can be called starting at the function `main`.

```
sgrep 'main is function <calls+>'
```

This `sgrep` query would translate into the following `grok` relational query:

```
1   isFunc o ( calls + )
```

This query computes the transitive closure relation across the `calls` relation for all `calls` that originate from a `function`. The pattern `"main"` is then applied to the domain of the resulting relation resulting in the following:

```
main calls parse_input
main calls scanner_init
main calls get_next_token
main calls getC
```

6 Case Study

To demonstrate the capabilities of `sgrep`, an architectural analysis of PostgreSQL [22] was performed. PostgreSQL is an open source Object-Relational DBMS, which is maintained and supported by the Postgres Global Development Group, and a large number of contributors. The source consists of approximately 400 KLOC of C.

In section 2 two maintenance questions were posed about the architecture of PostgreSQL. The following sections will explore how `sgrep` was used in helping to answer these questions.

6.1 How can the differences between the conceptual architecture and the concrete one be reconciled?

The focus of this analysis was the investigation of the differences between the conceptual (Figure 1) and concrete (Figure 2) architectures of PostgreSQL. The conceptual architecture was derived from studying developer documentation found on the PostgreSQL website located at `http://postgresql.org`, while the concrete architecture was produced by using SWAGKit.

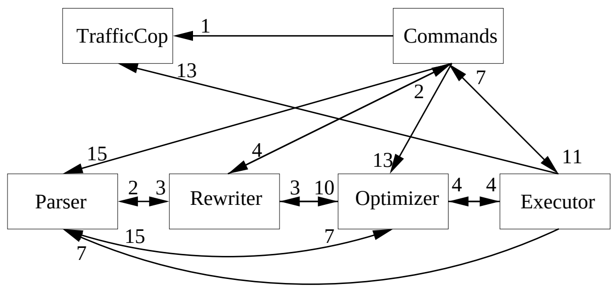

Figure 7. PostgreSQL Architectural Anomalies

Examining these differences in detail will help in performing architectural repair [30]. This type of repair is effective in correcting the conceptual architecture so that it accurately reflects the true architecture of the system. An accurate architecture will provide software engineers with a better understanding of the system, which will allow them to make better development and maintenance decisions.

The first step was to identify the subsystem dependencies in the concrete architecture that did not exist in the conceptual architecture. This step was done manually by analyzing the diagrams by hand and yielded 17 unexpected dependencies. Further investigation of these dependencies was done by using `sgrep`. A series of queries were run in order to determine what function calls were responsible for these unexpected dependencies. The example below shows the `sgrep` query used to identify the dependencies between the Executor and Optimizer subsystems, as well as the results.

```
sgrep '∗ is function in Executor
       <calls>
       ∗ is function in Optimizer'
```

```
ExecInitIndexScan calls get_rightop
ExecHashJoin calls get_leftop
ExecInitIndexScan calls get_leftop
ExecInitAgg calls pull_agg_clause
```

Queries, like the one above, were run for each of the 17 unexpected dependencies. The results of these queries are depicted in Figure 7. Each weighted arrow represents the number of unexpected function calls between the two subsystems. For instance, the Rewriter subsystem had 10 unexpected function calls to the Optimizer subsystem. In total, a list of 121 unexpected function calls were produced using `sgrep`.

6.2 How can the concrete architecture be repaired to fix the unexpected dependencies?

The analysis performed in the previous section produced 17 unexpected dependencies between subsystems that translated to 121 function calls. The next step is to examine each function call individually and determine which of the following two categories it falls in:

- The function call corresponds to behaviour that was left out of the conceptual architecture because it was minute and considered unimportant to the overall architecture, the creator(s) of the conceptual didn't know about it, or because it was introduced after the conceptual architecture was created.
- Part, or all, of the call or callee function resides in a file or subsystem in which it shouldn't.

In the former case, a dependency is simply added to the conceptual architecture. The latter case, however, involves modifying the source code itself and may result in an addition or deletion of a dependency in the concrete architecture, or it might not effect it at all. This type of change may also result in the addition of new subsystems as well. In order to make a decision about each function call, the source code must be analyzed. To assist in locating the function call (and the function being called), `sgrep` was used to determine the file and line number of the call and callee function. An example is shown below.

```
sgrep 'ExecInitIndexScan is function'
```

```
ExecInitIndexScan
file: nodeIndexscan.c (line 575)
```

```
sgrep 'get_righttop is function'
```

```
get_rightop
file: clauses.c (line 153)
```

Using this information, the source code is analyzed and the function call is categorized as mentioned above. After the analysis was completed it was proposed that the function `get_rightop` might be a candidate to be moved to another subsystem. The function was being used by both the Command and Executor subsystems as a generic function to retrieve operands from clauses. This function was not the only one being used in this manner, a number of functions which dealt with simple operations on different data structures were defined throughout the system and used by multiple subsystems. The intent was to determine whether it was ideal to remove them and place them in a new subsystem for generic operations on data structures.

Since this would involve modifying the source code, which could alter the concrete architecture, `sgrep` was used to determine the impact this move would have. This required finding all the functions that call the function to be moved, as well as all the functions that it calls. The example below shows the queries used to find which functions `get_rightop` interacts with.

```
sgrep '* is function <calls>
       get_righttop is function'
```

```
CheckPredClause calls get_rightop
ExecInitIndexScan calls get_rightop
switch_outer calls get_rightop
create_hashjoin_plan calls get_rightop
check_hashjoinable calls get_rightop
check_mergejoinable calls get_rightop
.
.
.
flatten_andors calls get_rightop
get_rels_atts calls get_rightop
get_relattval calls get_rightop
```

```
sgrep 'get_righttop is function<calls>
       * is function'
```

```
  No results.
```

From these results it was calculated that 23 functions were using `get_rightop`, and that `get_rightop` was not using any functions itself. Of the 23 functions calling `get_rightop`, only two of them were not in the Optimizer subsystem. In this case it was decided since the Optimizer subsystem used the function so heavily, and other subsystems (Commands and Executor) only used it twice, that moving the function was probably not necessary.

7 Summary and Future Work

The limitations of `grep` and lexical querying are well known, however it is still a popular tool because of its simple, light weight nature. This paper presents the idea of using binary relational calculus to quickly navigate system structure and component interaction. The advantages of DBMS systems to accomplish the same task are realized, however the ease of use, and performance increase gained from a simpler tool out weight the features of these larger systems.

Sgrep has been used on a small example to quickly navigate source code, and on a larger system to identify architectural anomalies. Software Engineers can use this tool to help ”ramp up” on a new project, explore source for maintenance activities and identify source reachability. Software Architects can also use this tool to identify subsystem interaction and containment hierarchy.

We consider that there is room for a tool like this incorporated in an *Integrated Development Environment* (IDE) such as the Microsoft Development Studio or IBM Visual Age. Often the IDEs do incremental parsing for syntax highlighting and tab completion. If the relational model was updated in core incrementally these queries could easily be applied.

We also believe that the results can easily be tied back to the original source since each entity contains attributes such as *line number*, *column number* and *source file*. The attributes can be extracted using an additional composition operation on the final results. By tying the queries back to the original source this will help the transition towards forward engineering.

Acknowledgements

We would like to thank Jen Campbell and Josh Taylor for their help with the development of `sgrep`. We would also like to thank Ric Holt for his valuable feedback on the `sgrep` interface and supplying us with a version of `grok` equipped with an API.

References

[1] R. Berghammer, B. von Karger, and C. Ulke. *Relation-algebraic analysis of Petri nets with RELVIEW*, pages 49–69. Springer-Verlag, Berlin, Passau, March 1996.

[2] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: Its extracted software architecture. In *Proc. of the* 21st *Intl. Conference on Software Engineering (ICSE-21)*, Los Angeles, CA, May 1999.

[3] Y. Chen, M. Nishimoto, and C. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, pages 3:325–334, 1990.

[4] J. Cordy, K. Schneider, T. Dean, and A. Malton. HSML: Design directed source code hot spots. In *Proc. of International Workshop on Program Comprehension*, Toronto, Canada, May 2001.

[5] Anthony Cox and Charles Clarke. Representing and accessing extracted information. In *Proc. of 2001 International Conference on Software Maintenance, Florence, Italy*, November 2001.

[6] J. Cross and T. Hendrix. Using generlized markup and SGML for reverse engineering graphical representa-

tions of sfotware. In *Proceedings of Second Working Conference on Reverse Engineering*, pages 2–6, Toronto, 1995.

[7] P. Devanba. GENOA - a customizable, front-end-retagetable source code analysis framework. *ACM Trans. on Software Engineering and Methodology*, 8(2):177–212, April 1999.

[8] P. Devanba, Y.-F. Chen, E. Gansner, H. Muller, and J. Martin. CHIME: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *Proc. 21st International Conference on Software Engineering*, pages 473–482, Los Angeles, May 1999.

[9] L. Feijs, R. Krikhaar, and R. Van Ommering. A Relational Approach to Support Software Architecture Analysis. *Software-Practice and Experience*, 28(4):371–400, 1998.

[10] Pat Finnigan, Richard C. Holt, Ivan Kalas, Scott Kerr, Kostas Kontogiannis, Hausi Müller, John Mylopoulos, Steve Perelgut, Martin Stanley, and Kenny. Wong. The software bookshelf. *IBM Systems Journal*, 36(4), November 1997.

[11] Judith Grass and Yin-Farn Chen. The C++ Information Abstractor. In *The Second USENIX C++ Conference*, April 1990.

[12] Ric Holt. Software Architecture as a Shared Mental Model. In *Proc. of 2002 International Workship on Program Comprehension, Paris, France*, May 2002.

[13] Richard C. Holt. The Grok Programming Language. http://plg.uwaterloo.ca/ holt/papers/grok-intro.html.

[14] Richard C. Holt. Binary relational algebra applied to software architecture. Technical Report 345, University of Toronto CSRI, 1996.

[15] Timothy C. Lethbridge and Nicolas Anquetil. Architecture of Source Code Exploration Tools: A Software Engineering Case Study. Technical Report 97-07, University of Ottawa, Computer Science, 1997.

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

[17] Hausi A. Müller and Karl Klashinsy. Rigi: A system for programming-in-the-large. In *Proc. of the* 10^{th} *Intl. Conference on Software Engineering (ICSE-10)*, Singapore, April 1988.

[18] J. Paukki, A. Salminen, and J. Koskinen. Automated hypertext support for software maintenance. *The Computer Journal*, 39(7):577–599, 1996.

[19] S. Paul and A. Prakash. A query algebra for program databases. *IEEE Trans. Software Engineering*, 22(3):202–217, 1996.

[20] Santanu Paul and Atul Prakash. A Framework for Source Code Search Using Program Patterns. *IEEE Transactions of Software Engineering*, 20(6), June 1994.

[21] PBS The Portable Bookshelf. Website. http://www.swag.uwaterloo.ca/pbs.

[22] PostgreSQL. Website. http://www.postgresql.org.

[23] Dennis Ritchie. An incomplete history of the QED text editor. http://cm.bell-labs.com/cm/cs/who/dmr/qed.html.

[24] Susan Elliot Sim and Margaret-Anne D. Storey. A Structured Demonstration of Program Comprehension Tools. In *Proc. of 2000 Working Conference on Reverse Engineering (WCRE-00)*, pages 184–193, Brisbane, Australia, November 2000.

[25] Susan Elliott Sim, Charles L.A. Clarke, Richard C. Holt, and Anthony M. Cox. Browsing and Searching Software Architectures. In *Proceedings of International Conference on Software Maintenance, Oxford, England*, 1999.

[26] Susan Elliott Sim and Richard C. Holt. The Ramp-Up Problem in Software Projects: A Case Study of How Software Immigrants Naturalize. In *Proc. of the 20th International Conference on Software Engineering*, , Kyoto, Japan, 1998.

[27] Janice Singer, Timothy C. Lethbridge, and Norman Vinson. Work Practices as an Alternative Method to Assist Tool Design in Software Engineering. In *Proceedings of International Workshop on Program Comprehension, Italy*, pages 173–179, 1998.

[28] Software Architecture Toolkit. Website. http://www.swag.uwaterloo.ca/swagkit.

[29] The Dali Architecture Reconstruction Workbench. Website. http://www.sei.cmu.edu/ata/products_services/dali.html.

[30] John B. Tran and R. C. Holt. Forward and reverse repair of software architecture. In *Proc. of CASCON 1999*, Toronto, November 1999.

[31] Andrew Trevors, Jen Campbell, and Josh Taylor. Conceptual Architecture of PostgreSQL. Prepared for CS798 at the University of Waterloo, 2002.