

Proceedings
2nd International Workshop on
Mining Software Repositories

MSR 2005

Proceedings
*2nd International Workshop on
Mining Software Repositories*
MSR 2005

Saint Louis, Missouri, USA
17th May 2004

Co-located With
International Conference on
Software Engineering
(ICSE 2005)

Edited by
Ahmed E. Hassan, Richard C. Holt, and Stephan Diehl



Contents

International Workshop on Mining Software Repositories

MSR 2005

<i>Message from the Workshop Chairs</i>	i
<i>Program Committee</i>	ii
<i>Additional Reviewers</i>	ii
<i>Program</i>	iii

Understanding Evolution and Change Patterns

Understanding Source Code Evolution Using Abstract Syntax Tree Matching.....	2
<i>Julian Neamtii, Jeffrey Foster, and Michael Hicks</i>	
Recovering System Specific Rules from Software Repositories.....	7
<i>Chadd Williams, and Jeffrey K. Hollingsworth</i>	
Mining Evolution Data of a Product Family.....	12
<i>Michael Fischer, Johann Oberleitner, Jacek Ratzinger, and Harald Gall</i>	
Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones	17
<i>Miryung Kim, and David Notkin</i>	

Defect Analysis

When do changes induce fixes?.....	24
<i>Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller</i>	
Error Detection by Refactoring Reconstruction.....	29
<i>Carsten Görg, and Peter Weißgerber</i>	

Education

Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System.....	36
<i>Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh</i>	
Mining Student CVS Repositories for Performance Indicators.....	41
<i>Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson</i>	

Text Mining

Toward Mining "Concept Keywords" from Identifiers in Large Software Projects.....	48
<i>Masaru Ohba, and Katsuhiko Gondow</i>	
Source code that talks: an exploration of Eclipse task comments and their implication to repository mining	53
<i>Annie Ying, James Wright, and Steven Abrams</i>	
Text Mining for Software Engineering: How Analyst Feedback Impacts Final Results.....	58
<i>Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundara</i>	

Software Changes and Evolution

Analysis of Signature Change Patterns.....	64
<i>Sunghun Kim, James Whitehead, and Jennifer Bevan</i>	
Improving Evolvability through Refactoring.....	69
<i>Jacek Ratzinger, Michael Fischer, Johann Oberleitner, and Harald Gall</i>	
Linear Predictive Coding and Cepstrum coefficients for mining time variant information from software repositories.....	74
<i>Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi</i>	

Process and Collaboration

Repository Mining and Six Sigma for Process Improvement.....	80
<i>Michael VanHilst, Pankaj Garg, and Christopher Lo</i>	
Mining Version Histories for Verifying Learning Process of Legitimate Peripheral Participants.....	84
<i>Shih-Kun Huang, and Kang-Min Liu</i>	

Taxonomies & Formal Representations

Towards a Taxonomy of Approaches for Mining of Source Code Repositories.....	90
<i>Huzeza Kagdi, Michael Collard, and Jonathan Maletic</i>	
A Framework for Describing and Understanding Mining Tools in Software Development.....	95
<i>Daniel German, Davor Cubranic, and Margaret-Anne D. Storey</i>	
SCQL: A formal model and a query language for source control repositories.....	100
<i>Abram Hindle, and Daniel German</i>	

Integration and Collaboration

Developer identification methods for integrated data from various sources.....	106
<i>Gregorio Robles, and Jesús M. González-Barahona</i>	
Accelerating Cross-Project knowledge Collaboration Using Collaborative Filtering and Social Networks.....	111
<i>Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken-ichi Matsumoto</i>	
Collaboration Using OSSmole: A repository of FLOSS data and analyses.....	116
<i>Megan Conklin, James Howison, and Kevin Crowston</i>	

Message From Workshop Chairs

MSR 2005

Welcome to MSR 2005, the 2nd international workshop on Mining Software Repositories. MSR 2005 brings together researchers and practitioners to consider methods of using data stored in software repositories to further understanding of software development practices. We expect the presentations and discussions in this workshop to facilitate the definition of challenges, ideas and approaches to transform software repositories from static record keeping systems to active repositories used by researchers to gain empirical understanding of software development, and by software practitioners to predict and plan various aspects of their project.

We received a large number of submissions – 38 papers from 14 countries. After the review process, 22 papers were chosen for publication. All accepted papers are presented. In order to fit all talks within the workshop day and based on input from the Program Committee during the review process, 11 papers are presented as Regular talks and 11 papers are presented as Lightning talks. Following the Lightning talks, we allocated an hour of informal discussions and demos, in which attendees are encouraged to interact with all presenters on topics of interest.

We are grateful for the excellent and professional review job done by the reviewers on such a tight schedule.

Ahmed E. Hassan
Richard C. Holt
University of Waterloo

Stephan Diehl
Catholic University Eichstätt

Program Committee

MSR 2005

Alexander Dekhtyar, *University of Kentucky, USA*
Premkumar T. Devanbu, *University of California at Davis, USA*
Stephen G. Eick, *SSS Research Inc., USA*
Harald Gall, *University of Zurich, Switzerland*
Les Gasser, *University of Illinois at Urbana Champaign, USA*
Daniel German, *University of Victoria, Canada*
Jane Huffman Hayes, *University of Kentucky, USA*
Katsuro Inoue, *Osaka University, Japan*
Philip Johnson, *University of Hawaii, USA*
Timothy C. Lethbridge, *University of Ottawa, Canada*
Gail Murphy, *University of British Columbia, Canada*
Audris Mockus, *Avaya Labs Research, USA*
Thomas J. Ostrand, *AT&T Research, USA*
Dewayne Perry, *University of Texas, USA*
Jelber Sayyad Shirabad, *University of Ottawa, Canada*
Annie Ying, *IBM Research, USA*
Andreas Zeller, *Saarland University, Germany*

Additional Reviewers

MSR 2005

Davor Cubranic, *University of Victoria, Canada*
Cory Kapser, *University of Waterloo, Canada*
Jingwei Wu, *University of Waterloo, Canada*
Thomas Zimmermann, *Saarland University, Germany*



MSR 2005: International Workshop on Mining Software Repositories
msr.uwaterloo.ca

9:00-9:15	Welcome and Introduction [slides] <i>Ahmed E. Hassan, Richard C. Holt, and Stephan Diehl</i>
9:15-10:30	Session 1: Understanding Evolution and Change Patterns <ul style="list-style-type: none"> • Understanding Source Code Evolution Using Abstract Syntax Tree Matching [slides] Iulian Neamtii, Jeffrey Foster, and Michael Hicks (University of Maryland) • Recovering System Specific Rules from Software Repositories [slides] Chadd Williams, and Jeffrey K. Hollingsworth (University of Maryland) • Mining Evolution Data of a Product Family [slides] Michael Fischer, Johann Oberleitner, Jacek Ratzinger (Vienna University of Technology), and Harald Gall (University of Zürich) • Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones [slides] Miryung Kim, and David Notkin (University of Washington)
10:30-11:00	Coffee Break
11:00-11:45	Session 2: Defect Analysis <ul style="list-style-type: none"> • When do changes induce fixes? [slides] Jacek Sliwinski (Max Planck Institute for Computer Science), Thomas Zimmermann, and Andreas Zeller (Saarland University) • Error Detection by Refactoring Reconstruction [slides] Carsten Görg (Saarland University), and Peter Weißgerber (Catholic University Eichstätt)
11:45-12:30	Session 3: Education <ul style="list-style-type: none"> • Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System [slides] Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh (University of Maryland) • Mining Student CVS Repositories for Performance Indicators [slides] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson (University of Toronto)
12:30-1:45	Lunch

1:45-3:45	<p>Session 4: Lightning Talks (5 mins each) and Walkaround Presentations [info]</p> <ul style="list-style-type: none"> • Session 4A: Text Mining <ul style="list-style-type: none"> ○ Toward Mining "Concept Keywords" from Identifiers in Large Software Projects [slides] Masaru Ohba, and Katsuhiko Gondow (Tokyo Institute of Technology) ○ Source code that talks: an exploration of Eclipse task comments and their implication to repository mining [slides] Annie Ying, James Wright, and Steven Abrams (IBM Research) ○ Text Mining for Software Engineering: How Analyst Feedback Impacts Final Results [slides] Jane Huffman Hayes, Alex Dekhtyar, and Senthil Karthikeyan Sundaram (University of Kentucky) • Session 4B: Software Changes and Evolution <ul style="list-style-type: none"> ○ Analysis of Signature Change Patterns [slides] Sunghun Kim, James Whitehead, and Jennifer Bevan (University of California, Santa Cruz) ○ Improving Evolvability through Refactoring [slides] Jacek Ratzinger, Michael Fischer, Johann Oberleitner (Vienna University of Technology), and Harald Gall (University of Zürich) ○ Linear Predictive Coding and Cepstrum coefficients for mining time variant information from software repositories [slides] Giuliano Antoniol, Vincenzo Fabio Rollo, and Gabriele Venturi (University of Sannio) • Session 4C: Process and Collaboration <ul style="list-style-type: none"> ○ Repository Mining and Six Sigma for Process Improvement [slides] Michael VanHilst (Florida Atlantic University), Pankaj Garg (Zee Source), and Christopher Lo (Florida Atlantic University) ○ Mining Version Histories for Verifying Learning Process of Legitimate Peripheral Participants [slides] Shih-Kun Huang, and Kang-Min Liu (National Chiao Tung University) • Session 4D: Taxonomies & Formal Representations <ul style="list-style-type: none"> ○ Towards a Taxonomy of Approaches for Mining of Source Code Repositories [slides] Huzefa Kagdi, Michael Collard, and Jonathan Maletic (Kent State University) ○ A Framework for Describing and Understanding Mining Tools in Software Development [slides] Daniel German, Davor Cubranic, and Margaret-Anne D. Storey (University of Victoria) ○ SCQL: A formal model and a query language for source control repositories [slides] Abram Hindle, and Daniel German (University of Victoria)
3:45-4:00	Coffee Break
4:00-5:00	<p>Session 5: Integration and Collaboration</p> <ul style="list-style-type: none"> • Developer identification methods for integrated data from various sources [slides] Gregorio Robles, and Jesús M. González-Barahona (Universidad Rey Juan Carlos) • Accelerating cross-project knowledge collaboration using collaborative filtering and social networks [slides] Masao Ohira, Naoki Ohsugi, Tetsuya Ohoka, and Ken-ichi Matsumoto (Nara Institute of Science and Technology) • Collaboration Using OSSmole: A repository of FLOSS data and analyses [slides] Megan Conklin (Elon University), James Howison, and Kevin Crowston (Syracuse University)
5:00-5:30	<p>Wrap-up: Common Themes and Future Direction [slides] <i>Ahmed E. Hassan, Richard C. Holt and Stephan Diehl</i></p>

Understanding Evolution and Change Patterns

Understanding Source Code Evolution Using Abstract Syntax Tree Matching

Iulian Neamtii
neamtii@cs.umd.edu

Jeffrey S. Foster
jfoster@cs.umd.edu

Michael Hicks
mwh@cs.umd.edu

Department of Computer Science
University of Maryland at College Park

ABSTRACT

Mining software repositories at the source code level can provide a greater understanding of how software evolves. We present a tool for quickly comparing the source code of different versions of a C program. The approach is based on partial abstract syntax tree matching, and can track simple changes to global variables, types and functions. These changes can characterize aspects of software evolution useful for answering higher level questions. In particular, we consider how they could be used to inform the design of a dynamic software updating system. We report results based on measurements of various versions of popular open source programs, including BIND, OpenSSH, Apache, Vsftpd and the Linux kernel.

Categories and Subject Descriptors

F.3.2 [Logics And Meanings Of Programs]: Semantics of Programming Languages—*Program Analysis*

General Terms

Languages, Measurement

Keywords

Source code analysis, abstract syntax trees, software evolution

1. INTRODUCTION

Understanding how software evolves over time can improve our ability to build and maintain it. Source code repositories contain rich historical information, but we lack effective tools to mine repositories for key facts and statistics that paint a clear image of the software evolution process.

Our interest in characterizing software evolution is motivated by two problems. First, we are interested in *dynamic software updating* (DSU), a technique for fixing bugs or adding features in running programs without halting service [4]. DSU can be tricky for programs whose types change,

so understanding how the type structure of real programs changes over time can be invaluable for weighing the merits of DSU implementation choices. Second, we are interested in a kind of “release digest” for explaining changes in a software release: what functions or variables have changed, where the hot spots are, whether or not the changes affect certain components, etc. Typical release notes can be too high level for developers, and output from `diff` can be too low level.

To answer these and other software evolution questions, we have developed a tool that can quickly tabulate and summarize simple changes to successive versions of C programs by partially matching their abstract syntax trees. The tool identifies the changes, additions, and deletions of global variables, types, and functions, and uses this information to report a variety of statistics.

Our approach is based on the observation that for C programs, function names are relatively stable over time. We analyze the bodies of functions of the same name and match their abstract syntax trees structurally. During this process, we compute a bijection between type and variable names in the two program versions. We then use this information to determine what changes have been made to the code. This approach allows us to report a name or type change as single difference, even if it results in multiple changes to the source code. For example, changing a variable name from `x` to `y` would cause a tool like `diff` to report all lines that formerly referred to `x` as changed (since they would now refer to `y`), even if they are structurally the same. Our system avoids this problem.

We have used our tool to study the evolution history of a variety of popular open source programs, including Apache, OpenSSH, Vsftpd, Bind, and the Linux kernel. This study has revealed trends that we have used to inform our design for DSU. In particular, we observed that function and global variable additions are far more frequent than deletions; the rates of addition and deletion vary from program to program. We also found that function bodies change quite frequently over time, but function prototypes change only rarely. Finally, type definitions (like `struct` and `union` declarations) change infrequently, and often in simple ways.

2. APPROACH

Figure 1 provides an overview of our tool. We begin by parsing the two program versions to produce abstract syntax trees (ASTs), which we traverse in parallel to collect type and name mappings. With the mappings at hand, we then detect and collect changes to report to the user, either

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR '05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

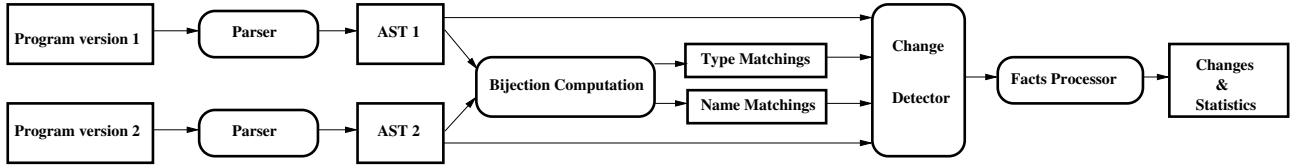


Figure 1: High level view of AST matching

<pre> typedef int sz_t; int count; struct foo { int i; float f; char c; }; int baz(int a, int b) { struct foo sf; sz_t c = 2; sf.i = a + b + c; count++; } </pre>	<pre> int counter; typedef int size_t; struct bar { int i; float f; char c; }; int baz(int d, int e) { struct bar sb; size_t g = 2; sb.i = d + e + g; counter++; } void biff(void) { } </pre>
---	---

Figure 2: Two successive program versions

directly or in summary form. In this section, we describe the matching algorithm, illustrate how changes are detected and reported, and describe our implementation and its performance.

2.1 AST Matching

Figure 2 presents an example of two successive versions of a program. Assuming the example on the left is the initial version, our tool discovers that the body of `baz` is unchanged—which is what we would like, because even though every line has been syntactically modified, the function in fact is structurally the same, and produces the same output. Our tool also determines that the type `sz_t` has been renamed `size_t`, the global variable `count` has been renamed `counter`, the structure `foo` has been renamed `bar`, and the function `biff()` has been added. Notice that if we had done a line-oriented `diff` instead, nearly all the lines in the program would have been marked as changed, providing little useful information.

To report these results, the tool must find a *bijection* between the old and new names in the program, even though functions and type declarations have been reordered and modified. To do this, the tool begins by finding function names that are common between program versions; our assumption is that function names do not change very often. The tool then uses partial matching of function bodies to determine name maps between old and new versions, and finally tries to find *bijections* i.e., one-to-one, onto submaps of the name maps.

We traverse the ASTs of the function bodies of the old and new versions in parallel, adding entries to a *LocalNameMap* and *GlobalNameMap* to form mappings between local variable names and global variable names, respectively. Two variables are considered equal if we encounter them in the same syntactic position in the two function bodies. For example, in Figure 2, parallel traversal of the two versions of `baz` results in the *LocalNameMap*

$a \leftrightarrow d, b \leftrightarrow e, sf \leftrightarrow sb, c \leftrightarrow g$

and a *GlobalNameMap* with `count` \leftrightarrow `counter`. Similarly,

```

procedure GENERATEMAPS(Version1, Version2)
   $F_1 \leftarrow$  set of all functions in Version 1
   $F_2 \leftarrow$  set of all functions in Version 2
  global TypeMap  $\leftarrow \emptyset$ 
  global GlobalNameMap  $\leftarrow \emptyset$ 
  for each function  $f \in F_1 \cap F_2$ 
  do {  $AST_1 \leftarrow$  AST of  $f$  in Version 1
        $AST_2 \leftarrow$  AST of  $f$  in Version 2
        $MATCH\_AST(AST_1, AST_2)$  }

procedure MATCH_AST( $AST_1, AST_2$ )
  local LocalNameMap  $\leftarrow \emptyset$ 
  for each ( $node_1, node_2$ )  $\in (AST_1, AST_2)$ 
  do {
    if ( $node_1, node_2$ ) = ( $t_1 x_1, t_2 x_2$ ) //declaration
    then {  $TypeMap \leftarrow TypeMap \cup \{t_1 \leftrightarrow t_2\}$ 
           $LocalNameMap \leftarrow LocalNameMap \cup \{x_1 \leftrightarrow x_2\}$ 
        }
    else if ( $node_1, node_2$ ) = ( $y_1 := e_1, y_2 := e_2 \text{ op } e_2$ )
    then {  $MATCH\_AST(e_1, e_2)$ 
           $MATCH\_AST(e_1, e_2)$ 
          if  $isLocal(y_1)$  and  $isLocal(y_2)$  then
             $LocalNameMap \leftarrow LocalNameMap \cup \{y_1 \leftrightarrow y_2\}$ 
          else if  $isGlobal(y_1)$  and  $isGlobal(y_2)$  then
             $GlobalNameMap \leftarrow GlobalNameMap \cup \{y_1 \leftrightarrow y_2\}$ 
          else if ...
          else break
        }
  }
  
```

Figure 3: Map Generation Algorithm

we form a *TypeMap* between named types (`typedefs` and aggregates) that are used in the same syntactic positions in the two function bodies. For example, in Figure 2, the name map pair `sb` \leftrightarrow `sf` will introduce a type map pair `struct foo` \leftrightarrow `struct bar`.

We define a *renaming* to be a name or type pair $j_1 \rightarrow j_2$ where $j_1 \leftrightarrow j_2$ exists in the bijection, j_1 does not exist in the new version, and j_2 does not exist in the old version. Based on this definition, our tool will report `count` \rightarrow `counter` and `struct foo` \rightarrow `struct bar` as renamings, rather than additions and deletions. This approach ensures that consistent renamings are not presented as changes, and that type changes are decoupled from value changes, which helps us better understand how types and values evolve.

Figure 3 gives pseudocode for our algorithm. We accumulate global maps *TypeMap* and *GlobalNameMap*, as well as a *LocalNameMap* per function body. We invoke the routine *MATCH_AST* on each function common to the two versions. When we encounter a node with a declaration $t_1 x_1$ (a declaration of variable x_1 with type t_1) in one AST and $t_2 x_2$ in the other AST, we require $x_1 \leftrightarrow x_2$ and $t_1 \leftrightarrow t_2$. Similarly, when matching statements, for variables y_1 and y_2 occurring in the same syntactic position we add type pairs in the *TypeMap*, as well as name pairs into *LocalNameMap* or *GlobalNameMap*, depending on the storage class of y_1 and y_2 . *LocalNameMap* will help us detect functions which are identical up to a renaming of local and formal variables, and *GlobalNameMap* is used to detect renamings for global variables and functions. As long as the ASTs have the same shape, we keep adding pairs to maps. If we encounter an AST mismatch (the **break** statement on the last line of the algorithm), we stop the matching process for that function and use the maps generated from the portion of the tree that did match.

```

----- Global Variables -----
Version1 : 1
Version2 : 1
renamed : 1

----- Functions -----
Version1 : 1
Version2 : 2
added : 1
locals/formals name changes : 4

----- Structs/Unions -----
Version1 : 1
Version2 : 1
renamed : 1

----- Typedefs -----
Version1 : 1
Version2 : 1
renamed : 1

```

Figure 4: Summary output produced for the code in Figure 2

The problem with this algorithm is that having insufficient name or type pairs could lead to renamings being reported as additions/deletions. The two reasons why we might miss pairs are partial matching of functions and function renamings. As mentioned previously, we stop adding pairs to maps when we detect an AST mismatch, so when lots of functions change their bodies, we miss name and type pairs. This could be mitigated by refining our AST comparison to recover from a mismatch and continue matching after detecting an AST change. Because renamings are detected in the last phase of the process, functions that are renamed don't have their ASTs matched, another reason for missing pairs. In order to avoid this problem, the bijection computation and function body matching would have to be iterated until a fixpoint is reached.

In practice, however, we found the approach to be reliable. For the case studies in section 3, we have manually inspected the tool output and the source code for renamings that are improperly reported as additions and deletions due to lack of constraints. We found that a small percentage (less than 3% in all cases) of the reported deletions were actually renamings. The only exception was an early version of Apache (versions 1.2.6-1.3.0) which had significantly more renamings, with as many as 30% of the reported deletions as spurious.

2.2 Change Detection and Reporting

With the name and type bijections in hand, the tool visits the functions, global variables, and types in the two programs to detect changes and collect statistics. We categorize each difference that we report either as an addition, deletion, or change.

We report any function names present in one file and not the other as an addition, deletion, or renaming as appropriate. For functions in both files, we report that there is a change in the function body if there is a difference beyond the renamings that are represented in our name and type bijections. This can be used as an indication that the semantics of the function has changed, although this is a conservative assumption (i.e., semantics preserving transformations such as code motion are flagged as changes). In our experience, whenever the tool detects an AST mismatch, manual inspection has confirmed that the function seman-

```

/ : 111
include/ : 109
linux/ : 104
fs.h : 4
ide.h : 88
reiserfs_fs_sb.h : 1
reiserfs_fs_i.h : 2
sched.h : 1
wireless.h : 1
hdreg.h : 7
net/ : 2
tcp.h : 1
sock.h : 1
asm-i386/ : 3
io_apic.h : 3
drivers/ : 1
char/ : 1
agp/ : 1
agp.h : 1
net/ : 1
ipv4/ : 1
ip_fragment.c : 1

```

Figure 5: Density tree for struct/union field additions (Linux 2.4.20 vs. 2.4.21)

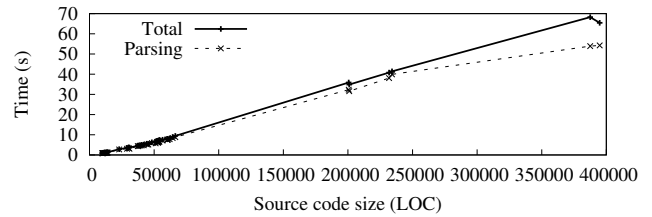


Figure 6: Performance

tics has indeed changed.

We similarly report additions, deletions and renamings of global variables, and changes in global variable types and static initializers.

For types we perform a deep structural isomorphism check, using the type bijection to identify which types should be equal. We report additions, deletions, or changes in fields for aggregate types; additions, deletions, or changes to base types for typedefs; and additions, deletions, or changes in item values for enums.

Our tool can be configured to either report this detailed information or to produce a summary. For the example in Figure 2, the summary output is presented in Figure 4. In each category, **Version1** represents the total number of items in the old program, and **Version2** in the new program. For brevity we have omitted all statistics whose value was 0 e.g., enums, etc.

Our tool can also present summary information in the form of a *density tree*, which shows how changes are distributed in a project. Figure 5 shows the density tree for the number of struct and union fields that were added between Linux versions 2.4.20 and 2.4.21. In this diagram, changes reported at the leaf nodes (source files) are propagated up the branches, making clusters of changes easy to visualize. In this example, the `include/linux/` directory and the `include/linux/ide.h` header file have a high density of changes.

2.3 Implementation

Our tool is constructed using CIL, an OCaml framework

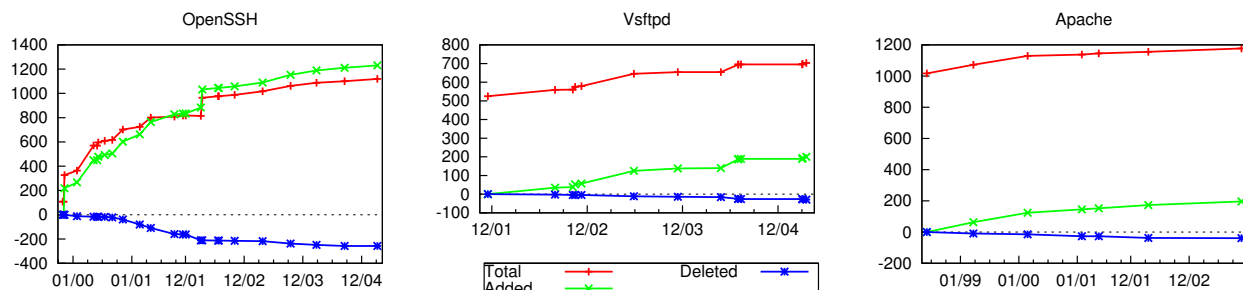


Figure 7: Function and global variable additions and deletions

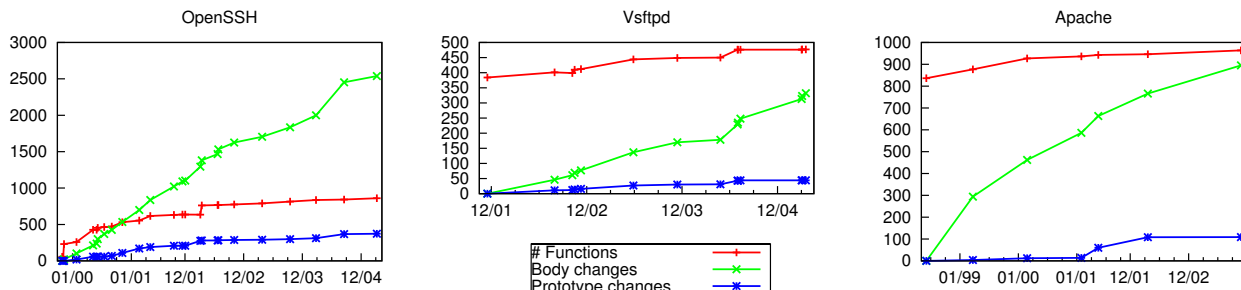


Figure 8: Function body and prototype changes

for C code analysis [3] that provides ASTs as well as some other high-level information about the source code. We have used it to analyze the complete lifetime of Vsftpd (a “very secure” FTP server, see <http://beasts.vsfptd.org/>) and OpenSSH (daemon); 8 snapshots in the lifetime of Apache 1.x; and portions of the lifetimes¹ of the Linux kernel (versions 2.4.17, Dec. 2001 to 2.4.21, Jun. 2003) and BIND (versions 9.2.1, May 2002 to 9.2.3, Oct. 2003).

Figure 6 shows the running time of the tool on these applications (we consider the tool’s results below), plotting source code size versus running time.² The top line is the total running time while the bottom line is the portion of the running time that is due to parsing, provided by CIL (thus the difference between them is our analysis time). Our algorithm scales roughly linearly with program size, with most of the running time spent in parsing. Computing changes for two versions of the largest test program takes slightly over one minute. The total time for running the analysis on the full repository (i.e., all the versions) for Vsftpd was 21 seconds (14 versions), for OpenSSH was 168 seconds (25 versions), and for Apache was 42 seconds (8 versions).

3. CASE STUDY: DYNAMIC SOFTWARE UPDATING

This section explains how we used the tool to characterize software change to guide our design of a dynamic software updating (DSU) methodology [4]. We pose three questions concerning code evolution; while these are relevant for DSU, we believe they are of general interest as well. We answer

¹Analyzing earlier versions would have required older versions of gcc.

²Times are the average of 5 runs. The system used for experiments was a dual Xeon@2GHz with 1GB of RAM running Fedora Core 3.

these questions by using the output of our tool on the programs mentioned above, which are relevant to DSU because they are long-running.

Are function and variable deletions frequent, relative to the size of the program? When a programmer deletes a function or variable, we would expect a DSU implementation to delete that function from the running program when it is dynamically updated. However, implementing on-line deletion is difficult, because it is not safe to delete functions that are currently in use (or will be in the future). Therefore, if definitions are rarely deleted over a long period, the benefit of cleaning up dead code may not be worth the cost of implementing a safe mechanism to do so. Figure 7 illustrates how OpenSSH, Vsftpd, and Apache have evolved over their lifetime. The x-axis plots time, and the y-axis plots the number of function and global variable definitions for various versions of these programs. Each graph shows the total number of functions and global variables for each release, the cumulative number of functions/variables added, and the cumulative number of functions/variables deleted (deletions are expressed as a negative number, so that the sum of deletions, additions, and the original program size will equal its current size). The rightmost points show the current size of each program, and the total number of additions and deletions to variables and functions over the program’s lifetime.

According to the tool, Vsftpd and Apache delete almost no functions, but OpenSSH deletes them steadily. For the purposes of our DSU question, Vsftpd and Apache could therefore reasonably avoid removing dead code, while doing so for OpenSSH would have a more significant impact (assuming functions are similar in size).

Are changes to function prototypes frequent? Many DSU methodologies cannot update a function whose type has

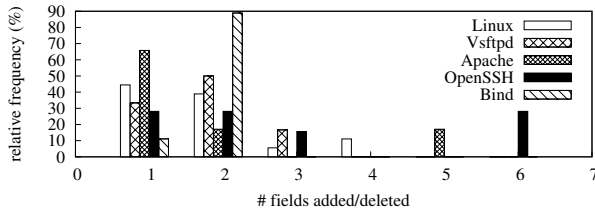


Figure 9: Classifying changes to types

changed. If types of functions change relatively infrequently, then this implementation strategy may be able to support a large number of updates. Figure 8 presents graphs similar to those in Figure 7. For each program, we graph the total number of functions, the cumulative number of functions whose body has changed, and the cumulative number of functions whose prototype has changed. As we can see from the figure, changes in prototypes are relatively infrequent for Apache and Vsftpd, especially compared to changes more generally. In contrast, functions and their prototypes have changed in OpenSSH far more rapidly, with the total number of changes over five years roughly four times the current number of functions, with a fair number of these resulting in changes in prototypes. In all cases we can see *some* changes to prototypes, meaning that supporting prototype changes in DSU is a good idea.

Are changes to type definitions relatively simple? In most DSU systems, changes to type definitions (which include `struct`, `union`, `enum`, and `typedef` declarations in C programs) require an accompanying *type transformer function* to be supplied with the dynamic update. Each existing value of a changed type is converted to the new representation using this transformer function. Of course, this approach presumes that such a transformer function can be easily written. If changes to type definitions are fairly complex, it may be difficult to write a transformer function.

Figure 9 plots the relative frequency of changes to `struct`, `union`, and `enum` definitions (the y-axis) against the number of fields (or enumeration elements for `enums`) that were added or deleted in a given change (the x-axis). The y-axis is presented as a percentage of the total number of type changes across the lifetime of the program. We can see that most type changes affect predominantly one or two fields. An exception is OpenSSH, where changing more than two fields is common; it could be that writing type transformers for OpenSSH will be more difficult. We also used the tool to learn that fields do not change type frequently (not shown in the figure).

4. RELATED WORK

A number of systems for identifying differences between programs have been developed. We discuss a few such systems briefly.

Yang [5] developed a system for identifying “relevant” syntactic changes between two versions of a program, filtering out irrelevant ones that would be produced by `diff`. Yang’s solution matches parse trees (similar to our system) and can even match structurally different trees using heuristics. In contrast, our system stops at the very first node mismatch in order not to introduce spurious name or type bijections. Yang’s tool cannot deal with variable renaming

or type changes, and in general focuses more on finding a *maximum syntactic similarity* between two parse trees. We take the semantics of AST nodes into account, distinguish between different program constructs (e.g., types, variables and functions) and specific changes associated with them.

Horwitz [1] proposed a system for finding *semantic*, rather than syntactic, changes in programs. Two programs are semantically identical if the sequence of observable values they produce is the same, even if they are textually different. For example, with this approach semantics-preserving transformations such as code motion or instruction reordering would not be flagged as a change, while they would in our approach. Horwitz’s algorithm runs on a limited subset of C that does not include functions, pointers, or arrays.

Jackson and Ladd [2] propose a differencing tool that analyzes two versions of a procedure to identify changes in dependencies between formals, locals, and globals. Their approach is insensitive to local variable names, like our approach, but their system performs no global analysis, does not consider type changes, and sacrifices soundness for the sake of suppressing spurious differences.

5. CONCLUSION

We have presented an approach to finding semantic differences between program versions based on partial abstract syntax tree matching. Our algorithm uses AST matching to determine how types and variable names in different versions of a program correspond. We have constructed a tool based on our approach and used it to analyze several popular open source projects. We have found that our tool is efficient and provides some insights into software evolution.

We have begun to extend the tool beyond matching ASTs, to measure evolution metrics such as common coupling or cohesion [6]. We are interested in analyzing more programs, with the hope that the tool can be usefully applied to shed light on a variety of software evolution questions.

6. REFERENCES

- [1] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, June 1990.
- [2] D. Jackson and D. A. Ladd. Semantic diff: A tool for summarizing the effects of modifications. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, pages 243–252, Sept. 1994.
- [3] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.
- [4] G. Stoye, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu. *Mutatis Mutandis*: Safe and flexible dynamic software updating. In *Proceedings of the ACM SIGPLAN/SIGACT Conference on Principles of Programming Languages (POPL)*, pages 183–194, January 2005.
- [5] W. Yang. Identifying Syntactic differences Between Two Programs. *Software - Practice and Experience*, 21(7):739–755, 1991.
- [6] E. Yourdon and L. L. Constantine. *Structured Design, 2nd Ed.* Yourdon Press, New York, 1979.

Recovering System Specific Rules from Software Repositories

Chadd C. Williams
Department of Computer Science
University of Maryland
chadd@cs.umd.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
hollings@cs.umd.edu

Abstract

One of the most successful applications of static analysis based bug finding tools is to search the source code for violations of system-specific rules. These rules may describe how functions interact in the code, how data is to be validated or how an API is to be used. To apply these tools, the developer must encode a rule that must be followed in the source code. The difficulty is that many of these system-specific rules are undocumented and "grow" over time as the source code changes. Most research in this area relies on expert programmers to document these little-known rules. In this paper we discuss a method to automatically recover a subset of these rules, function usage patterns, by mining the software repository. We present a preliminary study that applies our work to a large open source software project.

1 Introduction

Static analysis of source code has been used very successfully to locate bugs in software. One of the most successful applications of static analysis to find bugs has been tools that look for violations of system-specific rules in the source code. Source code must adhere to a large number of rules that describe how data should be handled, how to interact with objects or APIs and how to use functions safely. Violations of these system-specific rules are often a source of error [5].

The difficulty with these rules is that they are implicit and dynamic. As the source code changes new rules are added and old rules are removed. When functions are added to an API a new set of rules must be followed that describe how they are to be used. It is challenging for the

developers of a widely distributed project to keep track of the rules the code must follow. This task is complicated by the fact that many of these rules are not documented as they are created, or are only documented in a CVS commit message or an email on a developer mailing list.

This leaves the project to rely on developers learning these rules in a number of unsatisfactory ways. For example, senior developers relating the rules that they know to new developers, developers searching CVS commit messages and mailing lists when they have a question or code reading. New developers are not the only ones to suffer. Senior developers need to keep up on the rules being added and removed from the source code.

In this paper we propose recovering these system-specific rules by studying the changes made to the source code. We specifically focus on rules that describe function usage patterns, how functions should be invoked in relation to each other. We believe that these usage patterns can shed light on how an external API should be used or how internal functions should interact. We have developed a tool that analyzes each version of a file in the software repository and determines what new function usage patterns are introduced in subsequent versions of each file.

2 Related Work

There has been ample research in the area of detecting violations of system-specific rules to identify bugs. One such system, metal [2], allows the user to supply patterns to match against the source code and flag as warnings. The patterns the developer supplies are encoded via state machines that are then applied to the source code. This system has been used to find a large number of errors (500) in real software projects. The metal system was also used to try to infer system specific patterns that should be checked [3]. While Engler, et al., look only at the current source code, our work focuses on looking at the changes made to the source code over time and what system specific rules these changes highlight.

Work has also been done to validate the notion that violations of system-specific rules cause a significant number of the errors seen in software [5]. Matsumura, et al., describe a case study that shows 32% of failures detected during the maintenance phase of a software project were due to violations of implicit code rules. The

This work was supported in part by DOE Grants DE-FG02-93ER25176, DE-FG02-01ER25510, and DE-CFC02-01ER254489 and NSF award EIA-0080206.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MSR'05, May 17, 2005, Saint Louis, Missouri, USA

Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

implicit rules used to check the source code were generated by ‘expert’ programmers.

The need for information sharing in large, distributed open source software projects has been studied. Gutwin, et al., studied the need for *group awareness*, knowledge about who is doing what in the project [4]. One of the aspects of awareness they describe is *feedthrough*, which is defined as observations of changes to project artifacts to indicate who has been doing what.

There has also been work on identifying frequently applied changes to source code through mining the software change history [7]. Rysselberghe and Demeyer state that frequently applied changes can be used to study how software maintenance proceeds and to suggest solutions to future problems. They look for both system specific change patterns and more general patterns. While our work studies the state of the code after a change is made, their work looks exclusively at the changes applied to the code.

Pinzger and Gall identify patterns to recover software architecture [6]. They use code patterns specified by the user, and data describing the associations of these patterns, to reconstruct higher-level patterns describing the software architecture.

3 Function Usage Patterns

The system-specific rules that we are studying in this work are *function usage patterns*. We want to determine how functions are invoked with respect to each other, specifically which functions are often called in close proximity within the source code. Instances of these patterns in a software project build up a set of *relationships* between functions. We define an *instance of a function usage pattern* as a set of two particular function call sites such that the pattern template is satisfied. We will explore the relationship aspect in Section 5.3. Experience suggests that there are sets of functions that are smaller parts of the implementation of a larger conceptual goal that need to be invoked together. These functions may operate on common data, provide error recovery functionality or perform some type of pair-wise functionality like lock/unlock. The two specific function usage patterns we are looking for are the *called after* and *conditionally called after* patterns. The *called after* relation is simple, function X is called after function Y in the source code of some function Z. The *conditionally called after* pattern describes the case where function X is called after function Y, but its invocation is guarded by a conditional statement. These two function usage patterns

```
HDC hdc = BeginPaint( hwnd, &ps );
if( hdc )
    DrawIcon( hdc, x, y, hIcon );
EndPaint( hwnd, &ps );
```

Figure 1a: Called After Pattern

are the only two that we investigated for our preliminary study. Figure 1a provides an example of the *called after* pattern. Figure 1b provides an example of the *conditionally called after* pattern. Each of these code snippets highlight one instance of a function usage pattern identified by our tool in the Wine source code [10]. The code snippets have been edited for clarity.

There are a number of other patterns that might be useful. For example, in Figure 1b the function `GetProcessHeap` is called and its return value is used as an argument to both `HeapAlloc` and `HeapFree`. This type of pattern involving dataflow is something we plan to study in the future. A similar usage pattern is evident in Figure 1a between the functions `BeginPaint` and `DrawIcon`.

4 Our Tool

Our tool is very simple and casts a very wide net in terms of the instances of patterns it finds. This gives us the freedom to put off making decisions on how to filter the data until later in the process. This is important as retrieving the data from the software repository and generating our results is the most computationally expensive aspect of this work.

We use the framework developed for our previous work in mining software repositories to manage the data from the CVS repository and the results produced by our tool [9]. In summary, the data from the CVS repository and the raw results are stored in a database.

The tool we have produced is merely a prototype to support this preliminary study. It is based on the Edison Design Group C parser [1]. The tool parses the source file and scans for function call sites. Within each function in the source file, two function usage patterns are applied to each function call site. For every function call site in a function, every other function call site located later in that function is involved with it in a *called after* pattern (unless the later call site is guarded by an conditional). For each instance of a pattern, the tool records the names of each function, the line numbers of the call sites and the name of the enclosing function. The same process is used to determine *conditionally called after* patterns, with a bit more analysis to identify which functions are guarded by conditionals.

4.1 Mining the Source Code Repository

When mining the software repository we are looking for an instance of a function usage pattern in a revision of

```
mdi_cs = HeapAlloc(GetProcessHeap());
if (!mdi_cs)
    HeapFree(GetProcessHeap(), 0, cs);
```

Figure 1b: Conditionally Called After Pattern

a file, where that instance of the pattern did not exist in the revision immediately prior. We are looking for new instances of patterns entering the code. Specifically with this tool we are looking for either a *called after* or *conditionally called after* pattern that did not exist in the previous revision of the file. Note that we are doing this on a per file, rather than on a per function, basis.

4.2 Identifying New Instances of Patterns

Once the data is mined from the source code repository and stored in the database, we must analyze it to determine when a new instance of a pattern has been added to the source code. Since our tool casts such a wide net in identifying patterns we need some way to filter the data. We have chosen, as a simple heuristic, to only look at instances of patterns that involve function invocations that are separated by no more than 10 lines of source code. This heuristic was chosen with the notion that many functions in an API need to be invoked in quick succession and that error handling, a possible target for the *conditionally called after* pattern, usually happens in close proximity to the error producing function.

In the future, we plan on refining this heuristic to be based on a deeper analysis of control flow. For example, the entry and exit basic blocks of a function may contain some function pairs that perform some type of paired functionality (lock/unlock). The basic blocks before a control flow split and after a control flow union may contain function calls related in some interesting way. Also looking at the type of conditional may be interesting. The conditional of a *while* loop versus that of an *if* statement may provide an important distinction between the applications of the *conditionally called after* pattern.

4.3 Transitive Patterns

Currently the patterns we are searching for are binary. The specific patterns we are searching for may be transitive in some cases, allowing larger relationship to be created. If a call to function *foo* is often followed by a call to *bar*, which is often followed by a call to *zoo*, then a call to *foo* may often be followed by a call to *zoo*. This transitivity may or may not exist. The context in which *bar* follows *foo* may be different from the context in which *zoo* follows *bar*. We may find we need to add more context information to our tool to differentiate usage patterns for a particular context. Section 5.3 contains a discussion of how to visualize the patterns mined from the source code.

5 Wine Case Study

We have used our tool to mine the software repository for the Wine project to determine what types of patterns can be recovered [10]. Each revision of each file has been analyzed by our tool. All instances of patterns that our

tool finds are recorded in a database, tagged with the file and revision in which the pattern appeared.

Our tool identified over 50 million instances of these two patterns in the software repository. There were over 2,175 unique instances of patterns that were added to the source code 10 or more times. Sixty-five unique patterns were added to the source code 100 times or more. Many of these 65 patterns dealt with functions that manage the heap or provide tracing or debugging functionality.

5.1 Called After Pattern

As shown in Figure 1a, this pattern involves two functions, one called after the other. It is very simple and our goal with this pattern was to identify chains of functionality that need to be performed together. Our tool identified a number of patterns of this type, 1,253 unique instances of this pattern that were added to the source code 10 or more times. Some of the patterns identified were obvious, and while these did not provide novel insight, they did provide evidence that our analysis was working as expected. As mentioned, many of the instances found involved the heap management functions. In the Wine source code, almost every function that manipulates their internal heap must first retrieve the heap for the current process via `GetProcessHeap`. Consequently, many heap manipulation functions such as `HeapAlloc` and `RtlAllocateHeap` are called in close proximity to `GetProcessHeap`.

Our tool also identified a number of patterns that represent a notion of *paired functionality*. These patterns include pairings such as `BeginPaint` and `EndPaint`, `GlobalLock` and `GlobalUnlock` and `EnterCriticalSection/LeaveCriticalSection`. Again, these instances of the pattern are mainly interesting to validate the results.

A more interesting instance of the pattern involves the functions `DeleteCriticalSection` and `HeapFree`. In this case, once a critical section object has been deleted, the memory allocated for that object needs to be deallocated. This data structure appears to always be allocated off the internal heap (we also found, as another instance of the pattern, `HeapAlloc` followed by `InitializeCriticalSection`) and the memory on the heap needs to be freed to do this. Another instance of the pattern is `LoadCursorA` and `RegisterClassA`. The latter function takes as a parameter a data structure representing a class. One field of that data structure must be initialized with the return from the function `LoadCursorA`.

It is instructive to look at the categories of functionality that are being discovered in instances of these patterns. Table 1 shows how many new instances of the *called after* pattern fall into a selected group of categories. The number of new instances is broken down by how many times a particular instance of a pattern was flagged as new during the software repository mining.

Category	New Instances		
	> 99	99 - 25	24 - 10
Debug	17	80	278
Heap	14	16	16
String Manipulation	3	41	153
GUI	3	22	271
Memory	7	28	19
Paired Functionality	0	8	39
Error Handling	0	9	30

Table 1: Function Pairing Categories for Called After

Table 1 shows that debug statements are heavily used in the Wine source code. There are 97 instances of function usage patterns that involve a debug function and were added to the source code at least 25 times. This means that there are 97 functions that are called in close proximity to a particular debug function.

The instances of the pattern listed in the Heap category are instances in which both functions involved are part of the heap interface. There are a total of 46 instances found in the code, indicating that functionality provided by the heap interface may require a number of function calls.

The category Paired Functionality contains instances of the pattern where the invoked functions provide functionality that needs to surround some bit of code. This includes such function pairings as `BeginPaint/EndPaint` and `GlobalLock/GlobalUnlock`. Eight such instances were added to the code between 25 and 99 times. Many of these instances involve some type of synchronization.

5.2 Conditionally Called After Pattern

The *conditionally called after* pattern is shown in Figure 1b. Our goal with this pattern was to see whether or not adding a small amount of control flow context to the pattern would help to elicit more interesting patterns. We expected this pattern to be able to identify error handling code and debugging idioms, instances of code where the second function is only called if the first function fails. Many of the instances of this pattern our tool identified supported this expectation. Our tool found 922 unique instances of this patterns that were added to the source code 10 or more times.

One of the instances involved the function `RegQueryValueExA` being conditionally called after `RegOpenKeyA`. In this case, the function `RegOpenKeyA` may or may not find a key in the registry. If it is successful the value can be queried. The insight here is that the developer cannot assume a key exists and should do the proper error checking to ensure that it was found properly.

Another interesting instance of this pattern is conditionally calling `SetLastError` after calling `HeapAlloc`. This instance of the pattern describes how errors should be propagated in the code. Table 2 shows how many new instances of the *conditionally called after*

Category	New Instances		
	> 99	99 - 25	24 - 10
Debug	14	95	341
Heap	7	8	11
String Manipulation	0	25	121
GUI	0	3	94
Memory	0	19	17
Paired Functionality	0	6	26
Error Handling	0	3	34

Table 2: Function Pairing Categories for Conditionally Called After

pattern fall into a selected group of categories based on functionality.

5.3 Visualization

While the patterns we are searching for are binary, the functions involved may be part of many different instances of the pattern. Because of the type of patterns we are searching for, two functions that are each involved separately in an instance of a pattern with a common third function may themselves be related. This serves to build up a web of relationships, similar to those studied in the area of social networks. We have used a social network viewer, TouchGraph LinkBrowser [8], to explore the relationships between functions. Figure 2 shows the neighborhood of the network centered on `BeginPaint` and `EndPaint`.

Looking at this network graph gives quick insight into the functions that are invoked in close proximity to both `BeginPaint` and `EndPaint`. The function `BeginPaint` and `EndPaint` are used to wrap access to drawing functionality. We expect functions that provide this functionality to be found in instances of the *called after* pattern with either or both of these functions. The network in Figure 2 shows this clearly. We can see that `SetTextColors` and `GetClientRect`, for example, are attached to each of these functions. Further, the thin end of the edge is attached to the function which is called after the function at the thick end of the edge. We can see that `GetClientRect` is called after `BeginPaint`, and

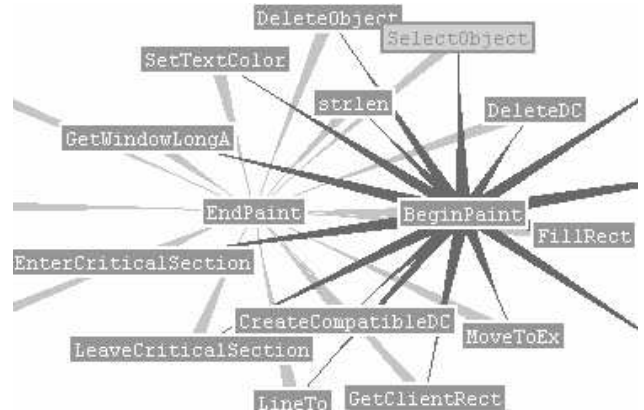


Figure 2: Social Network for BeginPaint/EndPaint

EndPoint is called after GetClientRect.

6 Why Mine the Full Repository?

We have chosen to mine each revision of each file to obtain a finer level of detail about changes made to the software. Since we gather data on what instances of patterns were added at each CVS transaction, we can investigate how instances of patterns entered the source code. Instances that are added to the source code steadily over time (over a large number of CVS transactions) may indicate a very important, frequently used pattern or a pattern that causes confusion among developers. On the other hand, patterns that are added to the source code in a relatively small number of CVS transactions may indicate refactoring. Determining the profile of how a pattern is added to the code may be useful in deciding the importance of that pattern, how to apply the instance in the future or how likely the pattern is to be misused by developers.

7 Future Work

The work we have presented here is still in its early stages. We have looked at only one software repository, and have only searched for instances of two patterns. In the future we will expand the number and complexity of patterns we search for and apply this technique to more software projects. We also do not track removed patterns. Knowing what patterns have been removed from the code could be useful in keeping an up-to-date list of important patterns in the project.

Mining the software repository of the Wine project has produced an enormous amount of data, a total of over 50 million instances of these two patterns were found in the repository. As we continue to work with this data we will need to find better ways of filtering out the more important, or more likely to be important, patterns. Currently our filter is based on the distance between, in terms of lines of code, the call sites of the two functions in the pattern. Clearly there is room for improvement. A filter that takes into account the files or directories the called functions (or the calling function) reside in may be useful in pulling out usage patterns of functions in the same module. Filters based on control flow graphs and deeper analysis of conditionals will provide more context as to the surrounding source code. Dataflow analysis as well will provide more context and may serve to provide a stronger link between two function calls. Finally, we need to not only think about patterns in terms of function calls. Patterns based on how data is accessed in a function, what parts of a structure need to be initialized or updated, need to be investigated as well.

We also need to explore how to use the instances of these patterns that are mined from the software repository. Providing these instances of patterns to a knowledge repository or as an appendix to a developer's guide may

be a useful way to inform developer's of the system-specific rules the source code. Potentially more interesting is the use of instances of these patterns to automatically identify problems in the code. This may be done by feeding the rules into static analysis tools that identify violations of the rules in the source code.

8 Conclusions

In this paper we have demonstrated how system-specific rules, in this case function usage patterns, can be recovered from source code change histories. We have run a preliminary study to recover such rules from a large, open source software project. This study has recovered a number of interesting and non-obvious rules that we think are critical for developers to understand and follow.

9 References

- [1] Edison Design Group, <http://www.edg.com/cpp.html>
- [2] Engler, D., Chelf, B., Chou, A., Hallem, S., Checking System Rules Using System Specific, Programmer-Written Compiler Extensions. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA, October 2000.
- [3] Engler, D., Chen, D. Y., Hallem, S., Chou, A., Chelf, B., Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code, In *Proceedings of the ACM symposium on Operating Systems Principles*, Banff, Canada, Oct 2001.
- [4] Gutwin, C., Penner, R., Schneider, K., Group Awareness in Distributed Software Development, In *Proceedings of ACM Conference on Computer Supported Cooperative Work*, Chicago, IL, Nov 2004.
- [5] Matsumura, T., Monden, A., Matsumoto, K., The Detection of Faulty Code Violating Implicit Coding Rules, *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE '02)*, Orlando, FL, USA, May 2002.
- [6] Pinzger, M., Gall, H., Pattern-supported architecture recovery. In *Proceedings of the International Workshop on Program Comprehension (IWPC'02)*, Paris, France, June 2002.
- [7] Rysselberghe, F., Demeyer, S., Mining Version Control Systems for FACs (Frequently Applied Changes), *Proceedings of International Workshop on Mining Software Repositories (MSR '04)*, Edinburgh, Scotland, UK, May 2004.
- [8] TouchGraph LinkBrowser, Available online at <http://touchgraph.sourceforge.net>
- [9] Williams, C. C., Hollingsworth, J. K., Bug Driven Bug Finders, In *Proceedings of International Workshop on Mining Software Repositories (MSR '04)*, Edinburgh, Scotland, UK, May 2004.
- [10] Wine, Available online at <http://www.winehq.org>

Mining Evolution Data of a Product Family*

Michael Fischer, Johann Oberleitner and Jacek Ratzinger
Distributed Systems Group
Information Systems Institute
Technical University of Vienna
A-1040 Vienna, Austria
{fischer,oberleitner,ratzinger}@infosys.tuwien.ac.at

Harald Gall
University of Zurich
Department of Informatics
s.e.a.l. – software
evolution & architecture lab
{gall}@ifi.unizh.ch

ABSTRACT

Diversification of software assets through changing requirements impose a constant challenge on the developers and maintainers of large software systems. Recent research has addressed the mining for data in software repositories of single products ranging from fine- to coarse grained analyses. But so far, little attention has been paid to mining data about the evolution of product families. In this work, we study the evolution and commonalities of three variants of the BSD (Berkeley Software Distribution), a large open source operating system. The research questions we tackle are concerned with how to generate high level views of the system discovering and indicating evolutionary highlights. To process the large amount of data, we extended our previously developed approach for storing release history information to support the analysis of product families. In a case study we apply our approach on data from three different code repositories representing about 8.5GB of data and 10 years of active development.

1. INTRODUCTION

Unanticipated evolution of a single software system enforced through changing requirements can lead to diversification and will result in different closely related products. These related products require a high maintenance effort which could be avoided by building a platform for a Product Family (PF) from existing software assets. To identify assets from related products which can be used as basis for a PF, retrospective software evolution analysis can help to point out artifacts which exhibit a strong change dependency.

Most of the proposed mining approaches such as Zimmermann et al. [14] for mining the change history or Collberg et al. [3] for visualizing a systems evolution are justified to analyze data from a single source and would therefore require adaption to support data from multiple product variants. Analyzing a single product vari-

ant implies a strict order on historical information such as check-ins into the source code repositories. In contrast to this, multiple product variants can be roughly characterized through arbitrary and asynchronous release dates, unanticipated information flow between variants, different development goals and requirements. Given these constraints, with our *PfEvo* approach we address the problem of handling multiple, *asynchronously* maintained version control systems to identify change dependencies through “alien” source code.

Artifacts with a strong change dependency often have architectural dependencies as research by Briand et al. has shown [1, 2]. Another prevalent reason is duplicated code through *copy’n paste*. For the analysis of such change dependencies it would be beneficial if existing approaches and techniques can be adapted and reused to study their impact onto the module structure.

As a result, an expert may draw conclusions about commonalities and dependencies between source code modules based on results obtained from the change history analysis. Then, the identified software artifacts can be used as foundation for building a platform for a product family. A Representative of such a family of related products is the BSD operating system with its variants and derivations such as *MacOS X*, *SunOS*, or *NetBSD*.

In this paper we (1) apply and extend our approach [5] for extracting change history information and generating a release history database; (2) compare product variants on quantitative level for a coarse assessment of the historical development and assessment of the repository information for further research; and (3) apply our approach for the visualization of change dependencies [4].

The remainder of this paper is organized as follows: Section 2 presents our approach for studying product family evolution. In Section 3 we present our case study about three BSD variants. Section 4 presents related work and Section 5 draws our conclusions and indicates future work.

2. AN APPROACH TO STUDY PRODUCT FAMILY EVOLUTION

Our *PfEvo* approach is an extension of existing techniques for the study of the evolution of a single software system and comprises the visualization of different aspects of the evolution of a software system. Besides some quantitative aspects such as the number of artifacts, check-in transactions, etc., these systems can be compared qualitatively as well. These quality aspects can be related to the type and extent of information flow between different systems, the impact of other related products on a single product, or hot-spots in the evolution of a single system with respect to information from other product variants.

To answer the research question of source code propagation within

*The work described in this paper was supported in part by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), the Austrian Industrial Research Promotion Fund (FFF), the European Commission in terms of the EUREKA 2023/ITEA project FAMILIES (<http://www.infosys.tuwien.ac.at/Cafe/>) and the European Software Foundation under grant number 417.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR’05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

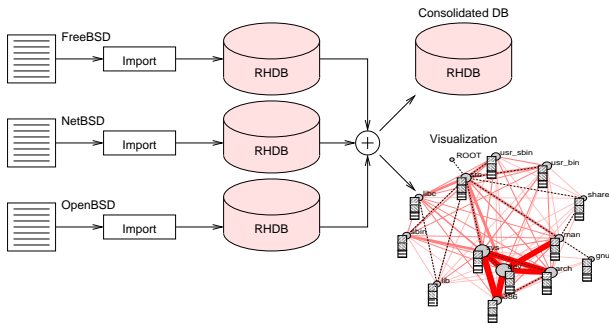


Figure 1: Process outline of *PfEvo*: results are a consolidated RHDB and visualizations

a product family we have adopted our earlier approach for building a release history [5] and visualization of evolutionary information of large-scale software [4] and propose the process depicted in Figure 1. Since all data sources must undergo the same pre-processing steps—log file extraction, import into Release History Database (RHDB), detection of change couplings—we use separate databases to store the results. For subsequent analysis transactional data from the separate databases are filtered and merged into a new *consolidated* database which is better suited for queries spanning multiple product variants. Currently we use modified variants of existing queries to gather data from the three product databases to compare them on a quantitative level. Another approach to compare system characteristics is by visually comparing graphs describing a systems history. We use a module graph indicating the impact of change dependency and their distribution with respect to different product variants onto the module structure of a single system.

In previous studies it was possible to use the release dates of the system under study as input for time scale information. Since the BSD variants are developed independently, an artificial, common time scale has to be created. This ensures comparability of the different system histories. Disadvantageous is that is not possible to examine and compare the processes between the release dates, since the release intervals of the different product variants are crosscut at arbitrary points. Since our requirement is the visualization of the resulting data-sets, we use a sub-sampling interval of one month.

To detect and relate information flow between BSD variants we decided to use lexical search in the change logs to find hints for information flow from other systems into the system under inspection. Alternatives to a pure lexical search are clone detection in source code, comparison of the structure of changes, or advanced indexing and text-analysis techniques.

3. CASE STUDY

For this case study we decided to use derivatives of the Berkley System Distribution also known as BSD Unix. The selected three variants—*FreeBSD*, *NetBSD*, and *OpenBSD*—of BSD are large software systems consisting of an operating system kernel and a number of external programs such as *ls*, *passwd*, the GNU Compiler Collection (GCC), or the X windows system. These variants have between 4800 for the *OpenBSD* variant and 8000 directories for the *NetBSD* variant. The number of files varies between 30,000 (*FreeBSD*) and about 68,000 (*NetBSD*). They are long lived, actively maintained software systems representing about 8.5GB of data stored in three different repositories. Furthermore, release information is available as CVS [7] data for all three variants with

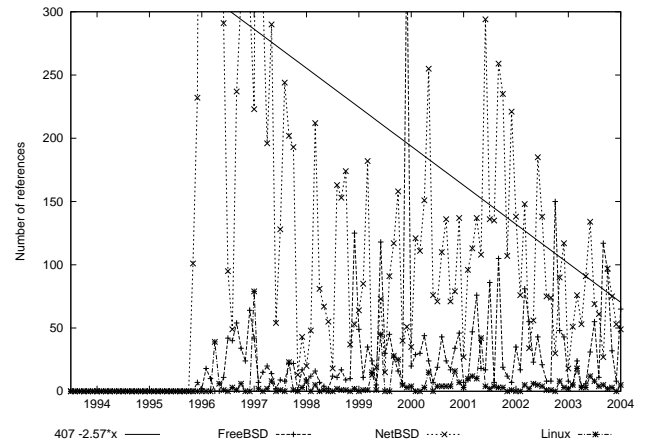


Figure 2: Number of references to keywords *FreeBSD*, *NetBSD*, and *Linux* found in *OpenBSD* change logs

direct access to the current repositories. The systems itself possess different characteristics which can be described as follows: The *FreeBSD*¹ projects aims to be more user application centric and thus it can be seen as desktop OS rather than server platform. Its first release was in December 1993. *NetBSD*² is targeted onto portability and supports more than 10 different CPU types with together more than 50 different hardware platforms. Among them are exotic platforms such as *Acorn*, *Amiga*, *Atari* or *VAX*. Its first release was in October 1994. As representative of a server platform the aim of the *OpenBSD*³ project lies on security and the integration of cryptography. Its first release was in October 1996. While *NetBSD* and *FreeBSD* were directly derived from the 4.3BSD branch, *OpenBSD* was derived from the *NetBSD* branch in October 1995.

3.1 Quantitative comparison

First we give a quantitative comparison of the number of artifacts which are common for the different systems. To determine the number of common C files in the different RHDBs we use multi-database SQL queries. Table 1 shows the result for the different variants. While column “*all modules*” indicates the total number of common files found, column “*src/sys/ only*” indicates the common files within this particular subtree. Interesting is the high number of artifacts which are common in *NetBSD* and *OpenBSD*. This can be explained by the fact that *OpenBSD* was derived from *NetBSD* as mentioned previously.

Table 1: Common files in different BSD variants

Variant	Variant	all modules	src/sys/ only
<i>FreeBSD</i>	<i>NetBSD</i>	3810	1333
<i>FreeBSD</i>	<i>OpenBSD</i>	3839	1079
<i>NetBSD</i>	<i>OpenBSD</i>	6969	6847

3.2 Change report text analysis

As substitution for a detailed text and code clone analysis, we use keywords which were frequently used by the program authors and recorded in change reports. As useful keywords we identified *freebsd*, *netbsd*, *openbsd*, and interestingly *linux*.

Table 2 lists the number of referenced artifacts between product variants based on a lexical search for the chosen keywords in the

¹<http://www.freebsd.org/> [31 December 2004]

²<http://www.netbsd.org/> [31 December 2004]

³<http://www.openbsd.org/> [31 December 2004]

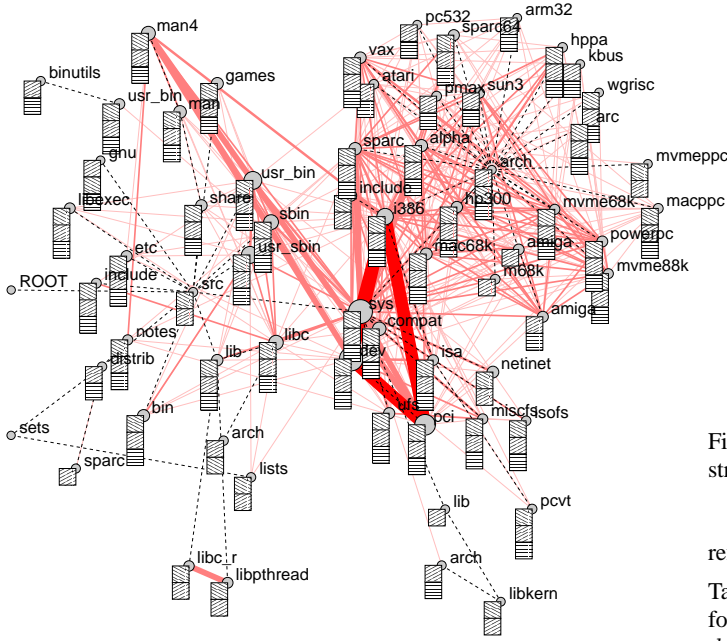


Figure 3: Change coupling between modules of the source code structure of the OpenBSD system with emphasize on the module structure

Table 2: Information flow between variants of the BSD systems based on lexical search

Variant	Keyword	all revisions	revision > 1.1
<i>FreeBSD</i>	<i>netbsd</i>	5131	3577
	<i>openbsd</i>	2729	1353
	<i>linux</i>	1791	1387
<i>NetBSD</i>	<i>freebsd</i>	2852	2186
	<i>openbsd</i>	2679	2224
	<i>linux</i>	1547	1125
<i>OpenBSD</i>	<i>freebsd</i>	2406	1933
	<i>netbsd</i>	16802	7423
	<i>linux</i>	775	463

change logs. Column one lists the name of the product variant used to retrieve the change logs and column two the respective keyword. Column three entitled “all revisions” lists the number of distinct artifacts found in the RHDB having change logs with the specified keyword. Column four titled “revision > 1.1” lists the number of distinct artifacts found in the RHDB having change logs with the specified keyword and not having a revision number of “1.1” (which denotes the initial revision). The significant difference between the values in column three and four can be interpreted in such a way, that a larger number of files were imported from other systems and further maintenance is decoupled from the originating version.

3.3 Reference distribution

During the lexical search for the given keywords we recorded in total 12,540 change logs for *FreeBSD*, 9,468 for *NetBSD*, and 20,906 for *OpenBSD*. Based on these results, Figure 2 depicts the distribution of references with respect to the observation period. Visually the histogram for *OpenBSD* suggest a strong decreasing trend in the information flow from other platforms into the *OpenBSD* source code repository.

To underpin the visual perception of the trends we use linear regression analysis to find the dependency between the number of

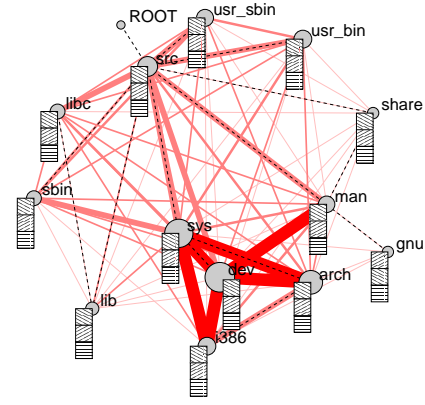


Figure 4: Change coupling between modules of the source code structure of the OpenBSD system

references and time-scale intervals.

Table 3: Linear regression for referenced keywords as $y = d + kx$ for the whole observation period, for the years 1995–2001 ($y = d_{1,2} + k_{1,2}x$) and the years 2001–2004 ($y = d_{3,3} + k_{3,3}x$)


Variant	d	k	$d_{1,2}$	$k_{1,2}$	$d_{3,3}$	$k_{3,3}$
<i>FreeBSD</i>	22.7	0.897	-2.67	1.46	387	-2.35
<i>NetBSD</i>	-22.7	1.28	-15.7	1.14	-21.3	1.31
<i>OpenBSD</i>	407	-2.57	543	-4.90	668	-4.48

To test the development of the references over the given observation period we computed the values for the whole period and two sub-intervals: the first interval accounts for about 2/3 (variables $k_{1,2}$ and $d_{1,2}$) of the observation period which corresponds to the years 1995–2001; the second interval accounts for about the last 1/3 (variables $k_{3,3}$ and $d_{3,3}$) of the observation period which represents the last 36 months of the development history (years 2001–2004).

Table 3 shows the results for the three variants indicating a strong increasing trend for *FreeBSD* and *NetBSD* ($k > 0$ for both variants over the whole observation period). For *FreeBSD* this trend reverses for the last 36 months ($k_{3,3} < 0$). The low number of total change logs found for *NetBSD* and the positive trend in the change dependency of *NetBSD* suggest that large amounts of source code are still derived from the other OS variants. This perception is also supported by Table 2 since *NetBSD* has the highest ratio between the two counted categories “revisions > 1.1” and “all revisions”. In contrast, *OpenBSD* exhibits a decreasing trend in both sub-intervals and the whole observation period starting from a high level (straight line in Figure 2).

In the next sections we provide a more detailed look onto the change relationships with respect to different products.

3.4 Change impact analysis

To show the impact of changes onto the module structure with respect to foreign source code we selected *OpenBSD* for a closer inspection since we counted here the most keywords referencing other OS (see Table 2). The relevant artifacts were identified through lexical search as previously described. Based on the search results and the change log data the impact of change dependencies on the module structure is evaluated. The result of this step is depicted in the Figures 3 and 4. It shows the module structure together with change dependencies derived from the change log data. While filled circles indicate the nodes of the directory tree, shaded boxes indicate different product variants. We use  as glyph for *FreeBSD*,

▣ for *NetBSD*, and ▢ is used for *Linux*. The approach for generating the layout for change dependencies information is based on Multi Dimensional Scaling (MDS) [9] and has been used by our group to visualize to impact of problem report data onto the module structure of large software [4].

To avoid cluttering of the figure with the several hundred modules of the source code package, we shifted relevant information from lower level nodes of the nested graph structure towards the root node until a predefined threshold criterion—at least 64 references through change couplings per node—is met. The node sizes indicate the number of references found for each node and its subtrees.

While dashed lines indicate the directory structure of the source package, solid gray and black lines (pink and red on color displays) indicate the logical coupling between different parts of the system.

Figure 3 shows the dependencies between modules with emphasis on the module structure (149 nodes). The distribution of the glyphs for *FreeBSD*, *NetBSD*, and *Linux* indicates a significant impact—though decreasing trend—of the other OS variants onto the development of *OpenBSD*. Only very few modules such as *libpthread*—POSIX threads are not part of the *Linux* kernel sources—or *lists* (on the bottom left in Figure 3) are not infected by “*Linux* virus”. This wide distribution of *Linux* related change dependencies is a surprising result since we did not expect such a distribution after the quantitative analysis. Interesting as well is that change dependencies occur mainly within the *src/sys* sub-structure which represents the kernel related source code parts.

After filtering of less relevant modules and shifting the information to higher level modules in the hierarchy we obtain the graph depicted in Figure 4 (14 nodes). Here, the graph layout respects the strength of coupling relationships—the stronger the coupling, the closer the nodes—between the different modules. This more comprehensible and less cluttered picture of couplings highlights the dependencies of the documentation in *src/share/man*, the system administration programs in *src/sbin*, user application programs such as *ls* in *src/usr_bin* and *src/usr_sbin* from the OS kernel related files underneath *src/sys*. Interesting to see is also the strong coupling via “foreign” source code changes between *src/sys/arch/i386* and *src/sys/dev* since this coupling spans across the module hierarchy.

Since the size of the nodes indicates the number of relevant change entries found, we can conclude that the strongest impact of change coupling was on *src/sys*, *src/sys/dev*, *src/sys/arch*, and *src/sys/arch/i386*. Table 4 lists an excerpt of the topmost referenced artifacts which suggests a high information exchange with other software systems.

Table 4: Topmost referenced files with one of the given keywords in the change logs of *OpenBSD*

Keyword	Count	Path
freebsd	59	src/sys/dev/pci/files.pci
.	52	src/sys/dev/pci/pciide.c
.	52	src/sys/dev/pci/pcidevs
netbsd	45	src/sys/arch/i386/i386/machdep.c
.	43	src/sys/dev/pci/pciide.c
.	39	src/sys/conf/files
linux	14	src/sys/compat/linux/linux_socket.c
.	14	src/sys/compat/linux/syscalls.master
.	5	src/sys/dev/ic/if_wireg.h

An example for the propagation of commonly required feature is the introduction of the PCI bus. Since this device type was not widely available at the time of the *OpenBSD* fork in 1996, support had to be added later requiring several separate changes as Table 4

suggests. Another interesting aspect is the relationship with *Linux*. The listing of *if_wireg.h* suggests that specific information about WLAN adapters are obtained from *Linux* as well.

3.5 Detailed change analysis

Since the three BSD variants originate from the same UNIX branch, it is to expect that also a number of source code changes exhibit the same or at least similar structure. For a manual verification we randomly selected one file which is available in all three variants. For this file—*ufs_quota.c* from the *src/sys/ufs/ufs/* directory—we manually inspected the revision history for significant changes.

One significant change was the modification of a function call in the *FreeBSD* version of *ufs_quota.c* on 1994-10-06 (revision 1.2 → 1.3) resulting in eight modified source lines. The *diff*-snippet—depicted below—for the affected source code revision shows a single change of a source line. The first line indicates the removed code, whereas the third one shows the replacement code. The three dashes in-between indicate a delimiter line.

```
< sleep(( caddr_t)dq, PINOD+2);
---
> (void) tsleep (( caddr_t)dq, PINOD+2, "dqsync", 0);
```

In the change log we found the following comment, which indicates the reason for the source code modification: “*Use tsleep() rather than sleep so that 'ps' is more informative about the wait.*”

The same modification in the *NetBSD* version has been applied on 2000-05-27 which is six years later than the original modification (revision 1.16 → 1.17) and in *OpenBSD* more than eight years later on 2001-11-21 (revision 1.7 → 1.8)—though without the *(caddr_t)* type cast listed in the preceding code snippet. The *diff*-snippet below depicts the modification.

```
< sleep(( caddr_t)dq, PINOD+2);
---
> (void) tsleep (dq, PINOD+2, "dqsync", 0);
```

In the *NetBSD* variant of the change log the comment is less informative: “*sleep() -> tsleep()*”. While in *NetBSD* this change still produces similar results when building the revision deltas via *diff*, in *OpenBSD* the change was part of a larger source code modification consisting of 380 added and 161 deleted source lines (CVS does not identify modified lines, instead every modified line accounts for one added and one deleted line). Analogues to the given example, many changes can be found with varying degree of similarity making it difficult to track source code propagation.

3.6 Discussion

During experiments with our RHDB we noticed some shortcomings which have to be resolved prior to a thorough analysis of the different product variants. First, through moving and renaming files in the CVS repository by the developers of the software systems, the historical information is segmented. Thus related segments have to be identified and concatenated to describe a continuous historical time-line of an artifacts history. Second, as result of the import process artifacts which have identical file names are assigned different IDs in the RHDB. This may negatively effect multi-database queries for comparison of artifacts since artifacts with common origins have to be identified for every evaluation of a database query. This mapping of IDs will be ideally stored in the consolidated part of the RHDB as indicated in Figure 1.

From the software evolution analysis point of view, BSD represents an interesting software system which opens a wide field for further analysis. Since detailed information about the source code

is available it would be beneficial to apply a tool for code clone detection such as [8] proposed by Kamiya et al. To improve the results of the lexical search we currently explore the application of techniques related to Latent Semantic Indexing (LSI) [10].

4. RELATED WORK

Within the EU projects ARES, ESAPS, CAFE, and Families much work has been done in areas such as the identification of assets for product family architectures, evolution and testing of existing product families, or architectural models for product families (Van der Linden [12]). More related with our work with respect to product family evolution is the approach presented by Riva and Del Rosso in [11]. They investigated the evolution of a family platform and describe approaches which enable assessment and reconstruction of architectures. In contrast to their work, we investigate the evolution of different variants to identify candidates for building a family platform.

In [6] Gall, Hajek and Jazayeri examined the structure of a large *Telecommunications Switching Software* (TSS) over more than 20 releases to identify logical coupling between system and subsystems. This coupling is used in further processing steps to reveal evolutionary aspects such as hot-spots. For the detection and visualization of evolutionary hot-spots we have developed a methodology which relates software feature and release history information [4]. In this paper we used information from the release history with respect to different keywords instead of feature data. This information was reflected onto the module structure of the source code and visualized to generate the high level views of a software system. Independent from our research work Yamamoto et al. investigated variants of the BSD system for similarities as well [13]. They mainly use *CCFinder* by Kamiya et al. [8] to compute similarity metrics of the source code. In contrast to our work, their aim lies on the overall similarities between different products, rather than the type, amount and distribution of information flow between the variants.

5. CONCLUSIONS

Retrospective analysis of variants of related products opens interesting perspectives on the evolution of large software systems. With minimal changes and additions to existing tools it is already possible to recover the information flow between the different variants and evolutionary hot-spots with respect to the module structure. Through the application of a lexical search in the change logs we were able to reveal the increasing information flow of two variants of the systems. For the third system we found a decreasing flow starting from a very high level. For one selected system we applied an adapted method which generates high-level views of the module structure of a system with respect to their coupling and information flow from other product variants. To support these findings about the information flow we performed detailed change analysis of a randomly selected file. Interesting results are: the wide distribution of *Linux* related change dependencies in the source code; the strong change coupling within the subtree of *src/sys*; and the propagation of source code taking several years.

For future work we plan the application of a code clone detection process to identify related modifications. An analysis can reveal the degree and frequency of how tight product variants are coupled. Another interesting area for future work is the detailed analysis of change log information for commonalities. Since change logs can provide additional hints about a particular modification, they provide relevant information which enables the identification of a modifications origin.

6. REFERENCES

- [1] BRIAND, L., DEVANBU, P., AND MELO, W. An investigation into coupling measures for C++. In *Proceedings of the 19th international conference on Software engineering* (1997), ACM Press, pp. 412–421.
- [2] BRIAND, L. C., DALY, J. W., AND WÜST, J. K. A Unified Framework for Coupling Measurement in Object-Oriented Systems. *IEEE Transactions on Software Engineering* 25, 1 (1999), 91–121.
- [3] COLLBERG, C., KOBOUROV, S., NAGRA, J., PITTS, J., AND WAMPLER, K. A system for graph-based visualization of the evolution of software. In *Proceedings of the 2003 ACM symposium on Software visualization* (2003), ACM Press, pp. 77–ff.
- [4] FISCHER, M., AND GALL, H. Visualizing Feature Evolution of Large-Scale Software based on Problem and Modification Report Data. *Journal of Software Maintenance and Evolution* 16, 6 (November/December 2004), 385–403.
- [5] FISCHER, M., PINZGER, M., AND GALL, H. Populating a Release History Database from Version Control and Bug Tracking Systems. In *Proceedings International Conference on Software Maintenance (ICSM'03)* (September 2003), pp. 23–32.
- [6] GALL, H., HAJEK, K., AND JAZAYERI, M. Detection of Logical Coupling Based on Product Release History. In *Proceedings International Conference on Software Maintenance* (March 1998), IEEE Computer Society Press, pp. 190–198.
- [7] GRUNE, D., BERLINER, B., POLK, J., KLINGMON, J., AND CEDERQVIST, P. *Version Management with CVS*, 1992. <http://www.cvshome.org/docs/manual/> [5 April 2004].
- [8] KAMIYA, T., KUSUMOTO, S., AND INOUE, K. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
- [9] KRUSKAL, J. B., AND WISH, M. Multidimensional Scaling. *Quantitative Applications in the Social Sciences* 11 (1978).
- [10] LETSCHE, T. A., AND BERRY, M. W. Large-scale information retrieval with latent semantic indexing. *Information Sciences* 100 (August 1997), 105–137.
- [11] RIVA, C., AND DEL ROSSO, C. Experiences with software product family evolution. In *Proceedings Sixth International Workshop on Principles of Software Evolution (IWPSE'03)* (September 2003), IEEE Computer Society Press, pp. 161–169.
- [12] VAN DER LINDEN, F., Ed. *Software Product-Family Engineering: 5th International Workshop, PFE 2003, Siena, Italy*, vol. 3014 of *Lecture Notes in Computer Science*. Springer-Verlag Heidelberg, 2004.
- [13] YAMAMOTO, T., MATSUSHITA, M., KAMIYA, T., AND INOUE, K. Measuring Similarity of Large Software Systems Based on Source Code Correspondence. In *Proceedings of the 6th International Conference on Product Focused Software Process Improvement (PROFES'05)* (June 2005). to appear.
- [14] ZIMMERMANN, T., WEISSGERBER, P., DIEHL, S., AND ZELLER, A. Mining Version Histories to Guide Software Changes. In *Proceedings 26th International Conference on Software Engineering (ICSE)* (May 2004), ACM Press, pp. 563–572.

Using a Clone Genealogy Extractor for Understanding and Supporting Evolution of Code Clones

Miryung Kim and David Notkin
Computer Science & Engineering
University of Washington
Seattle, USA.

{miryung,notkin}@cs.washington.edu

ABSTRACT

Programmers often create similar code snippets or reuse existing code snippets by copying and pasting. Code clones—syntactically and semantically similar code snippets—can cause problems during software maintenance because programmers may need to locate code clones and change them consistently. In this work, we investigate (1) how code clones evolve, (2) how many code clones impose maintenance challenges, and (3) what kind of tool or engineering process would be useful for maintaining code clones.

Based on a formal definition of clone evolution, we built a *clone genealogy tool* that automatically extracts the history of code clones from a source code repository (CVS). Our clone genealogy tool enables several analyses that reveal evolutionary characteristics of code clones. Our initial results suggest that aggressive refactoring may not be the best solution for all code clones; thus, we propose alternative tool solutions that assist in maintaining code clones using clone genealogy information.

1. INTRODUCTION

We define code clones as syntactically similar code snippets that resemble one another semantically, which are often created by copy and paste¹. Code clones may induce problems during software evolution. In particular, when a change is made to one element in a group of clones, a programmer must generally make consistent changes to the other elements in the group. Forgetting to update one or more elements may leave outdated code, a potential bug. In other words, code clones impose cognitive overhead because programmers must remember cloning dependencies to apply the same change consistently.

¹Code clones have no consistent definition in the literature, but most consider them to be identical or near identical fragments of source code [5, 11].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MSR'05, May 17, 2005, Saint Louis, Missouri, USA Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

Software engineering researchers have addressed problems surrounding code clones in many ways. First, several kinds of clone detectors have been built. Clone detectors [3, 4, 6, 7, 10, 11, 13, 14, 15, 17] identify similar code snippets automatically by comparing the internal representation of source code (e.g., a parametrized token string [3, 11], AST [6, 17], or PDG [13, 14]). Second, a few programming methodologists have educated programmers about how to avoid or remove code clones. Fowler [8] argues that code duplicates are bad smells of poor design and programmers should aggressively use refactoring techniques. The Extreme Programming (XP) community has integrated frequent refactoring as a part of development process. Nickell and Smith [17] argue that fewer code clones are found in XP process software, claiming that the XP process improves software quality. We believe that these previous research efforts are based on the following assumptions: (1) code clones indicate poor software quality, (2) aggressive refactoring would solve the problem of code clones, and (3) if programmers can locate code clones, they can improve the quality of the code base.

Based on our study of copy and paste programming practices [12], we became skeptical about the validity of some of these assumptions. We found that even skilled programmers sometimes had no choice but to create and manage code clones. Subjects copied and pasted code snippets to reuse the logic that is often not separable given the limitations of Java programming language. Our subjects often discovered an appropriate level of abstraction as they copied, pasted, and modified code; some subjects postponed refactoring until their design decisions become stable.

We hypothesize that programmers create and maintain code clones for two major reasons: (1) as programmers deal with volatile design decisions while they add new features or extend existing features, they prefer not to commit to a particular level of abstraction too quickly, and (2) programmers cannot refactor many code clones because of the primary design decisions in the software and the limitations of programming languages. To test our hypothesis, analyzed how code clones have evolved in two Java open source projects. We formally defined a model of clone evolution and then built an analysis tool that automatically extracts the history of code clones from a set of program versions. Using this tool, we investigated frequent clone evolution patterns.

Our initial result confirms some conventional wisdom about

code clones and also suggests that aggressive refactoring may not benefit many, perhaps not most, clones:

- Clones are not dormant and programmers often face the challenge of updating clones consistently. In fact, 32% ~ 38% of code clones changed consistently with their counterparts at least once in their history.
- Aggressive refactoring may not be the best solution; 64% ~ 68% of code clones were not factorable unless programmers sacrifice primary design decisions or make non-local changes.

Because programmers may not be able to remove or avoid all code clones, we propose clone maintenance tools as effective alternatives and supplements to refactoring. The proposed software engineering tools employ clone genealogy information—the history of code clones—to assist in maintaining clones.

The rest of this paper is organized as follows. Section 2 formally defines the model of clone evolution, which serves the basis of a clone genealogy extractor described in Section 3. Section 4 presents analysis of clone evolution patterns and discusses implications of our initial result. Section 5 proposes software engineering tools that use clone genealogy information. Section 6 summarizes and concludes our study.

2. MODEL OF CLONE EVOLUTION

We formally defined the model of clone evolution to reason how clones change regardless of underlying clone detection technologies.

The basic unit of our analysis is a *Code Snippet* which has two attributes, *Text* and *Location*. *Text* is an internal representation of code that a clone detector uses to compare code snippets. For example, when using CCFinder [11], a parametrized token sequence is *Text*, whereas when using CloneDr [6], *Text* is an isomorphic AST. A *Location* is used to track code snippets across multiple versions of a program; thus, every code snippet in a particular version of a program has a unique *Location*. A *Clone Group* is a set of code snippets with identical *Text*.

A *Cloning Relationship* exists between an old clone group and a new clone group in two consecutive versions if and only if the similarity between the clone groups is over a similarity threshold sim_{th} . An *Evolution Pattern* is defined between an old clone group *OG* in the version k and a new clone group *NG* in the version $k + 1$, where *NG* and *OG* have a *Cloning Relationship*.

- *Same*: all code snippets in *NG* did not change from *OG*.
- *Add*: at least one code snippet in *NG* is a newly added one. For example, programmers added a new code snippet to *NG* by copying an old code snippet in *OG*.
- *Subtract*: at least one code snippet in *OG* does not appear in *NG*. For example, programmers removed one clone snippet.

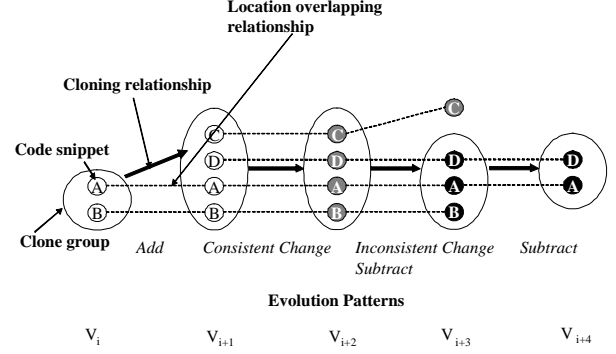


Figure 1: Example Clone Lineage

- *Consistent Change*: all code snippets in *OG* have changed consistently; thus they belong to *NG* together. For example, programmers applied the same change consistently to all code clones in *OG*.
- *Inconsistent Change*: at least one code snippet in *OG* changed inconsistently; thus it does not belong to *NG* anymore. For example, a programmer forgot to change one code snippet in *OG*.

Clone Lineage is a directed acyclic graph that describes the evolution history of a sink node (clone group). In a clone lineage, a clone group (node) in the version k is connected by an *Evolution Pattern* (directed edge) from a clone group in the version $k - 1$. For example, Figure 1 shows a clone lineage including *Add*, *Subtract*, *Consistent Change*, and *Inconsistent Change*.

Clone Genealogy is a set of clone lineages that have originated from the same clone group. A clone genealogy is a connected component where every clone group (node) is connected by at least one evolution pattern (edge). A clone genealogy approximates how programmers create, propagate, and evolve code clones by copying, pasting, and modifying code. Our model is written in the *Alloy* modeling language [2] and is available at [1].

3. CLONE GENEALOGY EXTRACTOR

Based on the clone evolution model in Section 2, we built a tool that automatically extracts clone genealogies over a project’s lifetime.

Given the source code repository (CVS) of a project, our tool prepares versions of the project in chronological order. We used Kenyon’s front-end to identify CVS transactions and check out the source code that corresponds to each transaction time [9].

Given multiple versions of a program, our tool identifies clone groups in each version using a clone detector. Our tool is designed to plug in different types of a clone detector. Currently we use CCFinder [11], a state-of-the-art clone detector, which compares a parametrized token string of code to detect code clones. Next, it finds cloning relationships between all consecutive versions using the same clone detector. Then, it separates each connected component of

Table 1: Clone Genealogies in *carol* and *dnsjava*

Number of Genealogies	<i>carol</i>	<i>dnsjava</i>
Total	122	95
False Positive	13	19
Locally Unfactorable	70 (64%)	52 (68%)
Consistent Changed	41 (38%)	24 (32%)

cloning relationships found over the project’s life time and labels evolution patterns in each connected component. This connected component is called a clone genealogy.

4. CLONE EVOLUTION ANALYSIS

To understand how clones evolve, we extracted clone genealogies from two Java open source projects, *carol* and *dnsjava*, and studied evolution patterns shown in the genealogies. *Carol* is a library that allows clients to use different RMI implementations and it has grown from 7878 lines of code (LOC) to 23731 LOC from August 2002 to October 2004 (carol.objectweb.org). *Dnsjava* is a implementation of DNS in Java, and it has grown from 5038 LOC to 20752 LOC from March 1999 to June 2004 (www.dnsjava.org).

In our analysis, we chose 37 versions out of 164 check-ins of *carol* and 39 versions out of 47 releases of *dnsjava* that resulted in changes of LOCC (the total number of lines of code clones).

We set the minimum token length of CCFinder to be 30 tokens because many programmers do not consider short clones as real clones. We set the similarity threshold sim_{th} for cloning relationships to be 0.3 because empirically we found that sim_{th} 0.3 does not underestimate or overestimate the size or the length of genealogies.

CCFinder occasionally detects false positive clones that are similar only in a token sequence, although common sense says that they are not clones. If clones comprise only a **syntactic template**, we consider the clones as false positives. In our previous study of copy and paste programming practices [12], we defined “a syntactic template” as a template of repeated code appearing in a row because a programmer often copies and pastes a code fragment when writing a series of syntactically similar code fragments. For example, a programmer often copies a field declaration statement when writing a block of field declaration, an invocation statement when writing a static initializer, or a case statement to write a series of case statements in a switch-case block. We manually removed 13 out of 122 genealogies in *carol* and 19 out of 95 genealogies in *dnsjava* because they comprise only a syntactic template.

Using the clone genealogy information, we intend to examine two research questions: (1) how serious is the problem of code clones? and (2) whether would refactoring benefit most code clones? For each research question, we describe our analysis approach, initial result, and implication of our result.

Q: How many code clones impose maintenance challenges?

If code clones stay dormant, these unchanging clones might not pose challenges during software evolution. But consistently changing clones would reduce productivity because programmers often need to locate code clones and apply the equivalent change to the code clones.

We define that a clone genealogy includes a consistently changing pattern if and only if all lineages in the clone genealogy include at least one “consistent change” pattern. Our definition is very conservative because, if one lineage in the genealogy does not include a consistent change pattern, the genealogy is considered not to have a consistent change pattern. We measured the number of genealogies with a consistent change pattern. Out of 109 genealogies in *carol*, 41 genealogies (38%) include a consistently changing pattern. Out of 76 genealogies in *dnsjava*, 24 genealogies (32%) include a consistently changing pattern (see Table 1). This result implies that programmers had faced the challenge of updating clones consistently with other elements in the same clone group.

Q: Would aggressive refactoring be the best solution for maintaining code clones?

Finding a new abstraction to remove code duplication has been a core approach for effective programming. There has been a broad assumption that code clones are inherently bad because code clones defy the principle of abstraction. To examine the validity of this assumption, we set up two hypotheses.

Hypothesis 1: Many code clones are not locally factorable given the primary design decisions of software and the limitations of programming languages.

In our analysis, we define that a clone group is “locally factorable” if a programmer can remove duplication with standard refactoring techniques, such as *pull up a method*, *extract a method*, *remove a method*, *replace conditional with polymorphism*, etc [8]. On the other hand, if a programmer must make non-local changes in the design or modify publicized interfaces to remove duplication of if a programmer cannot remove duplication due to programming language limitations, we consider that the clone group is not locally factorable. Our previous work describes a taxonomy of locally unfactorable code clones that are often created by copy and paste [12]. A clone lineage is locally unfactorable if the latest clone group (a sink node of the lineage) is locally unfactorable. We define that a clone genealogy is locally unfactorable if and only if all clone lineages in the genealogy are locally unfactorable. A locally unfactorable genealogy means that a programmer cannot discontinue any of its clone lineages by refactoring.

In the two subject programs, we inspected all clone lineages and manually labeled them as “locally factorable” or “locally unfactorable.” Then, we measured how many clone genealogies are locally unfactorable. 70 genealogies (64%) in *carol* and 52 genealogies (68%) in *dnsjava* comprise locally unfactorable clone groups; this result indicates that popu-

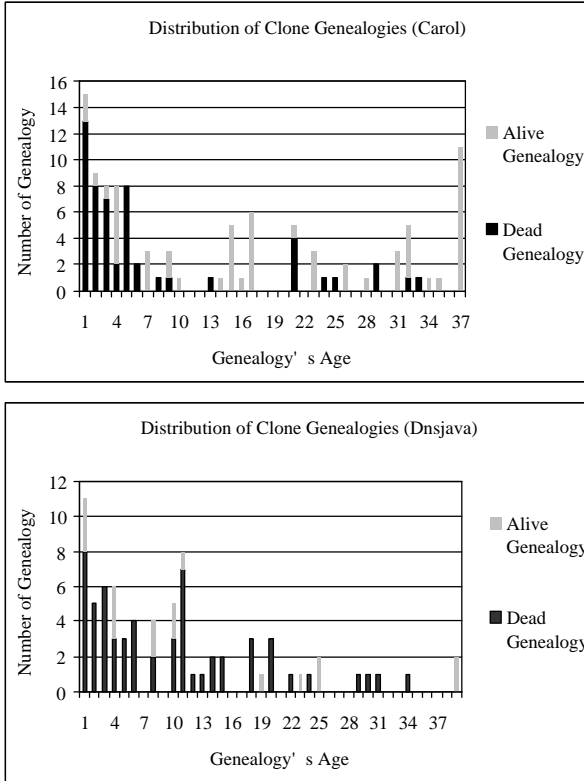


Figure 2: Many clone genealogies disappear after a relatively short time.

lar refactoring techniques would not benefit most clones. In fact, we found that many long-lived, consistently changing clones are locally unfactorable. Out of 37 genealogies that lasted more than 20 versions in carol, 19 of them include both consistent change patterns and locally unfactorable clones. Out of 11 genealogies that lasted more than 20 versions in dnsjava, 3 of them include both consistent change patterns and locally unfactorable clones.

Hypothesis 2: Programmers prefer not to commit to a particular abstraction immediately when dealing with volatile design decisions.

A dead genealogy means that all of its clone lineages were discontinued because the code clones disappeared, diverged, or they were refactored. An alive genealogy means that at least one of its clone lineage is still evolving and the clones have not disappeared yet. Figure 2 shows distribution of dead and alive clone genealogies over their age. The age of a clone genealogy is the number of versions that the genealogy spans. In carol, out of 53 dead genealogies, 42 genealogies disappeared less than 10 versions. In dnsjava, out of 59 dead genealogies, 41 genealogies disappeared less than 10 versions. We believe that programmers created and maintained code clones while they explored new design space, and then later, they removed, diverged, or refactored the code clones as the relevant design decisions became stable. When we manually inspected all dead lineages, we found that 25% (carol) \sim 48% (dnsjava) of them were discontinued because

of divergent changes in the clone group. Programmers would not get the best return on their refactoring investment if the clones are to diverge.

5. CLONE MAINTENANCE TOOLS

Our study result indicates that popular refactoring techniques may not remove most code clones, especially clones that are difficult to maintain. Thus, we propose clone maintenance tools as alternatives and supplements to refactoring. This section lists possible software engineering tools that can be built on top of our clone genealogy extractor.

5.1 Simultaneous Text Editing

Abstraction, isolating code duplication in a programming language unit, provides two advantages during software evolution. First, programmers can locate the duplicated logic in one place. Second, programmers can apply the change only once in the refactored code. Clone detectors automatically locate code clones, resolving the first issue. However, programmers still need to update code clones manually one by one when the same change is required, leaving the second issue unresolved. Simultaneous text editing [16] is a new method for automating repetitive text editing. After describing a set of regions to edit, the user can edit any one record and see equivalent edits applied simultaneously to all other records. We propose simultaneous editing of consistently changing clones. The proposed editor uses clone genealogy information to automatically identify code snippets that are likely to change consistently in the future. Then, when a programmer edits one of the clones, upon request, the equivalent edit is made to other clones simultaneously. This proposed editor not only provides the same advantages as abstraction but also allows divergent changes flexibly.

5.2 Cloning Related Bug Detection

Many programming errors occur when programmers create and update code clones. For example, Li *et al.*, found that a few errors in Linux were created when a programmer copied code but failed to rename identifiers correctly in the pasted code [15]. As another example, Ying *et al.*, also reported a cloning related bug in Mozilla [18]; a web browser using gtk UI toolkit and the version using xlib UI toolkit were code clones. When a developer changed the version using gtk but did not update the version using xlib, this missed update led to a serious defect, called “huge font crashes X Windows.” If a clone genealogy extractor finds clones that have changed similarly before but change inconsistently later, this information may strongly suggest a bug.

Programmers often copy and paste to reuse existing code snippets. If the copied code contains a bug, this bug can be propagated to many places via copy and paste. In Mozilla, we found that a buggy code snippet was copied for 12 times [12]. If the copied snippets did not change, a clone detector can locate the buggy snippets automatically. But if the copied code was modified very differently from its template, a clone detector may not be able to find it. Our clone genealogy tool infers how programmers copied, modified, and evolved existing code. By traversing a genealogy graph, we can locate code snippets that have originated from the same buggy code even if they have changed very much.

5.3 Decision Support for Maintaining Code Clones

Clone detectors assist programmers in locating code clones automatically. However, even if programmers can find all clones, they may not know which of them should be updated together when the clones change. The history of code clones may help programmers to make informed decisions about how to manage code clone. For example, if a set of clone snippets have changed consistently in the past, they might evolve similarly in the future as well. Programmers can decide what to change together based on the clone history.

We believe that there's a right timing to refactor code clones. If programmers refactor code clones too early, they might not get the best return on their investment because the code clones may diverge. On the other hand, if programmers wait too long before they restructure code, they would get only marginal benefit on their investment. Programmers can decide when to refactor code clones based on clone genealogy information: (1) how old clones are and (2) how clones have changed in the past.

5.4 Locating the Origin of Copied Code

Programmers often copy an example code snippet or a working component and then modify a small part of it. If programmers do not fully understand the logic of the copied code, they cannot adapt the copied code appropriately as the related design changes. Besides, programmers may have copied outdated example code and do not know how to make it up-to-date. In these cases, programmers may want to find the origin of copied code and consult the original author. However, CVS history retains only who checked in the copied code but does not provide who is the original author or when the original code was written. By overlaying authorship on a clone genealogy, programmers would be able to find the origin of frequently copied code.

6. CONCLUSIONS

There has been a broad assumption that code clones are inherently bad because they defy the principle of abstraction. Thus, previous research efforts focused on mainly two areas: automatically detecting code clones and educating programmers how to remove or avoid clones. However, the history of code clones indicates that this assumption may not be necessarily true and that the current refactoring solution may not work for many clones. We propose clone maintenance tools that use clone genealogy information—code clones' history that is automatically extracted from a source code repository.

7. ACKNOWLEDGMENTS

We thank Software Engineering Laboratory at the Osaka University for providing CCFinder and GRASE lab at the University of California, Santa Cruz for providing Kenyon.

8. REFERENCES

- [1] <http://www.cs.washington.edu/homes/miryung/cge>.
- [2] *Micromodels of Software: Lightweight Modelling and Analysis with Alloy*. <http://alloy.mit.edu>, 2004.
- [3] B. S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1992.
- [4] M. Balazinska, E. Merlo, M. Dagenais, B. Laguë, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *WCRE*, pages 98–107, 2000.
- [5] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. Beyond templates: a study of clones in the STL and some general implications. In *ICSE*, 2005.
- [6] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *ICSM*, pages 368–377, 1998.
- [7] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In *ICSM*, pages 109–118, 1999.
- [8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.
- [9] GRASE-Lab. *User Manual: Kenyon*. <http://dforge.cse.ucsc.edu/projects/kenyon>, 2005.
- [10] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *CASCON*.
- [11] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Trans. Software Eng.*, 28(7):654–670, 2002.
- [12] M. Kim, L. Bergman, T. A. Lau, and D. Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *ISESE*, pages 83–92, 2004.
- [13] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. In *SAS*, pages 40–56, 2001.
- [14] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE*, pages 301–309, 2001.
- [15] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI*, pages 289–302, 2004.
- [16] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX Annual Technical Conference, General Track*, pages 161–174, 2001.
- [17] E. Nickell and I. Smith. Extreme programming and software clones. In *the Proceedings of the International Workshop on Software Clones*, 2003.
- [18] A. T. T. Ying, G. C. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Software Eng.*, 30(9):574–586, 2004.

Defect Analysis

When Do Changes Induce Fixes?

(On Fridays.)

Jacek Śliwerski

International Max Planck Research School
Max Planck Institute for Computer Science
Saarbrücken, Germany

sliwers@mpi-sb.mpg.de

Thomas Zimmermann Andreas Zeller

Department of Computer Science
Saarland University
Saarbrücken, Germany

{tz, zeller}@acm.org

ABSTRACT

As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for *fix-inducing changes*—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as BUGZILLA). In a first investigation of the MOZILLA and ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*corrections, version control*; D.2.8 [Metrics]: Complexity measures

General Terms

Management, Measurement

1. INTRODUCTION

When we mine software histories, we frequently do so in order to detect patterns that help us understanding the current state of the system. Unfortunately, not all changes in the past have been beneficial. Any bug database will show a significant fraction of problems that are reported some time after some change has been made.

In this work, we attempt to identify those *changes that caused problems*. The basic idea is as follows:

1. We start with a bug report in the bug database, indicating a *fixed problem*.
2. We extract the associated change from the version archive, thus giving us the *location* of the fix.
3. We determine the *earlier change* at this location that was applied before the bug was reported.

This earlier change is the one that *caused* the later fix. We call such a change *fix-inducing*.

What can one do with fix-inducing changes? Here are some potential applications:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05 May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

Which change properties may lead to problems? We can investigate which properties of a change correlate with inducing fixes, for instance, changes made on a specific day or by a specific group of developers.

How error-prone is my product? We can assign a *metric* to the product—on average, how likely is it that a change induces a later fix?

How can I filter out problematic changes? When extracting the architecture via co-changes from a version archive, there is no need to consider fix-inducing changes, as they get undone later.

Can I improve guidance along related changes? When using co-changes to guide programmers along related changes, we would like to avoid fix-inducing changes in our suggestions.

This paper describes our first experiences with fix-inducing changes. We discuss how to extract data from version and bug archives (Section 2), and how we link bug reports to changes (Section 3). In Section 4, we describe how to identify and locate fix-inducing changes. Section 5 shows the results of our investigation of the MOZILLA and ECLIPSE: It turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied. Sections 6 and 7 close with related and future work.

2. WHAT'S IN OUR ARCHIVES?

For our analysis we need all changes and all fixes of a project. We get this data from *version archives* like CVS and *bug tracking systems* like BUGZILLA.

A CVS archive contains information about changes: Who changed what, when, why, and how? A *change* δ transforms a revision r_1 to a revision r_2 by inserting, deleting, or changing lines. We will later investigate changes on the line level. Several changes $\delta_1, \dots, \delta_n$ form a *transaction* t if they were submitted to CVS by the same developer, at the same time, and with the same log message, i.e., they have been made with the same intention, e.g. to fix a bug or to introduce a new feature. As CVS records only individual changes to files, we group these to transactions with a *sliding time window* approach [12].

A CVS archive also lacks information about the *purpose* of a change: Did it introduce a new feature or did it fix a bug? Although it is possible to identify such reasons solely with log messages [7], we combine both CVS and BUGZILLA for this step because this increases the precision of our approach.

A BUGZILLA database collects bug reports that are submitted by a *reporter* with a *short description* and a *summary*. After a bug has been submitted, it is discussed by developers and users who provide additional *comments* and may create *attachments*. After the

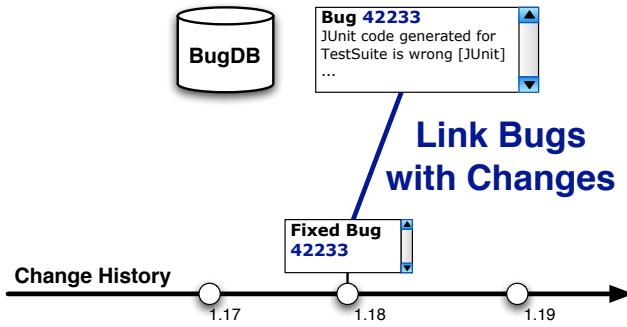


Figure 1: Link transactions to bug reports

bug has been confirmed, it is *assigned* to a developer who is responsible to fix the bug and finally commits her changes to the version control archive. BUGZILLA also captures the *status* of a bug, e.g., UNCONFIRMED, NEW, ASSIGNED, RESOLVED, or CLOSED and the *resolution*, e.g., FIXED, DUPLICATE, or INVALID. Details on the lifecycle of a bug can be found in the BUGZILLA documentation [10, Sections 6.3 and 6.4].

For our analysis, we mirror both CVS and BUGZILLA in a local database. Our mirroring techniques for CVS are described in [12]. To mirror a BUGZILLA database, we use its XML export feature. Additionally, we import attachments and activities directly from the web interface of BUGZILLA. Our local BUGZILLA database schema is similar to the one described in [2].

3. IDENTIFYING FIXES

In order to locate fix-inducing changes, we first need to know whether a change is a fix. A common practice among developers is to include a *bug report number* in the comment whenever they fix a defect associated with it. Čubranić and Murphy [4] as well as Fischer, Pinzger, and Gall [5, 6] exploited this practice to link changes with bugs. Figure 1 sketches the basic idea of this approach.

In our work, we refine these techniques by assigning every link (t, b) between a transaction t and a bug b two independent levels of confidence: a *syntactic* level, inferring links from a CVS log to a bug report, and a *semantic* level, validating a link via the bug report data. These levels are later used to decide which links shall be taken into account in our experiments.

3.1 Syntactic Analysis

In order to find links to the bug database, we split every log message into a stream of tokens. A token is one of the following items:

- a *bug number*, if it matches one of the following regular expressions (given in FLEX syntax):
 - `bug[# \t]*[0-9]+`,
 - `pr[# \t]*[0-9]+`,
 - `show_bug\.cgi\?id=[0-9]+`, or
 - `\[[0-9]+\]`
- a *plain number*, if it is a string of digits `[0-9]+`
- a *keyword*, if it matches the following regular expression: `fix(e[ds])?|bugs?|defects?|patch`
- a *word*, if it is a string of alphanumeric characters

Every number is a potential link to a bug. For each link, we initially assign a syntactic confidence *syn* of zero and raise the confidence by one for each of the following conditions that is met:

1. The number is a *bug number*.
2. The log message contains a *keyword*,
or the log message contains only *plain* or *bug numbers*.

Thus the syntactic confidence *syn* is always an integer number between 0 and 2. As an example, consider the following log messages:

- Fixed bug 53784: `.class file missing from jar file export`
The link to the bug number 53784 gets a syntactic confidence of 2 because it matches the regular expression for bug and contains the keyword *fixed*.
- 52264, 51529
The links to bugs 52264 and 51529 have syntactic confidence 1 because the log message contains only numbers.
- Updated copyrights to 2004
The link to the bug number 2004 has a syntactic confidence of 0 because there is no syntactic evidence that this number refers to a bug.

3.2 Semantic Analysis

In the previous section, we inferred links that point from a transaction to a bug report. To validate a link (t, b) we take information about its transaction t and check it against information about its bug report b . Based on the outcome we assign the link a semantic level of confidence.

Initially, a link (t, b) has semantic confidence of 0 which is raised by 1 whenever one of the following conditions is met:

- The bug b has been resolved as *FIXED* at least once.¹
- The short description of the bug report b is contained in the log message of the transaction t .
- The author of the transaction t has been assigned to the bug b .²
- One or more of the files affected by the transaction t have been attached to the bug b .

This list is not meant to be exhaustive. One could for example check whether a change has been committed to the repository within a small timeframe around the time when a bug has been closed.³

Consider the following examples from ECLIPSE, which all have low confidence levels:

- Updated copyrights to 2004
The potential bug report number “2004” is marked as *invalid* and thus the semantic confidence of the link is zero.
- Fixed bug mentioned in bug 64129, comment 6
The number “6” appears in the comment for a fix. The syntactic confidence is 1, but the semantic confidence is 0.
- Support expression like `(i)+= 3;` and `new int[] {1}[0] + syntax error improvement`
“1” and “3” are (mistakenly) interpreted as bug report numbers here. Since the bug reports 1 and 3 have been fixed, the links both get a semantic confidence of 1.

¹Notice that only 27% of all bugs in the MOZILLA project are *FIXED* (47% for ECLIPSE).

²For this check, we need a mapping between the CVS and BUGZILLA *user accounts* of a project. For ECLIPSE, we mapped the accounts of the most active developers manually; for MOZILLA, we derived a simple heuristic based on the observation that email addresses were used as logins for both CVS and BUGZILLA.

³Čubranić and Murphy already applied this as a standalone technique to relate bugs to transactions in their HIPIKAT tool [4].

- Fixed bug 53784: .class file missing from jar file export.
The bug 53784 has not been closed, but resolved as LATER. Its short description is: “Different results when running under debugger” and author of the change has not been assigned this bug. Thus the semantic confidence of the link is 0.

However, there exists a bug 53284 with the following short description: “.class file missing from jar file export”. If the comment had contained a correct number, the link would be assigned the semantic confidence 3.

3.3 Results

We identified 25,317 links for ECLIPSE, connecting 47% of fixed bugs with 29% of transactions and 53,574 links for MOZILLA, connecting 55.30% of fixed bugs with 43.91% of transactions. Tables 1 and 2 summarize the distribution of links across different classes of syntactic and semantic levels for both projects.

Based on a manual inspection of several randomly chosen links (see Section 3.2 for some examples), we decided to use only those links whose syntactic and semantic levels of confidence satisfy the following condition:

$$sem > 1 \vee (sem = 1 \wedge syn > 0)$$

Notice that we disregard less than 10% of links for both projects.

Our heuristics can be ported to almost any project that contains in the log messages links to a bug database. In some cases it may be necessary to implement further or different conditions to raise the confidence levels. However, the quality of the linking will always depend on the investigated project.

4. LOCATING FIX-INDUCING CHANGES

A fix-inducing change is a change that later gets undone by a fix. In this section, we show how to automatically locate fix-inducing changes.

Suppose that a change $\delta \in t$, which is known to be a fix for bug b (thus a link (t, b) must exist), transforms the revision $r_1 = 1.17$ of `Foo.java` into $r_2 = 1.18$ (see Figure 2), i.e., δ introduces new lines to r_2 or changes and removes lines of r_1 . First, we detect the lines L that have been touched by δ in r_1 . These are the locations of the fix. To locate them, we use the CVS `diff` command. In our example, we assume that line 20 and 40 have been changed and line 60 has been deleted, thus the fix locations in r_1 are $L = \{20; 40; 60\}$.

Next, we call the CVS `annotate` command for revision $r_1 = 1.17$ because this was the last revision without the fix; in contrast, revision $r_2 = 1.18$ already contains the applied fix. The annotations prepend each line with the most recent revision that touched this line. Additionally, CVS includes the developer and the date in the output. We show an excerpt of the annotated file in Figure 3. The CVS `annotate` command is only reliable for text files, thus we ignore all files that are marked as binary in the repository.

We scan the output and take for each line $l \in L$ the revision r_0 that annotates line l . These revisions are candidates for fix-inducing changes. We add (r_0, r_2) to the candidate set S , which is in our example $S = \{(1.11, 1.18); (1.14, 1.18); (1.16, 1.18)\}$.

From this set, we remove pairs (r_a, r_b) for which it is not possible that r_a induced the fix r_b —for instance, because r_a was committed to CVS *after* the bug fixed by r_b has been reported. In particular, we say that such a pair (r_a, r_b) is a *suspect* if r_a was committed after the *latest* reported bug linked with the revision r_b . Suspect changes could not contribute to the failure observed in the bug report. In Figure 2 the pairs $(1.14, 1.18)$ and $(1.16, 1.18)$ are examples of suspects.

We investigate suspects further on:

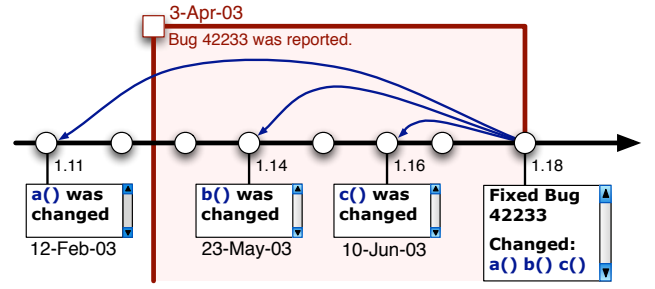


Figure 2: Locate fix-inducing changes for bug 42233

```
$ cvs annotate -r 1.17 Foo.java
```

```
...
19: 1.11 (john 12-Feb-03): public int a() {
20: 1.11 (john 12-Feb-03):     return i/0;
...
39: 1.10 (mary 12-Jan-03): public int b() {
40: 1.14 (kate 23-May-03):     return 42;
...
59: 1.10 (mary 17-Jan-03): public void c() {
60: 1.16 (mary 10-Jun-03):     int i=0;
...
```

Figure 3: CVS annotations for `Foo.java`

- We say that a suspect (r_a, r_b) is a *partial fix* if r_a is a fix.
Some bugs are fixed more than once. It may happen that one of the previous attempts was fixed by a later one, or that the bug is fixed across several transactions.
- We say that a suspect (r_a, r_b) is a *weak suspect* if there exists a pair (r_a, r_c) which is not a suspect.
A weak suspect indicates a revision for which there exists an alternative evidence of being fix-inducing, e.g., revision 1.14 may be a suspect for bug 42233 in Figure 2, but it still can be a strong candidate for another bug.
- We say that a suspect (r_a, r_b) is a *hard suspect* if it is neither a partial fix, nor a weak suspect.
A hard suspect indicates a revision for which there is no real evidence of being fix-inducing.

We say that a revision r is *fix-inducing* if there exists a pair $(r, r_x) \in S$ which is not a hard suspect. We say that a transaction t is *fix-inducing* if one of its revisions is fix-inducing.

5. FIRST RESULTS

We extracted fix-inducing changes for two large open-source projects: ECLIPSE and MOZILLA. We considered all changes and bugs until January 20, 2005; our database contains 78,954 transactions for ECLIPSE and 109,658 transactions for MOZILLA. They account for 278,010 and 392,972 individual revisions for both projects, respectively.

5.1 Fix-Inducing Transactions are Large

In our first experiment, we examined if the span of the transaction (i.e. the number of files touched) correlates with the fact that the transaction is fix-inducing. Table 3 presents the average sizes of transactions for ECLIPSE. The transactions are split into four classes, depending on whether the transaction is a fix, fix-inducing, both, or none. For instance, the top-left cell means that

syn / sem	0	1	2	3	4	total
0	270 (1%)	1,287 (5%)	2,057 (8%)	1,439 (6%)	2 (0%)	5,055 (20%)
1	324 (1%)	4,152 (16%)	9,265 (37%)	1,581 (6%)	5 (0%)	15,327 (61%)
2	110 (0%)	1,922 (8%)	2,421 (10%)	482 (2%)	0 (0%)	4,935 (19%)
total	704 (3%)	7,361 (29%)	13,743 (54%)	3,502 (14%)	7 (0%)	25,317 (100%)

Table 1: Distribution of links across different classes of syntactic and semantic confidence levels in ECLIPSE

syn / sem	0	1	2	3	4	total
0	560 (1%)	2,899 (5%)	4,281 (8%)	639 (1%)	8 (0%)	8,387 (16%)
1	1,211 (2%)	9,059 (17%)	16,336 (30%)	2,241 (4%)	22 (0%)	28,669 (54%)
2	478 (1%)	5,250 (10%)	9,133 (17%)	1,645 (3%)	12 (0%)	16,518 (31%)
total	2,249 (4%)	17,208 (32%)	29,750 (55%)	4,525 (8%)	42 (0%)	53,574 (100%)

Table 2: Distribution of links across different classes of syntactic and semantic confidence levels in MOZILLA

	fix-inducing	¬fix-inducing	all
fix	3.82±26.32	2.08± 7.42	2.73± 7.87
¬ fix	11.30±63.02	2.77±14.94	3.81±26.32
all	7.49±44.37	2.61±13.66	3.52±22.81

Table 3: Average sizes of fix and fix-inducing transactions for ECLIPSE

	fix-inducing	¬fix-inducing	all
fix	5.79±37.37	2.12± 9.74	4.39±30.05
¬ fix	4.61±30.59	1.91±10.30	3.05±21.39
all	5.19±34.12	1.97±10.13	3.58±25.23

Table 4: Average sizes of fix and fix-inducing transactions for MOZILLA

the average size of transactions which are fixes *and* induce later on a fix is 3.82 (with a standard deviation “±” of 26.32).

Additionally, Table 3 shows that fix-inducing transactions are roughly three times larger than non fix-inducing transactions. Table 4 presents the same breakdown for MOZILLA which shows a similar trend.

Such data can be automatically retrieved from all projects that supply both a version archive and a bug database. It is especially worthy when deciding where to spend efforts in *quality assurance*. If we were in charge of the ECLIPSE project, for instance, we would take care that large extensions are well reviewed and tested, as these have a high potential for inducing later fixes.

5.2 Don’t Program on Fridays

We broke down changes by the day of the week when they were applied. We distinguished between *bugs* as indicated by fix-inducing changes, and *fixes* as detected by links to the bug database. Bugs may be also fixes, we refer to such changes as *fix-inducing fixes*; they have been previously been used for visualization by Baker and Eick [1]. Finally, there are changes that are no bugs and no fixes.

$$P(\text{fix}) + P(\text{bug}) - P(\text{bug} \cap \text{fix}) + P(\neg \text{bug} \cap \neg \text{fix}) = 100\%$$

We measured the frequencies of the categories mentioned above. Table 5 presents the results for ECLIPSE. The likelihood $P(\text{bug})$ that a change will induce a fix is highest on Friday. The same holds

% of revisions	Day of Week							avg
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
$P(\text{fix})$	18.4	20.9	20.0	22.3	24.0	14.7	16.9	20.8
$P(\text{bug})$	11.3	10.4	11.1	12.1	12.2	11.7	11.6	11.4
$P(\text{bug} \cap \text{fix})$	4.6	4.8	4.6	5.2	5.6	4.5	4.5	4.9
$P(\neg \text{bug} \cap \neg \text{fix})$	74.9	73.5	73.5	70.8	63.4	78.1	76.0	72.7
$P(\text{bug} \text{fix})$	25.1	22.9	23.3	23.5	23.2	30.3	26.4	23.7
$P(\text{bug} \neg \text{fix})$	8.2	7.1	8.1	8.8	8.7	8.4	8.6	8.1

Table 5: Distribution of fixes and fix-inducing changes across day of week in ECLIPSE

% of revisions	Day of Week							avg
	Mon	Tue	Wed	Thu	Fri	Sat	Sun	
$P(\text{fix})$	42.5	46.5	49.7	45.9	48.4	50.2	61.1	48.5
$P(\text{bug})$	39.1	44.1	41.2	40.8	46.2	44.9	26.4	41.5
$P(\text{bug} \cap \text{fix})$	19.4	23.6	22.8	21.6	26.9	19.6	13.2	21.9
$P(\neg \text{bug} \cap \neg \text{fix})$	37.8	33.0	31.9	34.9	32.3	24.5	25.7	31.9
$P(\text{bug} \text{fix})$	45.7	50.8	45.8	47.1	55.6	39.1	21.6	45.2
$P(\text{bug} \neg \text{fix})$	34.1	38.3	36.7	35.5	37.3	50.6	33.9	38.1

Table 6: Distribution of fixes and fix-inducing changes across day of week in MOZILLA

for MOZILLA (see Table 6). Friday is the day where most ECLIPSE developers do fixes, for MOZILLA this is Sunday.

We used fix-inducing fixes to investigate whether non-fixes or fixes are more likely to be fix-inducing. Table 5 shows that for ECLIPSE, the average likelihood of introducing a fix-inducing change is almost three times higher for fixes, indicated by $P(\text{bug} | \text{fix})$, than for regular changes, indicated by $P(\text{bug} | \neg \text{fix})$. This does not hold for MOZILLA (see Table 6). The risk that a fix will be later undone is highest for ECLIPSE on Saturdays, and for MOZILLA on Fridays.

Almost every second change in MOZILLA is a fix and two out of five changes are fix-inducing. In the future we will investigate MOZILLA to find out what makes MOZILLA risky.

Besides the day of week, one can easily determine further properties of a change that correlate with inducing fixes—such as the development group, or the involved modules. Again, all this data is automatically retrieved for arbitrary projects.

6. RELATED WORK

To our knowledge, this is the first work that shows how to locate fix-inducing changes in version archives. However, fix-inducing changes have been used previously under the name *dependencies* by Purushothaman and Perry [9] to measure the likelihood that small changes introduce errors. Baker and Eick proposed a similar concept of *fix-on-fix changes* [1]. Fix-on-fix changes are less general than fix-inducing changes because they require both changes to be fixes.

In order to locate fix-inducing changes, we need first to *identify fixes* in the version archive. Mockus and Votta developed a technique that identifies reasons for changes (e.g., fixes) in the log message of a transaction [7]. In our approach, we refine the techniques of Čubranić and Murphy [4] and of Fischer, Pinzger, and Gall [6, 5], who identified references to bug databases in log messages and used these references to infer links from CVS archives to BUGZILLA databases.

Čubranić and Murphy additionally inferred links in the other direction, from BUGZILLA databases to CVS archives, by relating bug activities to changes. This has the advantage to identify fixes that are not referenced in log messages. For more details about this approach, we refer to [3].

Rather than searching for fix-inducing changes, one can also directly determine *failure-inducing changes*, where the presence of the failure is determined by an automated test. This was explored by Zeller, applying Delta Debugging on multiple versions [11].

7. CONCLUSION

As soon as a project has a bug database as well as a version archive, we can link the two to identify those changes that caused a problem. Such fix-inducing changes have a wide range of applications. In this paper, we examined the properties of fix-inducing changes in the ECLIPSE and MOZILLA projects and found, among others, that the larger a change, the more likely it is to induce a fix; checking for other correlated properties is straight-forward. We also found that in the ECLIPSE project, fixes are three times as likely to induce a later change than ordinary enhancements. Such findings can be generated automatically for arbitrary projects.

Besides the applications listed in Section 1, our future work will focus on the following topics:

Which properties are correlated with inducing fixes? These can be properties of the change itself, but also properties or metrics of the object being changed. This is a wide area with several future applications.

How do we disambiguate earlier changes? If a fixed location has been changed multiple times in the past, which of these changes should we consider as inducing the fix? We are currently evaluating a number of disambiguation techniques.

How do we present the results? Simply knowing which changes are fix-inducing is one thing, but we also need to present our findings. We are currently exploring visualization techniques to help managers as well as programmers.

For ongoing information on the project, see

<http://www.st.cs.uni-sb.de/softvevo/>

Acknowledgments.

This project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Christian Lindig and the anonymous MSR reviewers provided valuable comments on earlier revisions of this paper.

8. REFERENCES

- [1] M. J. Baker and S. G. Eick. Visualizing software systems. In *Proceedings of the 16th International Conference on Software Engineering*, pages 59–70. IEEE Computer Society Press, May 1994.
- [2] N. Barnes. Bugzilla database schema. Technical report, Ravenbrook Limited, July 2004. <http://www.ravenbrook.com/project/p4dti/master/design/bugzilla-schema/>.
- [3] D. Čubranić. *Project History as a Group Memory: Learning From the Past*. PhD thesis, University of British Columbia, Canada, Dec. 2004.
- [4] D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proc. 25th International Conference on Software Engineering (ICSE)*, pages 408–418, Portland, Oregon, May 2003.
- [5] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE.
- [6] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proc. International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, Sept. 2003. IEEE.
- [7] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. International Conference on Software Maintenance (ICSM 2000)*, pages 120–130, San Jose, California, USA, Oct. 2000. IEEE.
- [8] *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, UK, May 2004.
- [9] R. Purushothaman and D. E. Perry. Towards understanding the rhetoric of small changes. In MSR 2004 [8], pages 90–94.
- [10] The Bugzilla Team. *The Bugzilla Guide - 2.18 Release*, Jan. 2005. <http://www.bugzilla.org/docs/2.18/html/>.
- [11] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)*, volume LNCS 1687. Springer Verlag, 1999.
- [12] T. Zimmermann and P. Weißgerber. Preprocessing CVS data for fine-grained analysis. In MSR 2004 [8], pages 2–6.

Error Detection by Refactoring Reconstruction

Carsten Görg
Saarland University
Computer Science
D-66041 Saarbrücken
Germany
goerg@cs.uni-sb.de

Peter Weißgerber
Catholic University Eichstätt
Computer Science
D-85072 Eichstätt
Germany
peter.weissgerber@ku-eichstaett.de

ABSTRACT

In many cases it is not sufficient to perform a refactoring only at one location of a software project. For example, refactorings may have to be performed consistently to several classes in the inheritance hierarchy, e.g. subclasses or implementing classes, to preserve equal behavior.

In this paper we show how to detect incomplete refactorings – which can cause long standing bugs because some of them do not cause compiler errors – by analyzing software archives. To this end we reconstruct the class inheritance hierarchies, as well as refactorings on the level of methods. Then, we relate these refactorings to the corresponding hierarchy in order to find missing refactorings and thus, errors and inconsistencies that have been introduced in a software project at some point of the history.

Finally, we demonstrate our approach by case studies on two open source projects.

1. INTRODUCTION

Refactoring is the process of changing a software system such that it does not alter the external behavior of the code, but improves its internal structure [2]. In many cases the same refactoring has to be applied to more than one entity to achieve that the behavior really does not alter. For example, changing a method signature often requires to change the signature of methods in sub, super, and sibling classes as well. To help programmers with this task modern integrated development environments, like ECLIPSE [7], provide (semi-) automated application of refactoring to a software project. But unfortunately, not all programmers make consistent use of this feature and possibly introduce bugs, which are hard to find later on.

In this paper, we show an approach to investigate the application of refactorings over the lifetime of a software system. In particular we check if refactorings have been performed

consistently, i.e. in such a way that the behavior of the software system has not altered. We concentrate on refactorings of types *Add/Remove Parameter* and *Rename Method*. When a refactoring of one of these types is applied to a method in a class, it should also be applied to the corresponding methods of subclasses in most cases, and in some cases even of sibling classes. If a developer fails to do so, two kinds of errors can occur:

- *An interface method or an abstract method is no longer implemented in a subclass.* This kind of error results in compile time errors, and thus, is easy to detect.
- *The refactored method is inherited by a subclass instead of being overwritten.* However, the project can still be compiled without any problems and, thus, the developer may not notice the problem for a long time.

Our approach is applicable in two scenarios: First, we can search in existing software repositories for incomplete refactorings that have been done in the past and have not been corrected yet. In addition to this, we can assist the developer with the daily work: Every time the developer commits source code to the repository a tool can check the committed code for incomplete refactorings and warn if necessary.

The remainder of this paper is organized as follows: First, we explain in Section 2 how refactorings can be extracted from a software archive. Then, in Section 3 we show how to check for consistency of the reconstructed refactorings. Section 4 presents case studies of two open source projects. After that, we outline related work in Section 5. Finally, Section 6 summarizes our findings.

2. UNCOVERING REFACTORINGS

In this section we present a technique to detect refactorings on the level of methods in a software archive managed by cvs [1]. At first, we explain how we preprocess the repository data to get easy and fast access to it. After that, we take a closer look on how to retrieve classes and methods of JAVA files. We need this information in the following step where we analyze if methods have been refactored by renaming them, or adding resp. removing parameters. After that, we discuss in few words how we deal with additional changes to refactored methods and with ambiguous refactorings. Finally, we explain how we relate refactorings to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00

complete configurations of the software project. A more detailed and formal description of this technique is described in [3].

2.1 Preprocessing the CVS Data

Unfortunately, the direct access on the data is much too slow, and furthermore, some information has to be recovered from different places of the repository. Thus, the first step of our technique is to extract the repository completely, recover information where necessary, and store this data in a relational database. The details of this extraction step are described in [8]. After the extraction we can access the following information:

Versions. A *version* describes one revision of a file in the cvs repository (e.g. file `org/epos/epos.java` in revision 1.4). For each revision in the repository we store information about the committer, the log message, the timestamp, the state, the predecessor revision if one exists, and the text.

Transactions. A *transaction* is the set of versions that have been committed to the repository at the same time by the same developer. As cvs splits commits that contain more than one file into single check-ins for each file and does not store which of these check-ins have been issued together, we use a sliding time window heuristic to recover transactions quite precisely.

Additionally, we need information about particular *configurations*. A configuration is a set of versions of distinct files. In our application, we are only interested in *active configurations after transactions*. An active configuration after a transaction is the set of versions a developer can access in his working directory after performing the transaction.

If we want to examine a new commit to the repository instead of searching existing failures, it is sufficient to consider the set of versions belonging to the current commit and to build the active configuration after this committed transaction.

2.2 Parsing Syntactical Blocks of Versions

To gather information about which classes and methods are contained in a JAVA file (and thus, may be affected by refactorings) we use a light-weight parser that identifies a) classes in versions and b) methods in classes. The classes are identified by its fully-qualified name while methods are identified by their signature, e.g. `parseInt(String):int`. If we compare the classes of a version v to the classes of its predecessor version v' , we note by $\text{COMMON}_C(v, v')$ the set of classes that exist in both v and v' .

For each class $c \in \text{COMMON}_C(v, v')$ we can now compute the sets of added, removed, and common methods by comparing the methods contained in the class in v with the methods in the corresponding class in v' :

- $\text{ADDED}_M(v, v', c)$ method that have been added to c ;
- $\text{COMMON}_M(v, v', c)$ methods that are contained in c in both versions v' and v ;

- $\text{REMOVED}_M(v, v', c)$ methods that have been deleted from c .

2.3 Identifying Local Refactorings

To find refactorings performed in single classes we iterate over the set V_r of all versions of JAVA files in the repository. As refactorings describe changes with respect to the predecessor version, we ignore versions that have no predecessor. For all remaining $v \in V_r$, we take the predecessor version v' and test in all classes $c \in \text{COMMON}_C(v, v')$ if we can find one of the following refactorings:

Rename Method.

If we find a method $m_1 \in \text{REMOVED}_M(v, v', c)$ and a method $m_2 \in \text{ADDED}_M(v, v', c)$ that have exactly the same text, the same return type, the same parameters, but different names we consider this as a *Rename Method* refactoring.

Add Parameter.

If we find a method $m_1 \in \text{REMOVED}_M(v, v', c)$ and a method $m_2 \in \text{ADDED}_M(v, v', c)$ that have the same name and the same return type, but m_2 has additional parameters with respect to m_1 , we consider this as an *Add Parameter* refactoring. The *Remove Parameter* refactoring is recognized analogously.

2.4 Impure and Ambiguous Refactorings

A major problem in parsing the version archive for refactorings is that often the refactorings are *impure*: The developer has not only performed the refactoring, but has changed other things at the same location at the same time, or the developer has performed two different refactorings on the same entity. Our approach is not capable to find *Rename Method* and *Add/Remove Parameter* refactorings that have been applied to the same entity in the same transaction. In addition, these refactorings cannot be recognized if other changes have been simultaneously performed to the method signature (i.e. name, parameter list, and return type). However, if only the body of the method has been changed together with the refactoring, we still detect *Add/Remove Parameter* refactorings. As a consequence, it can happen that we fail to detect some inconsistent refactorings: Assume that in class A a method has been renamed and at the same time an additional parameter has been added. Furthermore, assume that class B is a subclass of A. Thus, the corresponding method in B has to be refactored the same way. But, as we cannot detect the refactoring in A we also cannot detect if A has been refactored but B has been missed.

Unfortunately, it is not always possible to unambiguously identify all refactorings as the following example illustrates: a class contains the methods $m(t_1, t_2) : t_r$ and $m(t_2, t_3) : t_r$ and after a new transaction it contains instead of these two the new method $m(t_1, t_2, t_3) : t_r$. Now it is undecidable if this is an *Add Parameter* refactoring from $m(t_1, t_2) : t_r$ by adding t_3 or from $m(t_2, t_3) : t_r$ by adding t_1 . In such cases, we take all matching refactorings into account.

2.5 Relating Refactorings to Configurations

To detect errors it is not sufficient only to look at the classes that have been currently changed and may contain refactorings, but also at the other classes that have been part of

the project and (maybe erroneously) not have been updated when the developer has performed the check-in to the repository. Thus, additionally to the changed versions, for each file that has not been changed in a transaction we take the most recent version into account and parse it for its classes and methods. We call the set of changed versions and most recent versions of non-changed files the *configuration active after transaction t* .

2.6 Reconstructing the Class Hierarchy

Before we can check if refactorings have been applied consistently to related classes with respect to the class hierarchy, we have to construct the inheritance tree of the examined JAVA project. Thus, we iterate over the set of transactions and build for each the configuration active after it. For each such configuration we know the set of *project classes* – these are the JAVA classes in the workspace of the developer after the transaction. Thus, the inheritance tree for both *implements* and *extends* relations is built by iterating over these project classes, parsing them for the declarations after the *implements* resp. *extends* keyword and relating the found class references (using the import declarations) to a) the classes of the JAVA standard API and b) the classes contained in JAR files.

3. CHECKING CONSISTENCY

In this section we explain how we check if the reconstructed refactorings of types *Rename Method* and *Add/Remove Parameter* have been applied consistently to the software project. For each refactoring we compute a list of other possible candidates in the current configuration and check if the refactoring has been applied also to them. If not, we regard this as a possible inconsistency.

The list of possible candidates is computed as follows: Let *ref* be a refactoring changing the parameter list of method *m* in class *c* in configuration *conf*₁ from *params*₁ to *params*₂ in configuration *conf*₂. If in configuration *conf*₂ exists a superclass, subclass or sibling class¹ of class *c* that contains the method *m* with parameter list *params*₁ then this method is a possible candidate for a missing refactoring. Analogously, we compute further candidates by taking refactorings renaming a method into account.

The methods found as candidates split into two different categories:

- **Methods in subclasses:** If methods in a class are refactored it is likely that overwritten methods in subclasses should be refactored the same way. For this type of error candidates one can distinguish between two subtypes: If the superclass is an interface, the unrefactored subclasses do not implement the interface correctly anymore and thus, cannot be compiled successfully. Otherwise, the affected subclass inherits the refactored method from the superclass and additionally, holds its own unrefactored method. Thus, the class can be compiled, but may behave incorrectly.

¹a sibling class of class *c* is an arbitrary subclass of a superclass of class *c*.

- **Methods in sibling classes:** There are also cases when sibling classes have to be updated although the superclass has not to be updated. For example, in one transaction of JEDIT at the same time in both classes *EnhancedMenuItem\$MouseHandler* and *EnhancedCheckBoxMenuItem\$MouseHandler* the method *mouseClicked* has been renamed to *mouseReleased*. As both classes are subclasses of the standard JAVA class *MouseAdapter* the actions implemented in these methods are not instantly triggered any more when the mouse has been clicked, but not until the mouse has been released again. Clearly, this refactoring should be applied at the same time to all classes extending from *MouseAdapter* to preserve a consistent user interface.

As we examine the whole software archive and start with the oldest configurations, it is possible that the missing parts of a detected inconsistent refactoring have been added some transactions later (this means the refactorings has been performed using multiple transactions). Hence, for each candidate we iterate over all configurations with a timestamp later than the considered configuration and look if the refactoring has been applied to the corresponding method. If we find such a later configuration, the inconsistency (and thus the error) has been resolved.

4. CASE STUDIES

We applied our techniques to detect inconsistent refactorings to the software archives of the open source projects JEDIT and TOMCAT. We found five candidates for erroneous transactions in JEDIT and seven in TOMCAT. For JEDIT two of these candidates are unrefactored methods in subclasses – both have been refactored in later transactions – and three are methods in sibling classes. For TOMCAT three candidates are methods in subclasses and four are methods in sibling classes. None of these unrefactored methods have been updated later. In the following paragraphs we explain some noticeable transactions in more detail.

4.1 Unrefactored Methods in Subclasses

We found an example of the subclasses type in Transaction 876 of JEDIT: An additional parameter has been added to the method *foldLevelChanged* of the interface *BufferChangeListener*. This interface is implemented by two classes: *BufferChangeAdapter* and *JEditTextArea\$BufferChangeHandler*. Thus, these classes should have been updated accordingly to ensure that they can be compiled. Interestingly, only *JEditTextArea\$BufferChangeHandler* has been initially updated. One month later the other class has been refactored, too, allowing it to be compiled again.

Also in JEDIT we found a second example where the developers seem to have noticed their mistake and corrected it some transactions later: In Transaction 1241 a developer has added a boolean parameter to the method *addTokenHandler* in the class *DefaultTokenHandler*. One day later, in the following transaction, the subclass *DisplayTokenHandler* has been updated accordingly. Note, that in this example the superclass is not an interface. This means, even the incorrect configuration was syntactically correct and did not produce compiler errors.

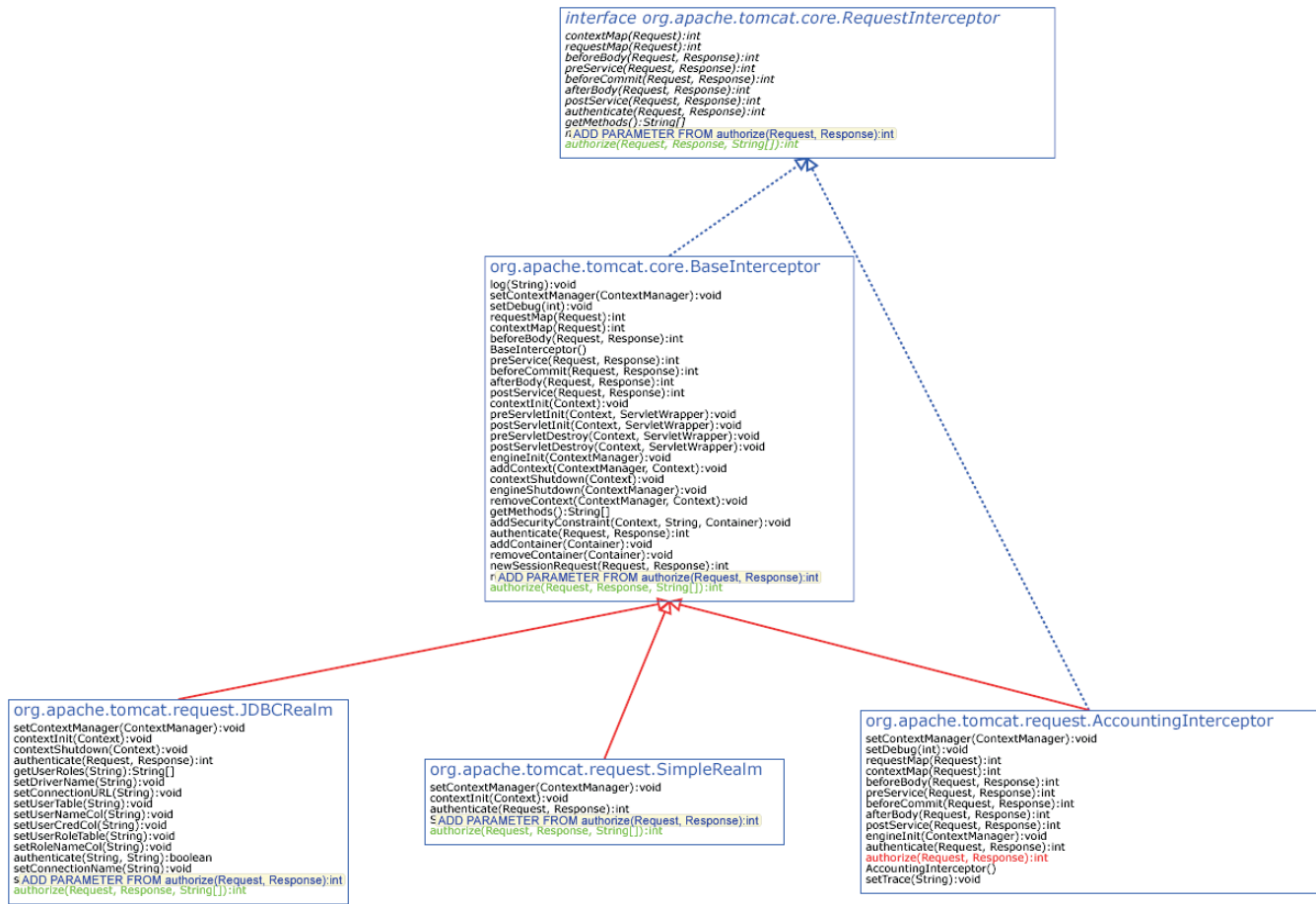


Figure 1: Add Parameter refactorings in transaction 1971 of TOMCAT.

But we also found transactions where the unrefactored class has not been updated later. For example, in Transaction 1475 of TOMCAT the method `handleTagBegin` has been extended by one parameter in different classes of the package `org.apache.jasper.compiler`, but not in the class `DumbParseEventListener`. The documenting comments in the source code of the class tell that the class is a testing class that should be removed some time.

This shows that our approach only finds candidate methods that may be buggy. Actually, these methods should be audited by a developer.

Figure 1 shows a more complex example for a maybe inconsistent *Add Parameter* refactoring in TOMCAT. In this visualization the involved classes and the inheritance relations between them are displayed in the UML notation, refactored methods are marked green (a tooltip describes the refactoring) and missing refactorings are colored red. In Transaction 1971 the method `authorize` of the interface `RequestInterceptor` has been extended by one parameter of type `String[]`. This new parameter provides the roles which the person that should be authorized owns (e.g. “admin”). Consequently, the class `BaseInterceptor` that implements this interface has been updated, too. Moreover, two of three subclasses of `BaseInterceptor` have also been changed: `Simple-`

`Realm` and `JDBCRealm`. Surprisingly, the class `AccountingInterceptor` that both *implements* `RequestInterceptor` and at the same time *extends* `BaseInterceptor` has not been updated. Anyway, this class can be successfully compiled because the `authorize` method as needed because of the interface is inherited from the superclass `BaseInterceptor`.

To get a deeper understanding of what happened in the described transaction we checked out the source code of the involved classes at the time of the transaction. We found that in the unrefactored class `AccountingInterceptor` the questionable method always returns the value “0” for “authorized”. This is exactly what the updated method that is inherited from `BaseInterceptor` does. Thus, there was no need to update the method in this class because the inherited one does the right thing and the unrefactored one does the same and can be used as abbreviation.

4.2 Unrefactored Methods in Sibling Classes

As we have explained in Section 3 in some cases it is also necessary to update sibling classes although the superclass of a class has not been refactored. In JEDIT we found three and in TOMCAT four transactions where a class has been refactored on method level but the siblings have not been updated. None of these have been updated later in the history.

As siblings do not inherit from each other, they can implement completely different methods. Thus, again the found transactions are only candidates and have to be checked by a developer.

For example in Transaction 3240 of TOMCAT, a parameter has been added to the method `addTagRules` in the class `ProfileLoader`. This class is a subclass of `BaseInterceptor` which has two other subclasses: `ContextXMLReader` and `ServiceXMLReader`. In these classes the described refactoring has not been performed.

5. RELATED WORK

General information on refactoring are presented in Fowler's book [2]. Demeyer et. al presented some metrics-based heuristics [6] to detect refactorings in successive configurations of software systems. They primarily concentrated on movements, splits, and merges of methods and performed case studies on three open source projects. In contrast to our work, they did not access the software archive to get successive configurations. Software metrics like the McCabe complexity [4] can also give hints about files and classes that are likely to contain errors. Ostrand and Weyuker [5] used a different approach to detect and predict possible problems in a software. They mined bug databases in order to predict fault-prone files.

6. CONCLUSIONS AND OUTLOOK

In this paper we introduced an approach to reconstruct refactorings on method level from software archives. Then we explained how to check if they have been consistently applied. The case studies on the open source projects JEDIT and TOMCAT show that our approach allows to detect candidates for incomplete refactorings. In two cases, the candidates have been applied later and so it turned out that they have been really missing. In two other cases, they have also been missing, but have not been applied, because they concerned classes, which have no longer been under active development. The remaining cases have to be inspected by the developer to decide whether the refactorings are really missing or not.

For future work we plan to investigate how to reconstruct more complex types of refactorings from software archives. Moreover, there are additional refactorings that could be checked for consistency without altering our current architecture but are not yet implemented: For example the *Extract Interface* refactoring that creates an interface containing the methods of a class and lets the class implement this interface could be detected by comparing the method declarations of new interfaces to the methods of existing classes. In addition to this, the *Generalize Type* refactoring that changes the type of an object to a more general type could be recognized using the class hierarchy.

As our approach can be used to support developers in their daily work to prevent errors in refactoring tasks, it would be desirable to let our algorithms run in the background while the developer is working. To achieve this we are planning to integrate it into a development environment like ECLIPSE.

Furthermore, we plan to extend our case study to more software projects of different size, complexity, and age.

7. REFERENCES

- [1] P. Cederqvist. Version Management with CVS. <http://www.cvshome.org/docs/manual/>.
- [2] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2001.
- [3] C. Görg and P. Weißgerber. Detecting and Visualizing Refactorings from Software Archives. In *Proceedings of the 13th International Workshop on Program Comprehension (IWPC 2005)*, St. Louis, Missouri, U.S., 2005.
- [4] T. J. McCabe. *A Complexity Measure*. IEEE Transactions on Software Engineering, Vol. 2, 1976.
- [5] T. J. Ostrand and E. J. Weyuker. A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, U.K., 2004.
- [6] O. N. Serge Demeyer, Stéphane Ducasse. Finding Refactorings via Change Metrics. In *Proceedings of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000)*, Minneapolis, Minnesota, U.S., 2000.
- [7] The Eclipse Foundation. Eclipse Homepage. <http://www.eclipse.org>.
- [8] T. Zimmermann and P. Weißgerber. Preprocessing CVS Data for Fine-Grained Analysis. In *Proc. International Workshop on Mining Software Repositories (MSR 2004)*, Edinburgh, Scotland, U.K., 2004.

Education

Software Repository Mining with Marmoset: An Automated Programming Project Snapshot and Testing System

Jaime Spacco, Jaymie Strecker, David Hovemeyer, and William Pugh
Dept. of Computer Science
University of Maryland
College Park, MD, 20742 USA
{jspacco,strecker,daveho,pugh}@cs.umd.edu

ABSTRACT

Most computer science educators hold strong opinions about the “right” approach to teaching introductory level programming. Unfortunately, we have comparatively little hard evidence about the effectiveness of these various approaches because we generally lack the infrastructure to obtain sufficiently detailed data about novices’ programming habits.

To gain insight into students’ programming habits, we developed Marmoset, a project snapshot and submission system. Like existing project submission systems, Marmoset allows students to submit versions of their projects to a central server, which automatically tests them and records the results. Unlike existing systems, Marmoset also collects fine-grained code snapshots as students work on projects: each time a student saves her work, it is automatically committed to a CVS repository.

We believe the data collected by Marmoset will be a rich source of insight about learning to program and software evolution in general. To validate the effectiveness of our tool, we performed an experiment which found a statistically significant correlation between warnings reported by a static analysis tool and failed unit tests.

To make fine-grained code evolution data more useful, we present a data schema which allows a variety of useful queries to be more easily formulated and answered.

1. INTRODUCTION

While most computer science educators hold strong opinions about the “right” way to teach introductory level programming, there is comparatively little hard evidence to support these opinions. The lack of evidence is especially frustrating considering the fundamental importance to our discipline of teaching students to program. We believe that the lack of evidence is at least partly attributable to a lack of suitable infrastructure to collect quantitative data about students’ programming habits.

To collect the desired data, we have developed Marmoset,

an automated project snapshot, submission, and testing system. Like many other project submission and testing systems ([11, 6, 7, 4]), Marmoset allows students to submit versions of their work on course projects and to receive automatic feedback on the extent to which submissions meet the grading criteria for the project. The grading criteria are represented by JUnit [8] tests, which are automatically run against each version of the project submitted by the student. In addition to JUnit tests, Marmoset also supports running the student’s code through static analysis tools such as bug finders or style checkers. Currently the only supported static checker is FindBugs [5]; we plan on trying with other static analysis tools such as PMD [10] and CheckStyle [1] in the future.

A novel feature of Marmoset is that in addition to collecting submissions explicitly submitted by students, an Eclipse [3] plugin called the Course Project Manager [13] automatically captures snapshots of a student’s code to the student’s CVS [2] repository each time she saves her files. These intermediate snapshots provide a detailed view of the evolution of student projects, and constitute the raw data we used as the basis for the experiments described in this paper.

Students can log in to the SubmitServer to view the results of the unit tests and examine any warnings produced by static checkers. The test results and static analysis warnings are divided into four categories:

- **Public Tests:** The source code for the public tests is made available to students upon their initial checkout of a project, and the results of public tests for submitted projects are always visible to students. (Students should already know these results since they can run these tests themselves).
- **Release Tests:** Release tests are additional unit tests whose source code is not revealed to students. The outcomes of release tests are only displayed to students if they have passed all of the public tests. Rather than allowing students unlimited access to release test results (as we do with public tests results), we allow limited access as follows. Viewing release tests costs one “release token”. Students receive three release tokens for each project and these tokens regenerate every 24 hours. Viewing release results allows the student to see the *number* of release tests passed and failed as well as the *names* of the first two tests failed. For example, for a project requiring the evaluation of various poker hands, a student may discover that they have passed

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR '05 St. Louis, MO USA

Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

6 out of 12 release tests and that the first two tests failed were *testThreeOfAKind* and *testFullHouse*. We have tried to make the release test names descriptive enough to give the student some information about what part of their submission was deficient, but vague enough to make the students think seriously about how to go about fixing the problem.

- **Secret Tests:** Like release tests, the code for secret tests is also kept private. Unlike release tests, the results of secret tests are never displayed to the students. These are equivalent to the private or secondary tests many instructors use for grading purposes. Although our framework supports them, the courses on which we report in this paper did not use any secret tests.
- **Static Checker Warnings:** We have configured Marmoset to run FindBugs on every submission and make the warnings visible to students. FindBugs warnings are provided solely to help students debug their code and to help us tune FindBugs; the results of FindBugs are not used for grading.

When compared to previous work, we feel Marmoset improves data collection in two major ways. First, by using the Course Project Manager Eclipse plugin, we can gather frequent snapshots of student code automatically and unobtrusively. Prior work on analyzing student version control data [9] focused on data that required the students to manually commit their code. One observation made by Liu et. al. is that students often don't use version control systems in a consistent manner. The Course Project Manager plugin has no such limitation.

Second, by providing the same testing framework for both development and grading, we can quantify the correctness of any snapshot along the development timeline of a project. This allows us to perform statistical analyses of the development history of each student.

2. STUDENT SNAPSHOT DATA

Of the 102 students in the University of Maryland CMSC 132 Fall 2004 course, 73 consented to be part of an IRB approved experimental study of how students learn to develop software. Other than signing a consent form and filling out an optional online survey about demographic data and prior programming experience, students participating in the experiment did not experience the course any differently than other students in the course, as the data collected for this research is routinely used during the semester to provide students with regular backups, automated testing and a distributed file system. From the 73 students who consented to participate in the study, we extracted from their CVS repositories over 51,502 snapshots, of which about 41,333 were compilable. Of the compilable snapshots, 33,015 compiled to a set of classfiles with a unique MD5 sum.

That 20% of the snapshots did not compile is not surprising, as snapshots are triggered by saving. In fact, we were pleasantly surprised that so many of our snapshots did compile.

We tested each unique snapshot on the full suite of unit tests written for that project. In addition, we checked each unique snapshot with the static bug finder FindBugs [5] and stored the results in the database. We also computed the CVS diff of the source of each unique submission with the

students	73
projects	8
student projects	569
snapshots	51,502
compilable	41,333
unique	33,015
total test outcomes	505,423
not implemented	67,650
exception thrown	86,947
assertion failed	115,378
passed	235,448

Table 1: Overall numbers for project snapshots and test outcomes

Problem	Exception			
	yes		no	
	Warning		Warning	
ClassCast	596	1,541	2,009	28,869
StackOverflow	627	792	255	29,437
Null Pointer	1,116	5,014	1,389	25,496

Table 2: Correlation between selected warnings and Exceptions

source of the preceding unique submission, and stored the total number of lines added or changed as well as the net change to the size of the files (we do not track deletes explicitly, though deletes do show up indirectly as net changes to the size of the source files).

We have performed a number of different kinds of analysis on the data, and continue to generate additional results. Unfortunately, space only allows us to present a small window into our research.

We have looked both at the changes between successive snapshots by an individual student, and at the features of each snapshot in isolation. When looking at changes between successive snapshots, we can examine the change in warnings between successive versions and whether there is any corresponding change in the number of unit test faults between versions. We can also look at the size of changes, and even manually examining the differences between versions where our defect warnings do not seem to correspond to the difference in actual faults (e.g., if a one line change caused a program to stop throwing `NullPointerExceptions`, but no change occurred in the number of defect warnings generated, is there something missing in our suite of defect detection tools?). While we have some results from this analysis, the complexity of those results makes them hard to present in the space available.

3. CORRELATION BETWEEN WARNINGS AND EXCEPTIONS

In this section, we show the correlation between selected bug detectors and the exceptions that would likely correspond to the faults identified by these detectors. We look at `ClassCastExceptions`, `StackOverflowError` and `NullPointerExceptions`. Before starting work on finding bugs in student code, we didn't have any bug detectors for `ClassCas-`

tExceptions or StackOverflowErrors. Based on our experience during class and leading up to this paper, we wrote some detectors for each. Table 2 shows the correlation between exceptions and the corresponding bug detectors in version 0.8.7 of FindBugs.

Note that cases where we warn about a possible infinite recursive loop, but do not experience a stack overflow exception during a test run, might not indicate a false positive warning. Instead, it is possible that the error signaled by the warning is, in fact, present but the presence of the error is masked during execution by the presence of other errors.

ClassCastExceptions typically arise in student code because of:

- An incorrect cast out of a collection. We believe that many of these would be caught by uses of parameterized collections.
- A collection is downcast to more specific class

```
(Set)Map.values()
```

- A cast to or from an interface that will not succeed in practice, but the compiler cannot rule out since it can't assume new classes will not be introduced. In the example below, although WebPage does not implement Map, we cannot rule out the possibility that a new class could be written that extends WebPage and implements Map:

```
public void crawl(WebPage w) {
    Map crawlMap = (Map)w;
```

- A cast where static analysis dooms the cast, even if additional classes are written, but the programmer has gone to some length to confuse the compiler:

```
public WebPage(URL u) {
    this.webpage = (WebPage)((Object)u);...+
```

We have written detectors to check for the last three casts. Surprising, all three (even the last one) also identify problems in production code; an instance of the the last error occurs in the Apache Xalan library.

Many of the StackOverflowErrors are caused by code that obviously implements infinite recursive loops, such as:

```
WebSpider() {
    WebSpider w = new WebSpider(); }
```

We wrote an initial detector based on experience during the fall semester, and that detector also found a number of infinite recursive loops in production code such as Sun's JDK 1.5.0 and Sun's NetBeans IDE. From examination of the research data collected from the course, we refined the infinite loop detector, to find more cases. This allowed us to find an additional infinite recursive loop in Sun's JDK. We found far more infinite recursive loops that we would have ever anticipated (17 of them in code shipped in JBoss 4.0.1sp1, 10 in code shipped with Sun's J2EE appserver).

For the NullPointerExceptions, we report the detectors that perform dataflow analysis to report possible NullPointerExceptions and a separate detector that looks for reference fields that are never written to but are read and deferred.

4. SCHEMA FOR REPRESENTING PROGRAM EVOLUTION

Our current analysis is somewhat limited, in that we can only easily measure individual snapshots, or changes between successive versions. We can't easily track, for example, which changes are later modified.

We want to be able to integrate code versions, test results, code coverage from each test run, and warnings generated by static analysis tools. In particular, we want to be able to ask questions such as:

- Which methods were modified during the period 6pm-9pm?
- During the period 6pm-9pm, which methods had more than 30 line modifications or deletions?
- Of the changes that modified a strcpy call into a strncpy call, how frequently was the line containing the strncpy call, or some line no more than 5 lines before it, modified in a later version?
- For each warning generated by a static analysis tool, which versions contain that warning?
- Which warnings are fixed shortly after they are presented to students, and which are ignored (and persist across multiple submissions)?

None of these questions can be easily asked using CVS based representations. We developed a schema/abstraction for representing program histories that make answering these questions much easier. A diagram of the schema is shown in Figure 1. Each entity/class is shown as a box, with arrows to other entities it has references to. This schema can be represented in a relational database, and most of the queries we want to ask can be directly formulated as SQL queries.

The schema we have developed is based on recognizing unique lines of a file. For example, we might determine that a particular line, with a unique key of 12638 and the text " i++; ", first occurs on line 25 of version 3 of the file "Foo.java", occurs on line 27 in version 4 (because two lines were inserted before it), occurs on line 20 in version 5 (because 7 lines above it were deleted in going from version 4 to version 5) and that version 5 is the last version unique line #12638 occurs.

It is important to understand that unique lines are not based on textual equality. Other occurrences of " i++; " in the same file or other files would be different unique lines. If a line containing " i++; " is reinserted in version 11, that is also a different unique line.

So in our database, we have a table that gives, for each line number of each file version, the primary key of the unique line that occurs at that line number.

4.1 Tracking Lines and Equivalence Classes

As given, two lines are considered identical only if they are textually identical: changing a comment or indentation makes it a different unique line. While we sometimes want to track changes at this granularity, we often want to track lines across versions as their comments are changed or even as small modifications are made.

We handle this by defining equivalence classes over unique lines of text. At the moment, we support the following equivalence relations:

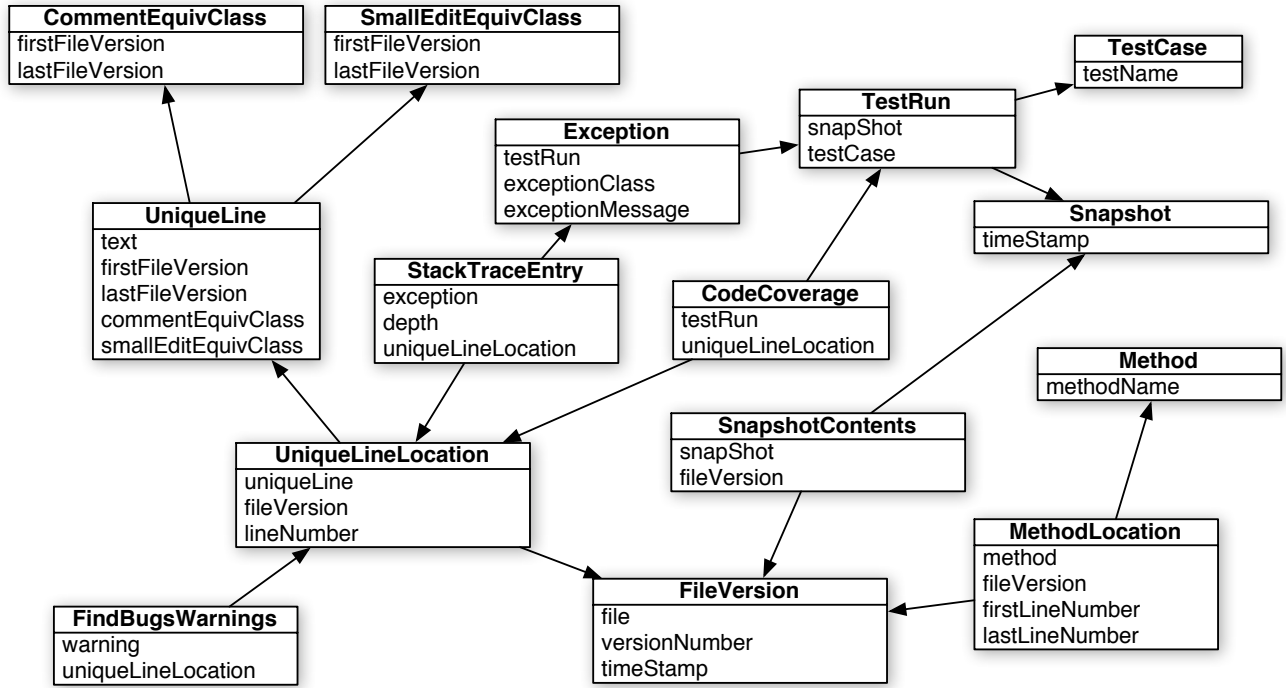


Figure 1: Schema for representing program evolution

- Identity: The lines are exactly identical.
- Ignore-Whitespace: When whitespace is ignored, the lines are identical.
- Ignore-SmallEdits: When whitespace is ignored, the lines are almost equal; their edit distance is small.
- Ignore-Comments: When whitespace and comments are ignored, the edit distance between the lines is small.

These equivalence relations are ordered from strictest to most relaxed. Thus, the lines " `a = b + c.foo();` " and " `a = b + x.foo(); /* Fixed bug */` " belong to the same Ignore-Comments and Ignore-SmallEdits equivalence classes, but not to the same Ignore-Whitespace and Identity equivalence classes. The equivalence classes are used to track individual lines as they evolve, not to identify textually similar lines of text.

There are various rules associated with identifying these unique lines and equivalence classes in a file:

- No crossings: If a line belonging to equivalence class X occurs before a line belonging to equivalence class Y in version 5 of a file, then in all versions in which lines belonging to equivalence classes X and Y occur, the line belonging to equivalence class X must occur before the line belonging to equivalence class Y.
- Unique representatives: In each version, only one line may belong to any given equivalence class.
- Nested equivalence classes: If two lines are equivalent under one equivalence relation, then they must be equivalent under all more relaxed relations.

The no crossing rule prevents us from recognizing cut-and-paste operations, in which a block of code is moved from one location to another. Recognizing and representing cut-and-paste (and other refactoring operations) is a tricky issue that we may try to tackle at some future point. However, handling that issue well would also mean handling other tricky issues, such as code duplication.

To calculate which lines belong to the same equivalence class, we have implemented a variation of the "diff" command to discover groups of mismatched lines, or deltas, between two versions. Our diff algorithm recursively computes deltas under increasingly relaxed equivalence relations. First, we find all deltas under the Identity relation, which is the strictest. For each delta, we apply the algorithm recursively, using the next strictest equivalence relation to compare lines. The final result is a "diff" of the versions for each equivalence relation. Because the recursive step of the algorithm only considers those deltas computed under stricter equivalence relations, the algorithm respects the three rules above.

4.2 Methods

Since we will sometimes wish to track which methods are modified by a change or covered by a test case, we also store, for each file version, the first and last line number associated with a method.

4.3 Other information

We represent a number of additional forms of information in our database. A snapshot consists of a set of file versions taken at some moment in time. Usually, a snapshot represents a compilable, runnable and testable snapshot of the

system.

Associated with a snapshot we can have test results and code coverage results. Typically, each project will have a dozen or more unit test cases. We run all of the unit tests on each snapshot, and also record which lines are covered by each test case. If a test case terminates by throwing an exception, we record the exception and stack trace in the database. The information we have linking lines in different versions of a file allows us to easily compare code coverage in different versions, or correlate code coverage with static analysis warnings or exceptions generated during test cases.

5. RELATED WORK

Many systems exist to automatically collect and test student submissions: some examples are [11, 6, 7, 4]. Our contribution is to control students' access to information about test results in a way that provides incentives to adopt good programming habits.

In [9], Liu et. al. study CVS histories of students working on a team project to better understand both the behavior of individual students and team interactions. They found that both good and bad coding practices had characteristic ways of manifesting in the CVS history. Our goals for the data we collect with our automatic code snapshot system are similar, although we consider individual students rather than teams. Our system has the advantage of capturing changes at a finer granularity: file modification, rather than explicit commit.

In [12], Schneider et al. advocate using a "shadow repository" to study a developer's fine-grained local interaction history in addition to milestone commits. This approach to collecting and studying snapshots is similar to our work with Marmoset. The principal difference is that we are not focused on large software projects with multiple developers, and so we can use a standard version control system such as CVS to store the local interactions.

In [4], Edwards presents a strong case for making unit testing a fundamental part of the Computer Science curriculum. In particular, he advocates requiring students to develop their own test cases for projects, using project solutions written by instructors (possibly containing known defects) to test the student tests. This idea could easily be incorporated into Marmoset.

6. ACKNOWLEDGMENTS

The second author is supported in part by a fellowship from the National Physical Science Consortium and stipend support from the National Security Agency.

7. REFERENCES

- [1] CheckStyle. <http://checkstyle.sourceforge.net>, 2005.
- [2] CVS. <http://www.cvshome.org>, 2004.
- [3] Eclipse.org main page. <http://www.eclipse.org>, 2004.
- [4] S. H. Edwards. Rethinking computer science education from a test-first perspective. In *Companion of the 2003 ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Anaheim, CA, October 2003.
- [5] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, October 2004.
- [6] D. Jackson and M. Usher. Grading student programs using ASSYST. In *Proceedings of the 1997 SIGCSE Technical Symposium on Computer Science Education*, pages 335–339. ACM Press, 1997.
- [7] E. L. Jones. Grading student programs - a software testing approach. In *Proceedings of the fourteenth annual consortium on Small Colleges Southeastern conference*, pages 185–192. The Consortium for Computing in Small Colleges, 2000.
- [8] JUnit, testing resources for extreme programming. <http://junit.org>, 2004.
- [9] Y. Liu, E. Stroulia, K. Wong, and D. German. Using CVS historical information to understand how students develop software. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.
- [10] PMD. <http://pmd.sourceforge.net>, 2005.
- [11] K. A. Reek. A software infrastructure to support introductory computer science courses. In *Proceedings of the 1996 SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, PA, February 1996.
- [12] K. A. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer's local interaction history. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburgh, Scotland, May 2004.
- [13] J. Spacco, D. Hovemeyer, and W. Pugh. An eclipse-based course project snapshot and submission system. In *3rd Eclipse Technology Exchange Workshop (eTX)*, Vancouver, BC, October 24, 2004.

Mining Student CVS Repositories for Performance Indicators

Keir Mierle
Dept. Electrical & Computer Engineering
University of Toronto
keir@cs.utoronto.ca

Kevin Laven, Sam Roweis, Greg Wilson
Dept. Computer Science
University of Toronto
{klaven,roweis,gvwilson}@cs.utoronto.ca

ABSTRACT

Over 200 CVS repositories representing the assignments of students in a second year undergraduate computer science course have been assembled. This unique data set represents many individuals working separately on identical projects, presenting the opportunity to evaluate the effects of the work habits captured by CVS on performance. This paper outlines our experiences mining and analyzing these repositories. We extracted various quantitative measures of student behaviour and code quality, and attempted to correlate these features with grades. Despite examining 166 features, we find that grade performance cannot be accurately predicted; certainly no predictors stronger than simple lines-of-code were found.

1. INTRODUCTION

Version control repositories contain a wealth of detailed information about the evolution of a codebase. In this paper, we outline our experiences analyzing data from a large collection of CVS repositories created by many students working on a small set of assignments in a second year undergraduate computer science course at the University of Toronto.

We believe our data set is rather unique. It contains hundreds of completely independent repositories, one for each student. Each student is implementing the same thing at the same time. Previous work analyzing logs from version control systems has tended to focus on a single large repository involving many coders working on different parts of the same software project[5, 4].

1.1 Goals

The broad goal of our research programme was to extract information about student behaviour and code from version control repositories, in order to find statistical patterns or predictors of performance. It was our hope that these results can be used to identify and assist undergraduate students having difficulties. This paper outlines our attempts.

We attempted to identify work habits captured in the CVS repository that are indicative of strong or poor performance. We investigated such hypotheses as *students who start assignments early tend to do well*, and *students who submit assignments close to (or after) the deadline tend to do poorly*. Quantitatively confirming the effectiveness of good work habits could help encourage students to follow them.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

In addition, we attempted to identify features of the code itself that are indicative of performance. As with work habits, confirming their effects on performance could be used to encourage students to write good code.

Finally, we were interested in finding early indicators of students who may be struggling in order to provide timely assistance. We hope to achieve this by finding a way to predict low grades (final course grades in the bottom third of the class) based on statistics extracted from a student's CVS repository.

1.2 CVS Background

CVS, the Concurrent Versions System[1], is a source code management system. It provides a facility for storing past and present versions of a project's codebase, as well as automating many aspects of writing software as a team.

A CVS repository consists of two parts: administrative files stored in a central location (known as CVSROOT), and RCS files associated with each file stored in the repository. The RCS files store revision histories for the individual files comprising the project.

This storage scheme is a bit complex to parse. To further complicate matters, in CVS there are two separate and occasionally disjoint records of activity in a repository: the CVSROOT/history file and the individual RCS files (ending in ,v). File histories are implicitly stored in the RCS files which record modifications, additions, and scheduled deletions. The history file tracks (almost) all interactions between a user and a repository, including check-outs, updates, and conflict resolutions, as well as those listed above. Unfortunately, the history file does not record the initial importing (addition to the repository) of a project.

2. DATA PREPARATION

A combination of Python, ViewCVS[3], and MySQL helped massage the data into a more usable form. Our original code for extracting data from CVS repositories is based on ViewCVS, which includes a fast RCS parser written in C++. This code, including our modifications to the ViewCVS parser, is freely available at www.cs.utoronto.ca/~keir/slurp-1.0.0.tar.gz. The code parses every file in a repository and loads the transactions into a MySQL database for convenient access.

2.1 Transaction Clumping

One problem with CVS is the grouping of transactions. CVS does not keep any record of which operations were executed as part of a single client command. In order to reconstruct the use of the CVS repository, these must be re-grouped based on temporal proximity and other details of the transactions.

We used a variant of the sliding window approach[6] to clump groups of transactions into single events. In this approach, any set

of transactions with the same user and comment string, in which neighbouring transactions occur within τ seconds of each other are grouped into a single event. For our data, we found that $\tau = 50$ worked well. While this disagrees with the results in [6], this is not unexpected, as our repositories are very small, with most students working on the local network, leading to much shorter operation times.

Two modifications were made to the sliding window approach to improve results. First, it was noted that some CVS clients allow the user to enter a different comment for each file involved in a single event. To account for this, all transactions by the same user that occur at the same second were grouped together. Second, it was noted that a single file cannot have multiple modifications within one event. After the clumping was complete, any event which contained the same file more than once was split into separate events, decreasing the amount of over-clumping.

2.2 Feature Extraction

The quantitative and visual analysis techniques we intended to use require numerical data. Accordingly, we converted the known data about each student into a set of numerical summary statistics (called *features*), to be evaluated as predictors of student performance. Our system extracts 166 unique features from the transaction histories, log comments, and details of the actual code files from each student. In order to be able to quantitatively compare the effects of different features, each feature was normalized to have zero mean and unit variance across the entire dataset.

Three classes of features were extracted. The first features were calculated from the database of CVS data. These largely represent student behaviour and work habits. They include things like the average number of revisions per file, number of local CVS operations, number of update transactions, how close to the deadline they submit the assignment, and more.

The second group of features came from parsing the Python and Java code. For these features, each student's repository was checked out and examined. Two simple parsers were written (in Python) to extract features directly from the Python and Java code, such as the number of while loops, number of comments, and number of instances of certain formatting habits.

For the final group of features, we used PMD[2] to examine all Java files. PMD detects many types of higher-level features, particularly style violations or bad practices. It detects things like variable names that don't follow a naming standard, boolean expressions that can be simplified, empty if statements, asserts without a message, and a whole slew of others.

In addition to extracting these features, records of student academic performance were added to the database. Along with grades for each student, each student was labelled as being in the top, middle, or bottom third of the class.

3. VISUAL & QUANTITATIVE ANALYSIS

A variety of visual and quantitative analysis techniques were applied to the data. Visualizations used include views which aggregate statistics across all students and assignments, as well as specialized views which allow us to examine the behaviour of a single student and/or a single assignment. Quantitative analysis techniques used include examining the mutual information between each feature and the student grade, as well as the application of statistical pattern recognition algorithms for predicting grades from the features.

The following sections present the results of our analysis in terms of the three goals of the project: investigating the effects of work habit on grades, the effects of code quality on grades, and attempt-

ing to predict performance based on all of the information available.

3.1 Mutual Information for Feature Evaluation

The mutual information between two (discrete) random variables x and y is defined as the cross-entropy between their joint distribution and the product of their marginal distributions:

$$I(x, y) = KL[p(x, y) || p(x)p(y)] = \sum_{x, y} p(x, y) \log_2 \frac{p(x, y)}{p(x)p(y)}$$

The mutual information between two random variables is a measure of their dependence or independence. It is a more stringent measure than the traditional correlation coefficient, which only measures average second order statistics but cannot capture complex higher order dependencies.

For our data, we did not have enough samples to accurately estimate the joint density between all the features we investigated. However, we did have enough data to estimate the mutual information between a single feature and student grades. Specifically, we created a binary random variable y which was 1 if a student achieved a grade which placed them in the top third of the class and zero if it placed them in the bottom third of the class. Students in the middle third were excluded from the estimation procedure. We then discretized each feature f into $K = 20$ bins (with equal sized ranges f_k) and computed the mutual information between this discretized random variable and the binary grade variable as follows:

$$I(f) = \sum_{y=0,1} \sum_{k=1}^K p(y)p(f \in f_k|y) \log_2 \frac{p(f \in f_k|y)}{p(f \in f_k)}$$

where $p(f \in f_k)$ is the overall observed frequency with which the feature falls into bin k and $p(f \in f_k|y)$ is the frequency for either the high or the low grade students. $p(y = 1) = p(y = 0) = 0.5$ since we select exactly equal numbers of students with high grades (the top third of the class) and low grades (the bottom third of the class).

Using these estimates, we can rank the features by how much information they contain about the course grade. figure 1 gives a short list of the top features and their estimated mutual information. The result was quite surprising to us:

Of the 166 features we examined, only 3 had significant correlation with grade. Of these, the most significant was the total number of lines of code written.

Given the number of student repositories used, only three features had statistically significant mutual information with the grade of the student at the $p = 0.05$ level: lines of code written by the student, number of commas followed by spaces¹ and total length of diff text² (In this case $I = 0.22$ bits was the significance cutoff.) These features are above the line in figure 1.

3.2 Effects of Work Habits on Grades

The first hypothesis investigated was that students who do well on the assignments will tend to do well on the final exam. Figure 2 shows a plot of term grade (from the assignments) versus exam grade, which suggests that such a relationship is present.

We compared student grades with the number of transactions of various types executed by the student, hoping to see that a certain

¹Consider `f○○(a,b,c,d)` instead of `f○○(a, b, c, d)`.

²Each time a student does `cv$ commit`, only changes to the code are stored. The "total diff length" feature is the total character count of all the deltas combined.

M.I. (bits)	Feature Description
0.29	newline characters in students files
0.28	times a space followed a comma, e.g. "foo(a, b, c)"
0.26	characters in diff text between successive revisions (CVS)
0.20	comments (Python)
0.20	literal strings (Python)
0.19	operators (Python)
0.16	characters in all comments
0.16	function definitions (Python)
0.14	while loops (Python)
0.14	Terminal tokens (Python)
0.13	4-space indents
0.13	comment-space-capital sequences e.g. "// Formatted"
0.12	commits (CVS)
0.12	for loops (Python)
0.11	newlines (Python)
0.11	files in repository
0.11	violations of "Assertions should include message" (PMD)
0.11	self references (Python)
0.10	modifies (CVS)
0.10	violations of "Avoid duplicate literals" (PMD)
0.10	except tokens (Python)
0.10	leading tabs
0.10	total transactions (CVS)
0.09	Average revisions per file (CVS)

Figure 1: Mutual information between various features of a student’s repository and the binary indicator of whether they fall into the top third or bottom third of the class (by final grade). Each feature is a *count* of how many times it occurred, for example ‘comments (Python)’ is the number of comments a student had in their code. Features marked (Python) were drawn from the students’ Python code. Likewise, the (CVS) marking means the feature was drawn from the CVS logs, and (PMD) denotes rule violations PMD found. Only values greater than 0.22 bits are statistically significant at the $p = .05$ level, given the number of students. Significant features are located above the horizontal line.

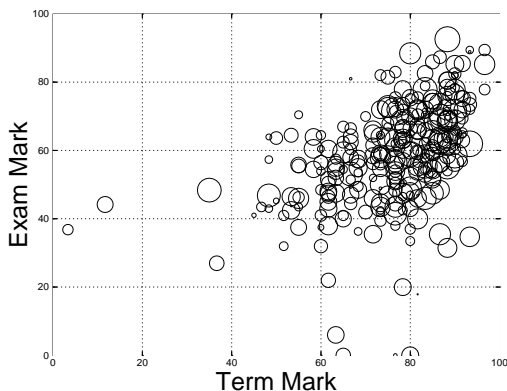


Figure 2: The relationship between the two components making up a final grade. *term mark* is the net mark on the coding assignments. Radius of each circle indicates the total number of CVS transactions executed by the student.

type of transaction (or mix of types) is particularly indicative of performance. The displays, however, suggest that no particular transaction types are indicative of high or low performance (see figure 6 below). This was confirmed by our mutual information analysis which shows that none of transaction type counts have statistically significant mutual information with grade.

Each of the features extracted from the work habits of the student was examined for a relationship with final course grade. While we anticipated several of these would have an impact, it turned out that none had mutual information with final grade that was statistically significant at the $p = 0.05$ level. In particular, we were surprised to note that how early a student starts assignments, and how close to the deadlines they submit, had essentially no predictive value for student grade. (Although some students may have started work on their home machines early but not checked into the CVS repository until the last moment.)

In fact, the only feature drawn from the CVS repository that had statistically significant mutual information with final grade was the total number of characters in the diff text between successive revisions. This, however, is actually an estimate of how much total code the student has written, as opposed to a feature of their work habits.

These results suggest that, contrary to the beliefs of many instructors, student work habits have very little effect on their performance, so long as they eventually do the work. Students who wait until the last minute to do an assignment appear just as likely to do well (or poorly) as those who both start and complete the assignment well ahead of time.

3.3 Effects of Code Quality on Grades

Both visual and quantitative analysis techniques were applied in an attempt to correlate the various features describing code quality with grade.

The feature with the strongest predictive value turned out to be lines of code written, as measured by the number of linefeed characters in the files in each student’s repository. Plots of grade versus lines of code are shown in figure 3. They show informally that students who write very little code tend to do poorly (but beyond a certain point writing more code does not correlate with higher grades) and that (with a few exceptions) students who do well on assignments also do well on the exam. We have also performed a more quantitative mutual information analysis (see below) showing that the number of lines of code written is a statistically significant (though weak) predictor of grade at the $p = 0.05$ level, and is a stronger predictor than any other complex feature we were able to find. Figure 7 provides an alternate view of this relationship, showing histograms of the lines of code written by students in the top third and bottom third of the class, as well as those written by the entire class.

This relationship is not surprising considering that students who do not write enough code to complete an assignment necessarily get low grades. Once a student writes enough code to finish an assignment, lines of code are no longer a strong indicator of quality.

One of the other two features that had statistically significant mutual information with the grade of students at the $p = 0.05$ level was also a code quality measure: the number of times a comma was followed by a space. This indicates care being taken in formatting code, and may well be an indication of the total time spend on the assignment by the student. It is true that some programmers prefer the `foo(a,b)` form; thus, if the code feature indicates the no-space form, no conclusion can be drawn from code formatting habits.

The fact that all three of the statistically significant indicators of performance were indicators of time spent on the assignments

suggests a simple conclusion:

In order to succeed in a course, students should invest the necessary time to complete assignments with care. It doesn't matter when they put this time in, so long as they do so.

3.4 A Machine Learning Approach to Grade Prediction

With numeric features in hand, we were ready to try a variety of statistical pattern recognition algorithms for predicting grades based on the features. Specifically, the algorithms were trained to distinguish between students in the top and bottom third of the class, with those in the middle third left out. Of course, the lack of significant mutual information between grades and most of the features we extracted did not bode well for such an enterprise, but we conducted several experiments nonetheless and report their results here.

We used three very basic algorithms from the machine learning and applied statistics fields: nearest neighbour classification, Naive Bayes, and logistic regression. Before we could classify, we normalized the features to have zero mean and unit variance. (We applied this normalization to all features, even those whose histograms were obviously not Gaussian.)

As expected, none of the classifiers were able to reliably predict grades for students based on the features given. While some algorithms managed to overfit the training sets and achieve 0% training error, the errors on an independently held out test set were always far inferior. A leave-one-out (LOO) cross validation estimate of test error was typically around 25%. In particular, for logistic regression, our LOO error was 29.7%, for Naive Bayes it was 23.9% (using discretized versions of the features), and for nearest neighbour it was also 23.9% (at $K = 21$ using Euclidean distance in the normalized feature space). In all these experiments, we used 166 normalized features to classify 69 students from the top third of the class (by grade) from 69 students from the bottom third.

4. CONCLUSION

We have described the results of analyzing data from a large collection of CVS repositories created by many coders, in this case students, working on a small set of identical projects (course assignments). We have implemented a complete system for parsing such repositories into a SQL database and for extracting, from the database and repositories, various statistical measures of the code and version histories.

Although version control repositories contain a wealth of detailed information both in the transaction histories and in the actual files modified by the users, we were unable to find any measurements in the hundreds we examined which accurately predicted student performance as measured by final course grades; certainly no predictor stronger than simple lines-of-code-written was found.

These results directly challenge the conventional wisdom that a repository contains easily extractable predictive information about external performance measures. In fact, our results suggest that aspects such as student work habits, and even code quality, have little bearing on the student's performance. We are eager to have other researchers suggest novel measures which, contrary to our efforts, contain substantial information about productivity, grades, or performance.

Acknowledgments

We thank Karen Reid, Michelle Craig, and Eleni Stroulia for helpful discussions about the data and analysis tools. STR is supported in part by the Canada Research Chairs program and by the IRIS program of NCE.

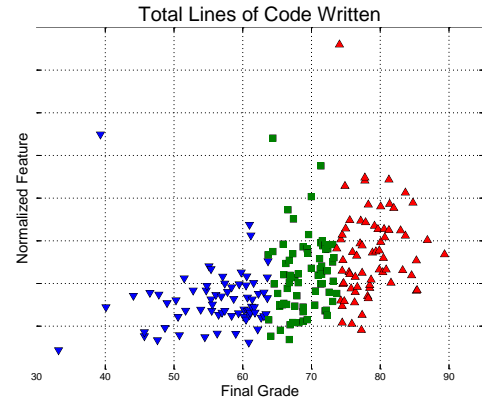


Figure 3: Final grade versus normalized lines of code the student wrote. The correlation visible in this graph between grade and lines of code is as strong as the correlation between grade and any other complex feature we were able to find.

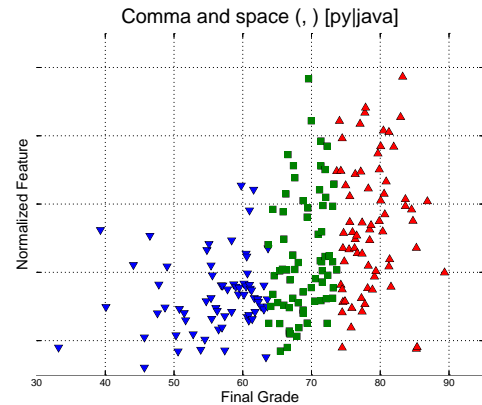


Figure 4: The only other feature to show significant correlation with grade; the number of times a space followed a comma.

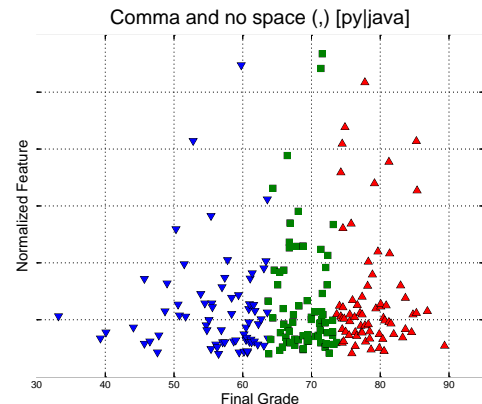


Figure 5: The compliment of the above feature is number of times a comma appears without a subsequent space. From the graph (and the mutual information calculations bear this out) there is very low correlation with grade.

5. REFERENCES

- [1] CVS. <http://www.cvs.org/>.
- [2] PMD: A style checker. <http://pmd.sourceforge.net/>.
- [3] ViewCVS. <http://viewcvs.sourceforge.net/>.
- [4] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. Predicting fault incidence using software change history. In *IEEE Transactions on Software Engineering*, volume 26, July 2000.
- [5] Y. Liu, E. Stroulia, K. Wong, and D. German. Using CVS historical information to understand how students develop software. In *Proc. International Workshop on Mining Software Repositories (MSR04)*, Edinburgh, 2004.
- [6] T. Zimmermann and P. Weiberger. Preprocessing CVS data for fine-grained analysis. In *Proc. International Workshop on Mining Software Repositories (MSR04)*, Edinburgh, 2004.

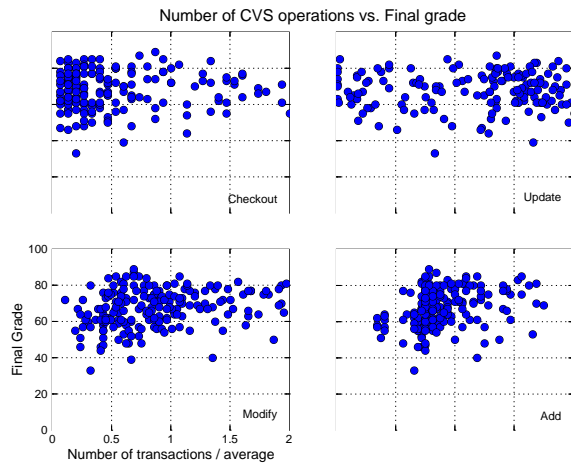


Figure 6: Final grade versus normalized frequency of transactions of various types. (Normalization was done by dividing out the mean.) Each circle represents a single student. The vertical position of the circle in each panel gives the student's final grade in the course while the horizontal position represents a normalized number of transactions of a particular type. Visually, there is no strong correlation present between any of these frequencies and grades; this is confirmed quantitatively by the mutual information analysis in figure 1.

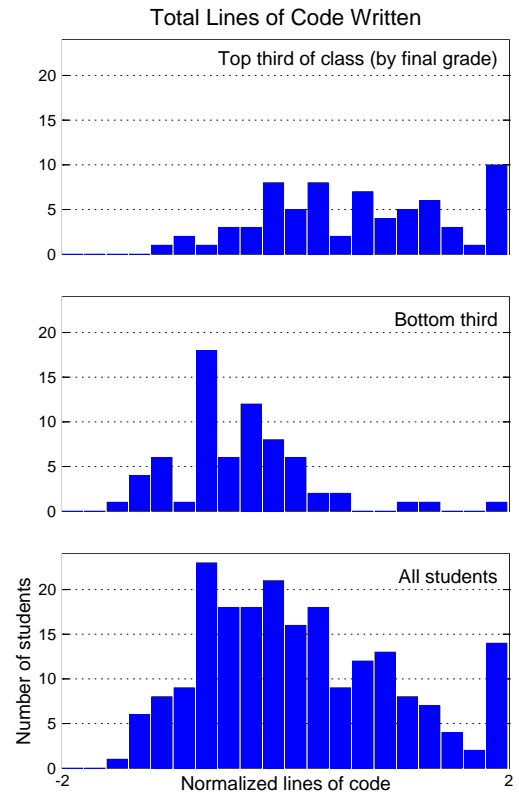


Figure 7: Histograms of (normalized) lines of code written, for the top third (by grade), bottom third, and entire class of 207 students. Visually, it can be seen that students who write more code are more likely to achieve high grades. This feature was the top scoring predictor of grade in our mutual information analysis and is a statistically significant (though weak) predictor of high grade at the $p = 0.05$ level.

Text Mining

Toward Mining “Concept Keywords” from Identifiers in Large Software Projects

Masaru Ohba
Tokyo Institute of Technology
2-12-1 Oookayama Meguro
Tokyo 152-8552, JAPAN

m-ohba @sde.cs.titech.ac.jp

Katsuhiko Gondow
Tokyo Institute of Technology
2-12-1 Oookayama Meguro
Tokyo 152-8552, JAPAN

gondow @cs.titech.ac.jp

ABSTRACT

We propose the Concept Keyword Term Frequency/Inverse Document Frequency (ckTF/IDF) method as a novel technique to efficiently mine *concept keywords* from identifiers in large software projects. ckTF/IDF is suitable for mining concept keywords, since the ckTF/IDF is more lightweight than the TF/IDF method, and the ckTF/IDF’s heuristics is tuned for identifiers in programs.

We then experimentally apply the ckTF/IDF to our educational operating system `udos`, consisting of around 5,000 lines in C code, which produced promising results; the `udos`’s source code was processed in 1.4 seconds with an accuracy of around 57%. This preliminary result suggests that our approach is useful for mining concept keywords from identifiers, although we need more research and experience.

Keywords

concept keywords, program understanding, identifiers, TF/IDF

1. INTRODUCTION

Many programmers make all possible effort to make their identifiers both concise and descriptive enough to suggest their role in a program (for example, “`read_dirent()`” in `udos`[10], as opposed to “`f()`”). Fortunately, such descriptive identifiers provide a wealth of information which can aid in program understanding. From the previous example, `dirent` implies “directory entry” - a key concept in understanding the FAT file system[2]. Thus, we see the benefits for program understanding that mining such terms can bring. In this paper we present a tool which can mine such key concepts, called *concept keywords*, from identifiers present in source code.

Concept keywords, for the most part, contribute to program understanding in three ways.

- Concept keywords highlight important parts in source code and implicit relations among them.

In program understanding, not all code fragments are equally important, and important parts are unequally distributed in source code. Also important parts change depending on your concern. By using, for example, text editor highlighting¹

¹For example, `highlight-regexp.el` for the Emacs editor.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MSR’05, May 17, 2005, Saint Louis, Missouri, USA

MSR’05 Saint Louis, Missouri USA

Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

for concept keywords in your concern (e.g., `dirent`), you would quickly find important parts in source code.

Also concept keywords help you to find implicit relations in source code. For example, by searching `dirent`, you can find a comment like “/* `FAT12_read` is used for sequential access to directory entries, while `read_dirent` for random access */”. This relation between `read_dirent` and `FAT12_read` is implicit in the sense that there is no control dependence nor data dependence between them.

- Concept keywords bridge the gap in understanding between source code and specifications, caused by abstraction mismatch.

By relating the parts with a same concept keyword, you can find the corresponding descriptions more efficiently. For example, when you find a function “`read_dirent`” in source code, you can imagine the function reads a “directory entry”, and know its structure by searching “directory entry” and “dirent” in FAT specification[2]. Of course concept keywords do not work well for vocabulary mismatch (e.g., the function name `read_folder_entry` for reading a directory entry); we assume most programmers try to avoid such vocabulary mismatch in naming identifiers.

- Identifiers have an good affinity for software repositories and a technique for mining concept keywords can also be applied to large software repositories.

This is because the characteristics of a typical software repository such as the combination of version control system, mailing list system and bug tracking system are almost the same as those of identifiers; both of them are language independent, text-based, machine-processable in a lightweight manner, and so on.

Many program understanding tools are already available such as call-graph extractors, cross-referencers, slicers, outlining tools, source code browsers, beautifiers, code metrics tools, documentation tools, debuggers, profilers, etc. However, none of these tools can mine concept keywords. Our goal is to develop a novel tool for mining concept keywords from identifiers.

Unfortunately, it is very challenging to efficiently and accurately mine concept keywords from identifiers in large software since concept keywords are hidden within numerous identifiers in a somewhat “unexpected” manner; thus we need to use a lightweight heuristic mining algorithm suitable for identifiers. Existing and well-known mining algorithms such as the TF/IDF weighting method[16] does not work well for identifiers, since the characteristics of concept keywords and identifiers are quite different from those of natural languages. For example, identifier prefixes such as `kbd_` (meaning “keyboard”), are often used to group strongly related identifiers, but this does not occur in natural language. The TF/IDF method gives high scores to prefixes which are not concept keywords, resulting in inaccurate mining.

Moreover, existing algorithms may prove too heavy for mining concept keywords. Software is continually getting larger and more complex - GCC (GNU Compiler Collection) has over 400,000 lines in C, and requires over 30 minutes to build². Furthermore, reducing the time-to-market is of paramount importance in today's software development projects. Hence programmers require efficient mining tools and need to find a new technique suitable for mining concept keywords.

In this paper we propose the Concept Keyword Term Frequency and Inverted Document Frequency (ckTF/IDF) method to efficiently and accurately mine concept keywords. Our basic ideas are:

- ckTF/IDF is a very lightweight method, obtained by simplifying TF/IDF scoring.
- ckTF/IDF is tuned for identifiers. For example, ckTF/IDF excludes meaningless prefixes with a high accuracy.

To see how this idea works in practice, we experimentally applied ckTF/IDF method to our educational operating system `udos`[10] (about 5,000 lines in C). `udos` is selected as a testbed, since:

- We are familiar with the `udos` source, since one of the authors (Gondow) developed `udos`. Thus we can examine the result of the experiment using our knowledge about `udos`.
- Concepts in operating systems can be enumerated from OS text books and specifications such as POSIX, hardware manuals, etc.
- `udos` is a relatively small operating system, yet complicated enough to realize the importance of concept keywords.

As a result, ckTF/IDF processed `udos`'s source code in 1.4 seconds with an accuracy of around 57%. This preliminary result suggests that our approach is helpful for mining concept keywords from identifiers, although we need more research and experience.

This paper is organized as follows. Section 2 describes the characteristics of concept keywords, and the difficulty of mining them. Section 3 introduces our new ckTF/IDF method. Section 4 explains our experimental implementation of the framework for the ckTF/IDF. Section 5 describes our preliminary experiment of applying ckTF/IDF to `udos` and its results. Section 6 describes related work. Section 7 gives our conclusions and suggestions for future work.

2. CONCEPT KEYWORDS

This section describes the characteristics of concept keywords, and the benefits and difficulty of mining them.

2.1 What is a concept keyword?

In Section 1, we mentioned a concept keyword is a word that represents a key concept in program understanding, and `dirent` (directory entry) is an instance of concept keywords. So what is the definition of concept keywords? Unfortunately there is no clear definition of concept keywords, since they are highly based on subjective judgement. To make our discussion clear, we use the following three terms for concept keywords.

- *Ideal concept keywords*, which have proven to improve program understanding by some objective measurements.
- *Human-selected concept keywords*, which a developer or reviewer believes are ideal concept keywords.
- *Machine-extracted concept keywords*, which a method like TF/IDF produced as an approximation of ideal or human-selected ones.

²By GCC-3.4.3 on Pentium 4(supported HT) 2.6GHz, 512MB RAM, Linux-2.6.8-1-686-smp

For example, `dirent` is a human-selected concept keyword in the sense that we just judged so. Although it is still unknown due to the lack of good metrics whether `dirent` is an ideal one or not, our software development experience suggests a hypothesis that concept keywords exist as a small subset of words in identifiers.

Table 1 shows the number of all human-selected concept keywords that we selected from `udos`'s source code, along with those of words in other categories. Table 1 tells around 22% (= 61/279) of words in identifiers are human-selected concept keywords, which supports the above hypothesis.

Hereafter, in this paper, we often use the term *concept keyword* for referring to a *human-selected concept keyword*. In Section 5, *machine-extracted concept keywords* by ckTF/IDF and TF/IDF are compared with *human-selected concept keywords* in Table 1.

2.2 Why concept keywords?

In this section, we explain why mining concept keywords from identifiers has a great potential to dramatically improve program understanding.

Program understanding is the most important activity in software development and software maintenance. In [12], for example, it is estimated that "some 30-35% of total life-cycle costs are consumed in trying to understand software after it has been delivered, to make changes". Thus improving program understanding is a key issue in software engineering.

To alleviate this problem, as mentioned in Section 1, many program understanding tools have already been developed and researched such as call-graph extractors[11], cross-referencers, slicers, outlining tools, etc. Although many successes have been achieved in individual areas by these tools, the cost of program understanding still remains high.

This reason is twofold. One reason is inaccuracy in tools. An empirical study [14], for example, reported that call graphs extracted by several broadly distributed tools vary significantly enough to surprise many experienced software engineers. The other reason is the limitations of tools. For example, even ideal call-graph extractors cannot cover the whole range of information required in program understanding. This observation leads us to the necessity of developing yet another kind of tools, and of discovering some ways to well combine them with the existing tools.

Our idea of mining concept keywords provides a new kind of support for program understanding. As mentioned in Section 1, concept keywords can contribute to programming understanding in various ways. Also it is easy to combine our mining technique with the existing tools like version control system, mailing list system and bug tracking system, since both of them have the same characteristics of being language independent, text-based, machine-processable in a lightweight manner. Thus, the idea of mining concept keywords seems very attractive and can have a great potential to improve program understanding. As far as we know, however, little work has been done so far for mining concept keywords in source code.

2.3 Why difficult to mine concept keywords?

As mentioned in Section 1, however, existing and well-known mining algorithms such as the TF/IDF weighting method do not work well for identifiers, since the characteristics of concept keywords and identifiers are quite different from those of natural languages, and since existing algorithms can be too expensive for mining concept keywords.

In order to accurately mine concept keywords, we feel it is necessary for users to experiment with parameters (such as the term frequency threshold) - hence, a lightweight algorithm is preferred over a more expensive one in order to enable frequent experimentation.

Hence, we intrude ckTF/IDF, an algorithm based on TF/IDF which is less expensive without sacrificing quality of keywords mined. The differences between the two algorithms will be shown in Section 3.3 and 5.

Table 1: Human-selected concept keywords and other category words in udos

	category	#	examples	description
1.	concept keywords	61	dirent, root, PTE, tss, path, signal, yield	helpful key concepts for program understanding
2.	grouping words	18	kbd_, vga_, FAT12_, sys_, FDC_, RTC_, console_, H_, t	prefixes and suffixes for grouping functions and variables, or for other purposes
3.	attributes, and less important concepts	70	busy, byte, offset, name, memory, end, int8, again	general nouns and adjectives used as attributes, modifiers, etc., being less informative in themselves.
4.	generic verbs	130	read, set, is, move, wait, print, dump, make, init	generic verbs to describe actions or operations; the same names are commonly used for unrelated functions

3. ckTF/IDF METHOD

In this section, we propose the *Concept Keyword Term Frequency and Inverted Document Frequency* (ckTF/IDF) method to efficiently mine concept keywords.

3.1 Overview of TF/IDF method

Before we define the ckTF/IDF method, this section gives an overview of the TF/IDF method, which ckTF/IDF is based on. The TF/IDF is a method for mining characteristic terms and document classification. The key feature of TF/IDF heuristics is to give a high score (i.e., high term weight) as a characteristic term if the term appears more frequently in a particular document and less in other documents. The weight of a term in a document is calculated by its term frequency (TF) and its inverse document frequency (IDF) in all the documents.

The inverse document frequency $idf(t)$ for a term t is defined as follows.

$$idf(t) = \log \frac{N}{df(t)} \quad (1)$$

where $df(t)$ is the number of documents including the term t , N the number of all documents.

The term weight $w(t, d)$ for a term t and a document d is defined as follows.

$$w(t, d) = tf(t, d) \cdot idf(t) \quad (2)$$

where $tf(t, d)$ (meaning the term frequency) is a count of occurrence of term t in document d .

Thus keywords can be mined from documents in natural languages by picking terms with high weights calculated by TF/IDF.

3.2 Definition of ckTF/IDF method

This section gives the definition of the ckTF/IDF method. The ckTF/IDF treats one source file as one document, since most source code written in C are expedient granularity for mining concept keywords from our experience. For example, `dirent` and `root` of concept keywords in FAT file system appear in `udos's fat12.c` only, and not appear in others.

The ckTF/IDF is a very simplified and thus speeded-up version of TF/IDF by quantizing $tf(t, d)$ and $idf(t)$ into 0 or 1. The $idf(t)$ for ckTF/IDF is defined as follows.

$$idf(t) = \begin{cases} 1 & \text{if } 1 \leq df(t) \leq n \text{ and } \neg \text{is_prefix}(t) \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

where $df(t)$ is the same as in TF/IDF, $n(\geq 1)$ the threshold (default is 1) to quantize $tf(t, d)$ and $idf(t)$, and $\text{is_prefix}(t)$ a predicate being true if and only if t is a prefix for all its occurrences.

The term frequency $tf(t)$ in all documents and the word weight $w(t)$ for ckTF/IDF are defined as follows.

$$tf(t) = \begin{cases} 1 & \text{if } \exists d, tf(t, d) > n \\ 0 & \text{otherwise} \end{cases} \quad (4)$$

$$w(t) = tf(t) \cdot idf(t) \quad (5)$$

In ckTF/IDF, $w(t) = 1$ implies the term t is characteristic, so t is selected as a (machine-extracted) concept keyword.

The value of $w(t)$ can be computed very fast by using the two flags: *local frequency flag* and *global frequency flag* for each term t ($local(t)$ and $global(t)$ for short, respectively). First all identifiers in source code are divided into terms by some delimiters like underscores. Then two flags are computed for each term, considering a language construct for grouping (e.g., a compilation unit for the programming language C) as a document. When a term t is found twice in a document, $local(t)$ is set without performing the remaining computation. Similarly, when t is found in two documents, $global(t)$ is set without performing the remaining computation. Note that $local(t)$ and $global(t)$ are not exclusive; both can be set at the same time. If $local(t)$ is set, $global(t)$ is clear, and the term is not a prefix, then $w(t)$ becomes 1. Otherwise, $w(t)$ becomes 0.

Thus ckTF/IDF realizes a lightweight way of mining concept keywords. The computational complexity of ckTF/IDF and TF/IDF is discussed in Section 3.3. Some actual measurements of their performance are shown in Section 5.

3.3 ckTF/IDF vs. TF/IDF

3.3.1 Characteristics of ckTF/IDF

For most cases, $w(t) = 1$ for ckTF/IDF when $\max_d w(t, d)$ for TF/IDF is high, and $w(t) = 0$ when $\max_d w(t, d)$ is low. Thus there is a high correlation between ckTF/IDF and TF/IDF. There are two exceptions for this.

- ckTF/IDF imposes a penalty for prefix terms, while TF/IDF not. This penalty is introduced to ckTF/IDF, since prefixes like `kbd_` (meaning “keyboard”) are not likely to be concept keywords from our experience, and also since this penalty can be computed fast.
- When two flags are set at the same time, ckTF/IDF’s score is always 0, while TF/IDF’s score varies. This can be the cause of inaccuracy of ckTF/IDF, when $tf(t, d)$ is very large or $df(t)$ is small but greater than 1. This can be alleviated by adjusting the threshold n in Equation (3) and (4).

3.3.2 Computational complexity

For computing all $w(t)$ and $\max_d w(t, d)$ from the scratch, computational complexity of both ckTF/IDF and TF/IDF are the same as $O(|D| + |T|)$ where D is a set of all documents and T a set of all terms, although ckTF/IDF is much faster than TF/IDF in practice (see also Section 5).

The difference arises when computing them incrementally. Suppose that we add a set of documents ΔD including a set of terms ΔT . The complexity for ckTF/IDF is $O(|\Delta D| + |\Delta T|)$, while that for TF/IDF is $O(|D + \Delta D| + |T + \Delta T|)$.

4. DESIGN AND IMPLEMENTATION

This section gives a brief description of the design and implementation of Identifier Exploratory Framework (IEF), which we experimentally developed as a framework for the ckTF/IDF method (IEF’s source code is available in [15]). Figure 1 shows the overview

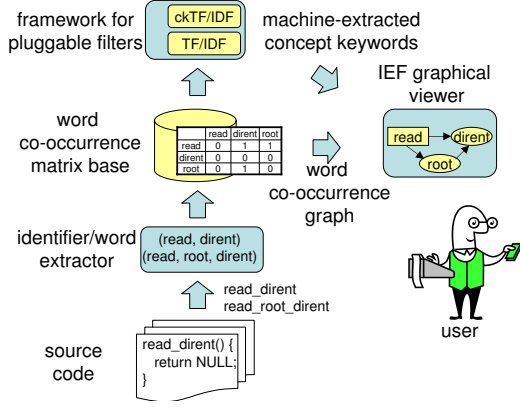


Figure 1: Components of Identifier Exploratory Framework (IEF)

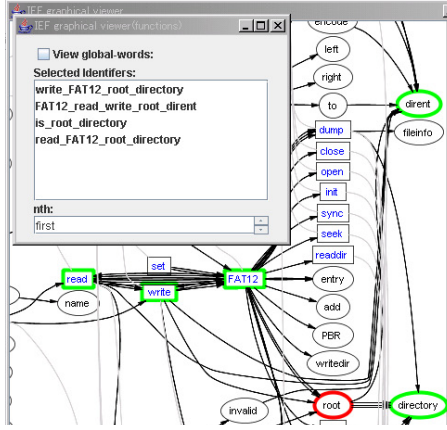


Figure 2: Screen snapshot of IEF graphical viewer

of IEF. Figure 2 shows a sample screen snapshot of IEF’s GUI, which displays a co-occurrence graph for words in `udos’s fat12.c`. The components of IEF are:

- **Identifier/word extractor** - extracts all definitions of function and global variables from a given source code by utilizing a cross-referencer GNU GLOBAL[3], and then tokenizes their names into words. Note that uses of functions/variables are not extracted to avoid the increase of document frequency for important global functions. It simply uses underscore (.) as delimiter to avoid heavyweight processing like morphological analysis or use of dictionary. Thus it is lightweight, although it cannot tokenize some identifiers like `kmalloc` (meaning “kernel memory allocation”). It consists of around 300 lines in Ruby script language[5] and some C code for GNU GLOBAL.
- **Word co-occurrence matrix base** (i.e., word corpus): is a simple database that stores all words extracted from identifiers, along with co-occurrence information in identifiers and their filenames.
- **Framework for pluggable filters**: is the core feature of IEF. It provides the extensibility for implementing additional filters, and also provides interface for filters to access the word co-occurrence matrix. Currently only the filters for `ckTF/IDF` and `TF/IDF` are available, which consist of around 400 lines in Ruby.
- **IEF graphical viewer**: allows users to browse the word co-occurrence graph and the output of the filters. IEF graph-

Table 2: Total # of word occurrences by position for `udos`

	category	word position		
		first	last	middle
1.	concept keywords	14	21	75
2.	grouping words	141	20	33
3.	attributes	78	17	199
4.	generic verbs	23	38	41
	total	256	96	348

ical viewer is implemented using Grappa graph-drawing library[4]. It consists of around 500 lines in Java.

Figure 2 shows a screen snapshot of IEF graphical viewer, which displays a co-occurrence graph of words in `fat12.c` of `udos`. In the graph, a rectangle node implies a term t such that $global(t) = 1$, an oval node implies a term t such that $global(t) = 0$, and an edge implies the two end nodes of the edge co-occurs in a same identifier. At the first time we saw the co-occurrence graph, we soon found that human-selected concept keywords like `dirent` or `root` almost correspond to non-global nodes with many edges in the graph. This experience actually motivated us to start this research.

5. PRELIMINARY EXPERIMENT

To see how `ckTF/IDF` works in practice, we experimentally applied the `ckTF/IDF` and `TF/IDF` to our educational operating system `udos’s` source code[10]. This section gives the results of this experiment.

5.1 Accuracy and Coverage of `ckTF/IDF`

In this paper, we use the measures of accuracy and coverage to evaluate the performance `ckTF/IDF` and `TF/IDF` by comparing the concept keywords extracted by a human programmer with that of the algorithm. Accuracy and coverage correspond to a measure of precision and recall, respectively. We define $Accuracy = \frac{C_r}{C_m}$ and $Coverage = \frac{C_r}{C_h}$, where C_m is the total number of all machine-extracted concept keywords, C_h is the total number of human-extracted concept keywords, and C_r is the total number of concept keywords that both human and algorithm have chosen.

Figure 3 shows accuracy and coverage when applying `ckTF/IDF` the codebase of `udos`. We found an accuracy of 57% ($=16/28$) and a coverage of 26% ($=16/61$) and that the accuracy in mining concept keywords increases by removing unnecessary words, those which are not considered to be of less significance as defined in “(c)” and “(d)” in Figure 3.

In contrast, the accuracy and coverage of `TF/IDF` for `udos` are 28% ($=8/28$)³ and 13% ($=8/61$), respectively (not shown in any figure). Thus, as far as this experiment is concerned, `ckTF/IDF` delivers twice better accuracy and coverage than `TF/IDF`.

5.2 Removing Prefixes Improves Accuracy?

In Section 3.3.1, we mentioned `ckTF/IDF` imposes a penalty for prefix terms under the hypothesis that prefixes are unlikely to be concept keywords. Table 2 supports this hypothesis, which tells that concept keywords rarely occur in the first position (around 5% = 14/256), while grouping words often occur there (around 55% = 141/256).

So does the heuristics of `ckTF/IDF` work? The answer is subtle. Figure 3’s (a) and (b) explain this subtlety. (a) is the result with the prefix penalty, while (b) is the result without it. By using the prefix penalty, the accuracy increased to 57% (7% up), but the coverage decreased to 26% (5% down).

³`TF/IDF` outputs word weighting for all entries as results, while `ckTF/IDF` outputs only candidates of concept keywords. Therefore, in order to compare `ckTF/IDF`’s results with that of `TF/IDF`, we limited the comparison to the first n candidates, where n is the number of candidates returned by `ckTF/IDF`.

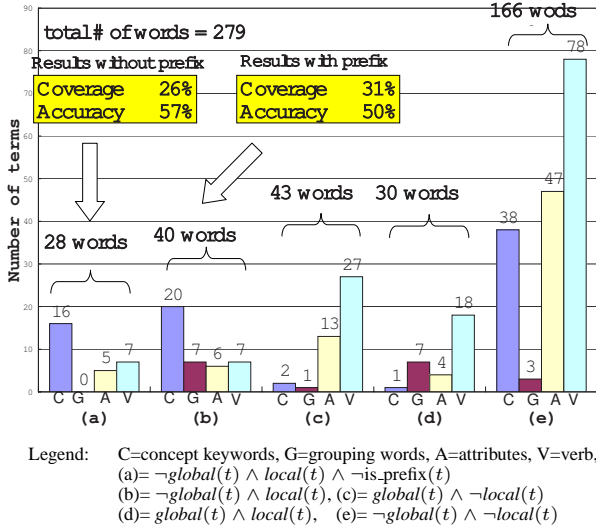


Figure 3: Accuracy and coverage of ckTF/IDF for uDOS

5.3 Performance of ckTF/IDF and TF/IDF

As mentioned before, ckTF/IDF processed uDOS's source code in 1.4 seconds (including file I/O time). uDOS (around 5,000 lines in C), however, is too small to compare the performance, so we used the Ruby interpreter instead of uDOS. Ruby consists of around 48,000 lines in C (excluding blank lines), and has 3,385 identifiers.

Table 3 shows a comparison of the execution speeds⁴ of ckTF/IDF and TF/IDF for the Ruby interpreter. "Computation from scratch" in Table 3 means the term weight computation of the whole source code of the Ruby interpreter, and "incremental computation" means re-computation of the above results after a document including 7 words is added. As far as the results are concerned, ckTF/IDF is about 6 times faster than TF/IDF. Note that "0 sec" in Table 3 means a very small amount of time, not really zero.

6. RELATED WORK

To our knowledge, little work has been done so far for mining concept keywords in program identifiers.

Caprile et al.[8, 9] proposes an identifier restructuring tool, which uses a semiautomatic technique for the restructuring of identifiers, and enforces a standard syntax for their arrangement. They consider identifiers as an important tool for programming understanding, but their research is not for mining concept keywords in program identifiers.

Anquetil[6] attempts manually mining concepts from program identifiers and comments. He applied his manual technique to the Mosaic system, relating, for example, `xm` and `xmx` to the X Window System. His experiment took quite a long time (around 30 hours), while our mining by ckTK/IDF took only around 1.4 seconds to automatically process uDOS's source code.

Anquetil et al.[7] proposes a technique for extracting concepts from the abbreviated filenames (such as "dbg" for debug or "cp" for call processing). Although they achieved a high accuracy (80% to 85% of abbreviations found when used with an English dictionary), their technique seems too heavy for mining identifiers in large software projects.

Knuth[13] tries to achieve better program understanding by integrating both of source code and documents using the WEB language. In contrast, our aim is to achieve better program understanding for large open source code like GCC, not written in WEB, by mining concept keywords from program identifiers.

7. CONCLUSION

⁴File I/O time is excluded.

Table 3: Execution speeds of ckTF/IDF and TF/IDF for Ruby interpreter (in elapsed time)

	ckTF/IDF	TF/IDF
computation from scratch	0.24 sec	1.57 sec
incremental computation	0 sec	0.44 sec

We have proposed the Concept Keyword Term Frequency/Inverse Document Frequency (ckTF/IDF) method as a novel technique to efficiently mine *concept keywords* from identifiers in large software projects. Testing the algorithm using the source code of uDOS (5,000 lines of C), we found that it was processed in 1.4 seconds with an accuracy of around 57% and coverage of around 26%. Coverage of around 26% is not necessarily high, although the ckTF/IDF method is extremely lightweight and high in accuracy. We assume that the coverage can be increased by incorporating approaches used in other mining algorithms.

This preliminary result suggests that our approach is helpful for mining concept keywords from identifiers, although we need more research and experience.

Our future works include: (1) to apply our mining technique to large practical software like GCC or Apache and to provide comprehensive evaluation, (2) to apply concept keywords and/or the ckTF/IDF method to a Bug Tracking System (BTS) like bugzilla[1] to relate keywords in bug reports to the corresponding source code,

8. REFERENCES

- [1] Homepage for bugzilla.
<http://bugzilla.mozilla.org/>.
- [2] Homepage for FAT32 file system specification.
<http://www.microsoft.com/whdc/system/platform/firmware/fatgen.mspx>.
- [3] Homepage for GNU GLOBAL.
<http://www.gnu.org/software/global>.
- [4] Homepage for Grappa.
<http://www.research.att.com/~john/Grappa/>.
- [5] Ruby language homepage.
<http://www.ruby-lang.org>.
- [6] Nicolas Anquetil. Characterizing the informal knowledge contained in systems. In *WCRC: Proc. 8th Working Conf. on Reverse Engineering*, pages 166–175, 2001.
- [7] Nicolas Anquetil and Timothy Lethbridge. Extracting concepts from file names: a new file clustering criterion. In *ICSE '98: Proc. 20th Int. Conf. on Software Engineering*, pages 84–93. IEEE Computer Society, 1998.
- [8] Bruno Caprile and Paolo Tonella. Nomen est omen: Analyzing the language of function identifiers. In *WCRC '99: Proc. 6th Working Conf. on Reverse Engineering*, page 112. IEEE Computer Society, 1999.
- [9] Bruno Caprile and Paolo Tonella. Restructuring program identifier names. In *ICSM: Int. Conf. on Software Maintenance*, pages 97–107, 2000.
- [10] K. Gondow. Homepage for an educational operating system uDOS.
<http://www.sde.cs.titech.ac.jp/~gondow/ud0s/>.
- [11] K. Gondow, T. Suzuki, and H. Kawashima. Binary-level lightweight data integration to develop program understanding tools for embedded software in c. In *Proc. 11th Asia-Pacific Software Engineering Conference (APSEC)*, pages 336–345, 2004.
- [12] P. A. V. Hall. Overview of reverse engineering and reuse research. *Information and Software Technology*, 34(4):239–249, 1992.
- [13] Donald E. Knuth. *Literate Programming (Center for the Study of Language and Information - Lecture Notes, No. Van Nostrand Reinhold Computer*, 1989.
- [14] G.C. Murphy, D. Notkin, and E.S.-C. Lan. An empirical study of static call graph extractors. In *Proc. 18th Int. Conf. on Software Engineering (ICSE-18)*, pages 90–99, 25–29 Mar 1996.
- [15] M. Ohba. Homepage for the concept keyword mining tool.
<http://www.sde.cs.titech.ac.jp/~m-ohba/cktfidf/>.
- [16] Gerard Salton and Christopher Buckley. Termweighting approaches in automatic text retrieval. *Information Processing and Management*, Vol. 24(5), 1988.

Source code that talks: an exploration of Eclipse task comments and their implication to repository mining

Annie T.T. Ying, James L. Wright, Steven Abrams
IBM Watson Research Center
19 Skyline Drive, Hawthorne, NY, 10532, USA
{aying,jimwr,sabrams}@us.ibm.com

ABSTRACT

A programmer performing a change task to a system can benefit from accurate comments on the source code. As part of good programming practice described by Kernighan and Pike in the book *The Practice of Programming*, comments should “aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings.” In this paper, we explore the widely varying uses of comments in source code. We find that programmers not only use comments for describing the actual source code, but also use comments for many other purposes, such as “talking” to colleagues through the source code using a comment “Joan, please fix this method.” This kind of comments can complicate the mining of project information because such team communication is often perceived to reside in separate archives, such as emails or newsgroup postings, rather than in the source code. Nevertheless, these and other types of comments can be very useful inputs for mining project information.

1. INTRODUCTION

Accurate comments on source code can be useful to a programmer performing a change task. As Knuth suggested in the literate programming technique, programs should not only be intended to be executed by computers, but also intended to be read by human [4]. As part of good programming practice, Kernighan and Pike suggested that programmers should write comments that “aid the understanding of a program by briefly pointing out salient details or by providing a larger-scale view of the proceedings” [3].

Many programmers use comments for purposes other than describing source code, but yet these comments contain very useful information to a programmer performing a change task. One example of such a kind of comments is the Eclipse task comments [1]. Since March 2003, Eclipse—a popular open-source integrated development environment—has provided support for comments that describe tasks to be performed on the source code through the task tag mechanism.

Using the Java perspective in Eclipse, Java programmers can embed pre-defined task tag strings, such as “TODO”, in the comments on the source code and use the task view to browse a summary of the places in the code with a comment that contains a task tag. From the task view, a user can click on an entry and navigate to the corresponding source code.

In this paper, we perform an informal empirical study on the use Eclipse task comments in Java source code. As a preliminary study, we look at an IBM internal codebase, the Architect’s Workbench (AWB). We found that although many of these comments do not describe the actual source code, they describe other interesting development aspects, such as communication and changes that were performed or to be performed to the source code. For example, some developers “talk” to colleagues through the source code using a comment such as “Joan, please fix this method.” Such kinds of comments can complicate the mining of project information because such team communication and task-oriented information is often perceived to reside in separate archives, such as emails or change request management systems, rather than in the source code. In addition, these comments typically contains ad-hoc meta-data, depends on the context of the code, have a implied scope, and are informal.

The rest of the paper is organized as follows: first, in Section 2, we present a categorization of Eclipse task comments from our study on the AWB codebase. In Section 3, we describe the challenges of analyzing task comments in the context of mining project information. In Section 4, we discuss some issues with our study. Finally, in Section 5, we conclude and outline future work.

2. TASK COMMENTS CATEGORIZATION

To explore what information Eclipse task comments contains and what they are intended for, we studied the Eclipse task comments that were in the AWB code checked out from the AWB CVS repositories on February 9, 2005. The codebase consists of 2,213 files. The code contains 221 task comments¹.

The AWB project consists of two major parts: a platform that provides customizable representations and tool support for models, and a particular instantiation of this platform in the system architecture domain, which embodies a tool that helps IT architects transform informal notes into various formal system architecture models. The source code of AWB is written primarily in Java and is implemented as an Eclipse

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR’05 May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

¹We define the number of task comments as the number of lines of Java comments that contain the string “TODO”.

plug-in.

Five developers contributed to the task comments in the version of the AWB code we studied. To preserve the privacy of the developers, whenever we paraphrase a comment from the AWB codebase, we have substituted the name of a developer in a comment with a made-up name—Beth, Joan, Pam, Rea, or Sue.

In the AWB codebase, we found different uses of Eclipse task comments. We categorized these different uses, as shown in Table 1. The first column shows the categories, each of whose cell belongs to one of the seven main groups: communication, past tasks, current tasks, future tasks, pointers to a change request, location markers, and concern tags. The second column presents an example of Eclipse task comment found in the AWB code. Some comments belong to multiple groups, for example, a comment that is both for communication and for describing a task. For the rest of this section, we describe the seven categories of comments and present an examples of comment from each categories.

Each of the sub-sections in the rest of this section describes a main category and the examples listed in Table 1.

2.1 Communication

We found some cases where developers use the source code as a medium to communicate to each other.

- In the example labelled “communication: point-to-point” in Table 1, Sue wrote an Eclipse task comment dedicated to Joan. Prior to this message, Sue and Joan had actually discussed the error that was fixed by the hack referred in the comment. In their discussion, Sue suggested the hack. Although Joan was not satisfied with the hack, Joan could not come up with a better fix. Because of the urgency to get the bug fixed, Sue just temporarily implemented the hack. To remind Joan to better fix the error, Sue wrote this comment.
- In the example labelled “communication: multi-cast/broadcast” in Table 1, Joan may have intended to only direct this question to Sue, the implementer of the method referred in the comment. However, this question may worth directing to other team members who may be thinking to call this method and thus may advocate against making the method non-public.
- In the example labelled “communication: self-communication” in Table 1, the example serves as a reminder to Pam herself to clean up the tracing statements in the code.

2.2 Pointers to change requests

Some Eclipse task comments denote a task that is part of a bigger change logged in the change tracking system. AWB uses their own change tracking system called the ECR (Enhancement Change Request) system.

- In the example labelled “pointer to a change request” in Table 1, Pam wrote the two comments to redirect further details to the change report ECR with ID 327. Because an Eclipse task comment is in a particular location in the code, it often denotes a finer-grained task that a task logged into the change tracking system.

2.3 Bookmarks on past tasks

We found in the AWB that some comments describe changes that had been completed. These comments often denote places where changes to the code may require further work.

- In the example labelled “bookmark: hack” in Table 1, which is the same example as an example we described in Section 2.1, Sue indicated that she had performed a code modification which was a hack.
- Another example shows that Eclipse task comments are used to mark places in the code where the developer is uncertain about whether the change actually fixed the defeat. In the task comment labelled as “bookmark: experimental fix” in Table 1, Joan wrote this same comment in several places in the code. Although she has completed a fix to a threading problem, she is not totally confident that fix actually solves the problem until the system has been used for a while. Therefore, she marked use this comment to mark the places that contributed to the fix.

2.4 Current tasks

Most of the comments in the AWB code denotes outstanding tasks that need to be done currently.

- In the example labelled “current task: refactoring” in Table 1, Pam uses a comment to suggest refactoring, briefly outlining the current strategy and the suggested strategy.
- Another example of a current task is a task comment generated by the Eclipse code generator. When using Eclipse to generate a Java class from a super-class or an interface, Eclipse automatically inserts a “TODO” comment for the generated methods and constructor stubs, as demonstrated in the example labelled “current task: from automatically generated code” in Table 1. Eclipse also generates a “TODO” comment for an empty Java `catch` block when Eclipse “Encode try-catch block” functionality is used to generate a `catch` block.

2.5 Future tasks

Some tasks cannot be done currently because those tasks depend on something to be available in the future:

- In the example labelled “future task: once the library is available...” in Table 1, Pam cannot proceed with the task of using the “Eclipse-icon-Decorator” mechanism in the code depends on the availability of that mechanism.
- Similarly, in the example labelled “future task: once some code modification is complete” in Table 1, the developer cannot perform the task until ECR 317 is complete.

2.6 Location markers

All tasks comments are location markers – the Eclipse task view enables a developer to easy view and navigate to the places in the code with task comments:

Categories	Example
communication: point-to-point	// TODO an ugly hack for now -sue. Joan, please fix it
communication: multi-cast/broadcast	// TODO joan: explain why this [method] is public, since it is used only internally
communication: self-communication	// TODO [...] remove tracery if cell-editing is ever stable
pointer to a change request	RichAttributeComparison.java: // TODO pam: ECR 311: get copy-text button to work AttributeViewerImpl.java: // TODO pam: ECR 311: handle the case of multiple Node-*types*
bookmark: hack	// TODO an ugly hack for now -sue. Joan, please fix it
bookmark: experimental fix	// TODO joan EXPERIMENTAL
current task: refactoring	// TODO [...] make this work using subtyping, not parsing the String type-name!
current task: from automatically generated code	// TODO Auto-generated method stub
future task: once the library is available...	// TODO pam: once we have the Eclipse-icon-Decorator mechanism, use it here
future task: once some code modification is complete ...	// TODO [...] eliminate this once ECR 317 complete
location marker: point location	// TODO
location marker: range location	// TODO Workaround for [...] [...] End Workaround
concern tag	in 12 places in the code: TODO pam: null-guard case of [...] [input] corruption

Table 1: Eclipse task comment categorization

- For example, the empty comment labelled “location marker: point location” in Table 1 serves as a location marker. Considering the context, such a comment can serve as a reminder that something needs to be done to the code around the comment.
- Another example, the example labelled “location marker: range location” in Table 1, precisely denotes a range of source code that the task comment applies to.

2.7 Concern tags

To mark the places in the code that are related to a single concern [5], developers place the same identifying tag—which we call concern tag—in the task comments. This is concern tagging approach is an example of Griswold’s information transparency techniques, which aim to capture related parts of the code—especially the ones that are not well-modularized—by non-programming language constructs, such as naming convention, formatting style, or tags embedded in comments [2].

- In the example labelled “concern tag” in Table 1, the developer used the same comment to denote 13 places in the code that relates to an input corruption.

3. ANALYZING COMMENTS

Having investigated the task comments in the AWB code-base, we see some challenges in using Eclipse task comments as inputs in repository mining, which are discussed in the rest of this section.

3.1 Inferring meta-data from a task comment

An Eclipse task tag only provides two pieces of meta-data than a Java comment, tag creation time and tag severity: Eclipse logs the time when the task tag is first saved, and also supports users in defining a severity value for each task tag type (not for each instance of task tag).

From our study, we see that developers employ common convention to encode additional meta-data that is not explicitly supported by a Eclipse task tag. However, some of types of meta data can still be hard to infer from comments.

Author

Many comments contain the name of the author of the comment. This author information can be helpful for searching all the comments written by the author. However, parsing the author information from the comment may require some care because the format of the format of the author information can vary. For example, Pam tends to put her name preceding a colon, as in “// TODO pam: [...]”. Sue sometimes types her name all in letters followed by a dash, as in “// TODO [...] -sue.” If the source code is kept in a code repository, an alternative way to infer the author information is to associate the author information in the change log with the comment.

Change request identifiers

An Eclipse task comment sometimes represents a task that a developer needs to perform as part of the change described in the change tracking system, as shown in Section 2.2. In such a case, the developer usually put the change request number in the comment. For example, in AWB, a change request

is denoted as an ECR (Enhancement Change Request) and a particular ECR is referred to by its ID, such as in “// TODO pam: see ECR 327.” The convention for specifying an ECR is pretty standard, with the ECR number followed by the string “ECR”.

3.2 Implied context in a task comment

Because the tags are embedded in the code, task comments tend to depend a lot on the context of the surrounding code. For example, some task comments tend to use context-sensitive words which need to be interpreted with the surrounding code. For example, in the task comment we have shown in Section 2.1, “// TODO an ugly hack for now -sue. Joan, please fix it,” the word “it” requires the previous discussion between Sue and Joan and the code context to make sense.

Some task comments may not even have words at all, but the meaning of the task may be apparent to a human. We demonstrate by an example not from the AWB code, an empty task comment “// TODO.” Such a comment does not mean much on its own. However, if we notice that the task comment is enclosed by a method with no statements, it is apparent to us that the task is to implement the method. Such a case can pose challenges to mining algorithms.

3.3 Inferring the scope of a task comment

The scope of the comment is often not apparent because the comment only denotes a single point in the code. Developers use different assumptions on what region of code the comment applies. For example, comments may not contain any region information, but a developer sometimes uses a comment that refers to the statement immediately following the comment, sometimes uses a comment to refer to all the statements until a blank line is encountered, and sometimes uses a comment to denote the code in the whole enclosing scope, such as the empty comment denoting an unimplemented method we describe in this section. Although we have shown in Section 2.6 of one example where the developer have precisely denote a region that the comment applies to, that is the only such example from the whole study.

In addition, the task comment may apply to multiple non-contiguous places in code. For example, the task comment “// TODO pam: remove tracery when NPE [NullPointerException] is solved.” refers to tracing statements in many places, not just the statement immediately below the comment. Finding all the places the developers implied can be challenging for a mining tool.

Furthermore, even the developer may only have a fuzzy idea of all the places the comment denotes. For example, the task comment “// TODO -- Beth changed these at some point, to something Eclipse 3.0 compliant” denote a fuzzy area of code that was to be changed, as porting the code to work with Eclipse 3.0 is not a trivial task and requires changes to many places in the code. Thus, a developer cannot easily specifies all the places in the code that need to be changed in complete when planning the change. Therefore, it is very hard for a mining tool to infer such information.

3.4 Informality in a task comment

In the study, we see that the task comments are typically more informal and shorter than description from the bug report or JavaDoc comments. For example, many of the

comments only contain one single word or incomplete sentences. This is not surprising because many task comments are meant for personal reminders and for temporarily use. Also, because the task comments are embedded in the code, the fear of clutteriness in code may have prevented developers on elaborating a comment to make it formal. However, this informality in the task comments can make the mining tools that use natural language processing techniques challenging to apply.

4. DISCUSSION

In this section, we discuss some issues with our study.

4.1 Significance of Eclipse task comments

To “talk” to other team members through source code, a developer may use a Java comment, not necessarily a task comment. However, we did not investigate all the Java comments: The codebase contains 15,748 JavaDoc comments² and 13,457 non-JavaDoc comments³, and it was impossible to analyze all of them manually. Although task comments only accounts for a small fraction of all the comments in the AWB codebase, we still chose to examine task comments. Task comments are likely to be good candidates to contain information that is relevant to the current development context, as task comments are intended to be more transient—created and deleted more often—than other comments.

4.2 Generalizability of the results

In this preliminary study, we examined the task comments of one project. We cannot draw general conclusions about our task comment categorization from only one project. In addition, the results from this study may not be generalizable to other projects. The AWB is a small team of less than ten developers. Programming practices that are peculiar to a particular developer can dramatically affect the results.

5. CONCLUSION AND FUTURE WORK

In this paper, we have described our preliminary study on Eclipse task comments on the AWB codebase. We have found that these task comments contain rich and a wide range information, as shown in the categorization of task comments we have presented. Many task comments from study illustrate some challenges in treating task comments as input for analysis.

Although the conclusion drawn from our study is not generalizable to all projects, our study has shown some examples of task comments being a promising input to analyze. As future work, one direction of research is to infer the meta-data and contextual information of task comments, as such information is not captured by the current Eclipse task mechanism. Another direction of research is to come up with novels ways to analyze the inferred meta-data and contextual information, together with the content of the task comments.

6. ACKNOWLEDGMENT

We are grateful to the AWB team for lending their codebase for this study, as well as the prompt and useful help in understanding the intention of the comments. We would also like to thank Mark Chu-Carroll and Martin Robillard for many inspirational discussions. Moreover, we would like to thank anonymous reviewers for the useful feedback.

7. REFERENCES

- [1] Eclipse task tags website. <http://127.0.0.1:55317/help/index.jsp?topic=/org.eclipse.jdt.doc.user/reference/preferences-task-tags.htm>.
- [2] W. G. Griswold. Coping with crosscutting software changes using information transparency. In *Reflection 2001: International Conference on Metalevel Architectures and Separation of Crosscutting*, pages 250–265, 2001.
- [3] B. W. Kernighan and R. Pike. *The practice of programming*. Addison-Wesley Longman Publishing Co., Inc., 1999.
- [4] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.
- [5] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communication of ACM*, 15(12):1053–1058, 1972.

²We define the number JavaDoc comments as the number Java tokens “/” in the source code.

³We define the number of non-JavaDoc comments as the number of Java tokens “/”, plus the number of Java tokens “/” in the source code.

Text Mining for Software Engineering: How Analyst Feedback Impacts Final Results

Jane Huffman Hayes and Alex Dekhtyar and Senthil Sundaram
Department of Computer Science
University of Kentucky
{hayes,dekhtyar}@cs.uky.edu, skart2@uky.edu

Abstract

The mining of textual artifacts is requisite for many important activities in software engineering: tracing of requirements; retrieval of components from a repository; location of manpage text for an area of question, etc. Many such activities leave the “final word” to the analyst – have the relevant items been retrieved? are there other items that should have been retrieved? When analysts become a part of the text mining process, their decisions on the relevance of retrieved elements impact the final outcome of the activity. In this paper, we undertook a pilot study to examine the impact of analyst decisions on the final outcome of a task.

1. Introduction

One of the distinguishing features of data mining versus, for example, similar database tasks, is the fact that knowledge acquired from mining need not be exact. In fact, it may, in part, be inaccurate. Methods for typical data mining tasks, such as classification, discovery of association rules, and retrieval of relevant information, do their best to produce the most accurate results. However, the accuracy is subject to the internal properties of the method, as well as the quality and complexity of the artifacts (data) under consideration.

In the field of Software Engineering, we can see two distinct and well-defined ways in which data mining, information retrieval, and machine learning methods are applied. The first direction is the exploratory study of existing artifacts of software development: document hierarchies, code repositories, bug report databases, etc., for the purpose of learning new, “interesting” information about the underlying patterns. Research of this sort is tolerant to the varying accuracy of data mining methods: while certain subtleties of some datasets might be missed, the most general patterns

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR '05, May 17, 2005, St. Louis, MO, USA

Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

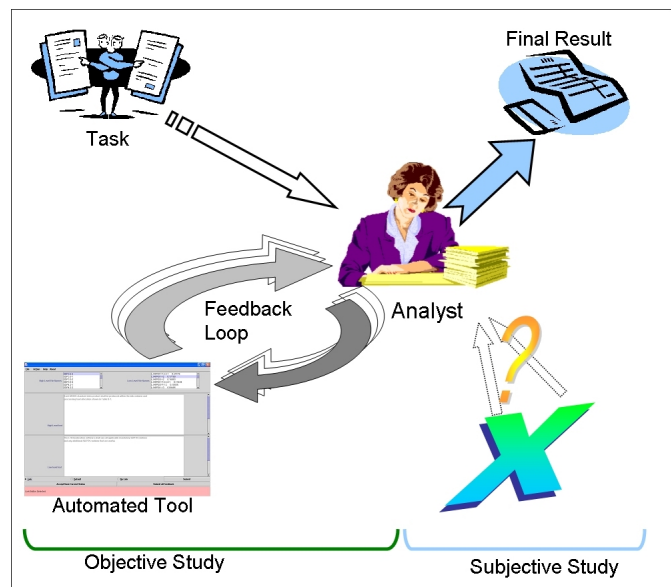


Figure 1. Human analyst will always stand between computer-generated results and the final result.

will, most likely, be discovered in analysis.

The second direction is the application of data mining¹ techniques to different processes in the software lifecycle with the purpose of automating and improving performance on the tasks involved. Potential benefits of such automation are significant. Data mining techniques are typically applicable to some of the most labor-intensive tasks, and are capable of speeding up the performance on them by orders of magnitude. At the same time, such applications of data mining methods **are not very error-tolerant**: undetected inaccuracies that creep into the results of data mining procedures may beget new inaccuracies in the later stages of development, thus producing a snowball effect.

¹Here and elsewhere in the paper we use the term “data mining” in its broadest sense, including certain related activities and methodologies from machine learning, natural language processing, and information retrieval in its scope.

To be able to obtain the benefits of applying data mining methods to specific tasks (good accuracy, *fast*), without the drawbacks (inaccuracies are very costly), a *human analyst must always assess and possibly correct the results of the automated methods*. The process of involving data mining methods in task execution during the software development lifecycle is described in Figure 1. A specific task is assigned to an analyst. The analyst has software to help execute the task. The analyst consults the software, obtains preliminary results, and provides the software with feedback. At some point, the analyst decides that the obtained answer is correct and outputs the final results of the task.

As the goal of introduction of data mining methods is improvement of the process, we are naturally concerned with the results produced by the automated tool. However, we observe that **the only result that is seen by others is generated by the analyst!** Therefore, we can only succeed if the final result, prepared by a human analyst, is good. In general, this is *not* equivalent to producing good results automatically.

We view this process from the point of view of the developers of the automated tool. Traditionally, the success of a data mining tool is measured by the accuracy of its results. However, in the process described in Figure 1, the ultimate concern lies with the accuracy of the final, analyst-generated output. This output is affected by a number of factors, including the accuracy of the automated tool. But is better accuracy of the tool equivalent to better accuracy of the analyst? And are there any other factors that play a role in analyst decision-making? Level of expertise? Trust of the software?

In order to claim success of the software, we must study not only the quality of the software output, but also *what the analysts do with it*. The ultimate success of the software is then determined by the quality of the final output.

What we have done. We have performed a pilot study on how human analysts work with machine-generated data. Our study was conducted using the task of IV&V requirements tracing [2, 3, 1] on a relatively small dataset. While the study was too small in size (only three analysts participated) to draw any far-reaching conclusions, its results (in all cases, the quality of the results decreased) suggest that we are looking at the right problem. The pilot study is discussed in Section 4 of the paper.

What we are planning to do. In Section 3 we outline the framework for a large scale experiment measuring the work of analysts with computer-generated data. Our goal is to determine the “regions” of precision-recall (see Section 2) space representing the quality computer-generated answer sets that allow human analysts to produce final results of high quality. We are also interested in studying what external factors affect analyst interaction with computer-generated artifacts.

We begin by briefly outlining our research on the use of Information Retrieval (IR) methods for candidate link generation in requirements tracing tasks, and by describing how we came across the problem discussed in this paper.

2. Motivation: Humans Matter!

We first came across the issue of the quality of analyst evaluation of computer-generated results during our preliminary experiments with the application of information retrieval (IR) to requirements tracing [2]. At its core, requirements tracing boils down to comparing the content of pairs of high- and low-level require-

	SuperTracePlus	Analyst
Correct links (total)	41	41
Correct links found	26	18
Total number of candidate links	67	39
Missed requirements	3	6
Recall	63.41%	43.9%
Precision	38.8%	46.15%

Table 1. SuperTracePlus and analyst performance on the MODIS dataset.

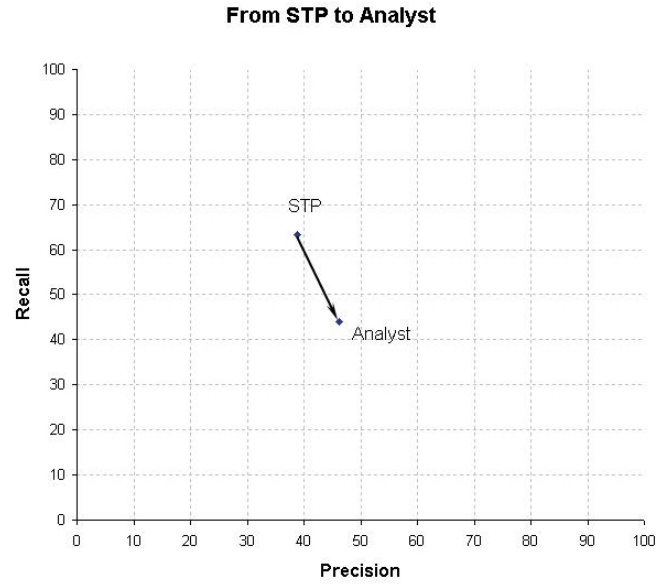


Figure 2. From SuperTracePlus trace to Analyst's trace.

ments and determining whether they are similar/relevant to each other. We hypothesized that IR methods, that basically do the same thing, can be applied successfully to tracing.

We had implemented two IR methods and wanted to compare the results these methods produced with the results obtained by a senior analyst working with a familiar advanced requirements tracing tool (SuperTrace Plus). The complete results of that experiment can be found in [2]. The analyst received a tracing task (19 high-level and 50 low-level requirements, 41 true links from the MODIS [5],[4] documentation) and performed it in two steps. First, he used SuperTracePlus (STP) [6], a requirements tracing tool developed by Science Applications International Corporation (SAIC), to obtain a list of candidate links. The analyst then used the candidate trace generated by STP as a starting point and examined each link in detail. He removed from the trace links that he deemed unnecessary and also introduced some new links (whenever he felt that a link was missing). In Table 1, we have summarized this part of the experiment (all data comes from [2]).

As can be seen from the table, the analyst improved the precision of the final trace. However, the analyst significantly lowered

the recall². Using a recall-vs.-precision plot, we can illustrate the shift in these key measures of the quality of the trace from STP to the analyst (see Figure 2). In this experiment, the senior analyst had a high level of familiarity with SuperTracePlus, however, he was not very familiar with the MODIS project (beyond the dataset that we provided).

While a single point of anecdotal evidence is insufficient for any conclusions, it prompted us to consider the implications of the analyst’s work with the software on the final results.

3. Large Scale Study

As mentioned in Section 1, when data mining tools are used directly in the software lifecycle process, rather than for after-the-fact analysis, high accuracy of the outcome must be ensured. Human analysts play the role of inspectors, examining the output of the tools and correcting it where necessary. The result of their work is passed along to the next tasks.

We ask ourselves a question: in the presence of mining tools, what factors affect the result produced by the analyst?

Right now, we only have a partial answer. Clearly, there are some overall factors that affect the quality of the analyst work, with or without software: analyst expertise with the task, level of domain expertise, and even such mundane and hard-to-control factors such as boredom with the task. However, in the presence of mining software designed to provide good approximations fast, there are other factors. The accuracy of the tool must play a role. Also, the degree of the analyst’s familiarity with the tool and the degree of analyst trust in the results of the tool play a role.

However, simply stating that there is a dependence is not enough - as the exact character of such dependence is not obvious. For example, we would like to hypothesize that the quality of the tool results (measured in terms of precision and recall) affect the quality of analyst results in a monotonic way: better recall-precision of the tool yields better recall-precision of the final result. However, we note that if the precision and recall of the tool are very low (e.g., around 10% each), the analyst has “nowhere to go but up.” At the same time, when the precision and recall of the tool are very high (e.g., around 95%), the analyst has almost “nowhere to go but down.” Should we observe such results, how do we interpret them and what conclusions do we draw for the tool? Should we be “watering down” the results of an accurate tool, to ensure that an analyst will not decrease the precision and recall?

The goal of the large scale study we plan to undertake is to discover the patterns of analyst behavior when working with the results of data mining tools during the software lifecycle and to establish the factors that affect it and the nature of their effects.

The key factor we are looking at in the first stage is software accuracy, that we represent via the precision-recall pair of measures. The space of all possible accuracy results is thus a two-dimensional unit square as shown in Figure 3. For both precision and recall, we establish the regions where the values are *low*, *medium*, and *high*³. The nine regions are shown in Figure 3.

²Precision is defined as the number of correct answers returned divided by the total number of answers returned. Recall is defined as the number of correct answers returned divided by the total number of answers.

³There is a certain asymmetry between recall and precision in this respect. We assume that precision is high if it is above 60%, and is low when it is under 33%. However, the recall is high when its

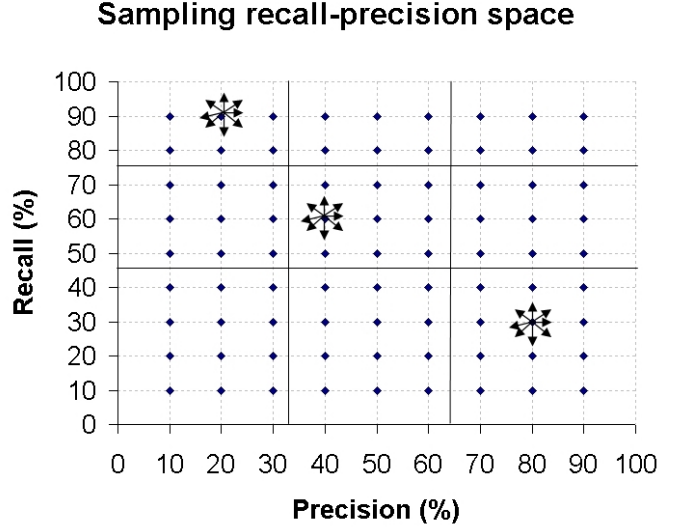


Figure 3. Sampling the space of possible outputs: what will the analysts do?

Our experiment consists of offering senior analysts, who have experience with requirements tracing, a computer-generated candidate trace with a preset accuracy from one of the nine regions. The analyst would then be asked to check the candidate trace and submit, in its place, the final trace. We will measure the accuracy of the final trace and note the shift from the original (such as in Figure 2).

Our goal is to establish under which levels/conditions of the software, analyst expertise, and analyst attitude towards software, the resulting output improves (significantly) upon the computer-generated candidate trace. Such discovery has two implications on the software and the process. We must ensure that the mining tool delivers results in the accuracy range that allows the analysts to improve it. We must also strive to create the right conditions for the analysts to work with the data.

4. First Steps

In our preliminary study, our goal is to investigate the feasibility of our hypothesis, that the accuracy of computer-generated candidate traces affects the accuracy of traces produced by the analysts. We also want to understand if a more detailed study is needed.

For the pilot study, we used the final version of the MODIS dataset described in [2, 3]. It contains 19 high-level requirements, 49 low-level requirements, and 41 true links between them. Using the output of one of our IR methods as the base, we generated candidate traces for a number of sampled points from the precision-recall space described in Section 3, including the three candidate traces (also shown in Figure 3) with the following parameters⁴:

value is above 70-75%, and is low when it is below 45%.

⁴Altogether, we have generated six different candidate traces and distributed them to six analysts. However, only three analysts have completed their work at this time.

You may perform this task on your own schedule, at your convenience, in the location that you prefer (home, work, etc.). The work need not be completed all in one sitting.

We have provided you with the following items:

- 1 - A textual listing of the high level requirements;
- 2 - A textual listing of the low level requirements;
- 3 - A trace report indicating potential low-level requirements for a given high-level requirement;

These items may be used in softcopy form (i.e., you may feel free to use a word processing tool to perform interactive searches, etc.) or may be printed out and used in hardcopy. We will discuss each below.

The trace report contains the list of candidate links for each high-level requirement. The candidate links for a single high-level requirement are shown in the order of decreasing relevance.

Your task is to produce the final traceability report from the given trace report and ensure that all high level requirements have been traced correctly and that any low level requirements for them have been identified. The end product that you should provide to us is:

- A marked up trace report (cross through low level requirements that you do not agree with, write in low level requirements that you feel are missing);
- A brief description of how you performed the task (did you use Word to search through the softcopy?, did you read everything from the hardcopy?);
- A log of how much time you spent on this task.

Figure 4. Instructions for the pilot study participants (abridged).

T1: Precision=60%; Recall=40%;
T3: Precision=20%; Recall=90%;
T4: Precision=80%; Recall=30%⁵.

The candidate traces were distributed to experienced analysts with tracing experience (manually or with a tool). The abridged version of the instructions provided to the analysts is shown in Figure 4.

Each analyst was provided with one of the trace sets described above. They were given a one-week period to perform the work, but were not given any time constraints (i.e., they could spend as many hours on the task as they desired). The analysts were asked to return their answer sets (all chose to return softcopies in various formats), a brief description of the process employed during the experiment (to determine conformance), and a brief log of activities. From each answer set we have collected the following information: **Or-Pr**, **Or-Rec**, original recall and precision; **Pr**, **Rec**, precision and recall achieved by the analyst; **Rmv**, **Rmv-Tr**, total number of links and number of true links removed and **Add**, **Add-Tr**, total number of links and number of true links added by the analyst; **Delta-Pr**, **Delta-Rec**, the change in precision and recall, and **Time**, the time spent on the task. Table 2 and Figure 5 summarize the results of the pilot study.

5. Conclusions and Future Work

As stated in Section 1, we are aware of some shortcomings of

⁵Because we had 41 true links in the dataset, for some values of recall and precision, we had to take the nearest achievable point (e.g., 12 true links in the trace, equal to 29.2% recall, was used for the 30% recall point).

	T1	T3	T4
Or-Pr:	39.6%	20%	80%
Or-Rec:	60.9%	90.2%	29.2%
Rmv:	38	155	10
Rmv-Tr:	6	11	5
Add:	26	16	43
Add-Tr:	4	2	6
Pr:	45.1%	58.7%	22.9%
Rec:	56.1%	65.8%	26.8%
Delta-Pr:	+ 3.1%	+ 38.7%	- 57.1%
Delta-Rec:	- 5.8%	- 24.4%	- 2.4%
Time:	2.5 hrs	2 hrs	3 hrs

Table 2. Summarized results of the pilot study.

IR and text mining methods, such as that they admit inaccurate results. This is why, when used in tasks within the software life-cycle, an analyst needs to inspect computer-generated results to prevent the snowball effect.

It is clear from our anecdotal study that there are factors at work influencing analyst decision making, and, hence, the final results. For example, examining Table 2, we can see that analysts who were given trace sets with low recall took longer to complete the task (25 - 50% longer). They did not necessarily produce any “worse” final results than the analyst with a high recall trace set (note that the analyst who had the high recall trace set ended with recall that was 24.4% lower). This observation is particularly interesting because the analyst with that trace set, T3 (recall of 90% and precision of 20%), had a large amount of false positives to go through. That means many more candidate links had to be examined. One would think that such an arduous process would result in worse results than an analyst who did not have to “wade through” many false positives. But in this very small study, that was not the case.

In the pilot study, the analysts did not exhibit a pattern of improving the results of the candidate traces “no matter what.” That would have rendered our questions moot. On the contrary, analyst behavior consistently shifted the answers towards the vicinity of the *precision = recall* line (see Figure 5). This was evident if recall was higher than the *precision = recall* line to begin with, or if it was lower than the *precision = recall* line to begin with.

It is clear, then, that we must undertake a larger, controlled experiment, as described in Section 3. This must be done to ensure that we account for important factors that may influence analyst decisions, such as expertise, familiarity with/trust in the software, domain knowledge, etc... At the same time, we must factor out some extraneous variables, such as environmental issues (e.g., ambient noise), etc.

Acknowledgements. Our work is funded by NASA under grant NAG5-11732. We thank Stephanie Ferguson, Ken McGill, and Tim Menzies. We thank the analysts who assisted with the pilot study.

6. References

- [1] Dekhtyar, A., Hayes, J. Huffman, and Menzies, T., Text is Software Too, Proceedings of the International Workshop on Mining of Software Repositories (MSR) 2004, Edinburgh, Scotland, May 2004, pp. 22 - 27.

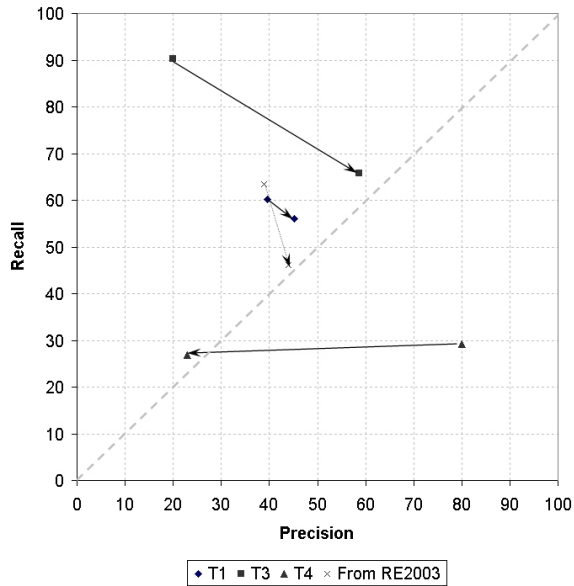


Figure 5. Pilot study: what the analysts did.

- [2] Hayes, J. Huffman, Dekhtyar, A., Osbourne, J. Improving Requirements Tracing via Information Retrieval, in *Proceedings of the International Conference on Requirements Engineering (RE)*, Monterey, California, September 2003, pp. 151 - 161.
- [3] Hayes, J. Huffman, Dekhtyar, A., Sundaram K.S., Howard S., Helping Analysts Trace Requirements: An Objective Look, in *Proceedings of the International Conference on Requirements Engineering (RE)*, Kyoto, Japan, September 2004.
- [4] Level 1A (L1A) and Geolocation Processing Software Requirements Specification, *SDST-059A, GSFC SBRs*, September 11, 1997.
- [5] MODIS Science Data Processing Software Requirements Specification Version 2, *SDST-089, GSFC SBRs*, November 10, 1997.
- [6] Mundie, T. and Hallsworth, F. Requirements analysis using SuperTrace PC. In *Proceedings of the American Society of Mechanical Engineers (ASME) for the Computers in Engineering Symposium at the Energy & Environmental Expo*, 1995, Houston, Texas.

Software Changes and Evolution

Analysis of Signature Change Patterns

Sunghun Kim, E. James Whitehead, Jr., Jennifer Bevan

Dept. of Computer Science

Baskin Engineering

University of California, Santa Cruz

Santa Cruz, CA 95060 USA

{hunkim, ejw, jbevan}@cs.ucsc.edu

ABSTRACT

Software continually changes due to performance improvements, new requirements, bug fixes, and adaptation to a changing operational environment. Common changes include modifications to data definitions, control flow, method/function signatures, and class/file relationships. Signature changes are notable because they require changes at all sites calling the modified function, and hence as a class they have more impact than other change kinds.

We performed signature change analysis over software project histories to reveal multiple properties of signature changes, including their kind, frequency, and evolution patterns. These signature properties can be used to alleviate the impact of signature changes. In this paper we introduce a taxonomy of signature change kinds to categorize observed changes. We report multiple properties of signature changes based on an analysis of eight prominent open source projects including the Apache HTTP server, GCC, and Linux 2.5 kernel.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics – *Product metrics*, K.6.3

[Management of Computing and Information Systems]:

Software Management – *Software maintenance*

General Terms

Measurement, Experimentation

Keywords

Software Evolution, Signature Change Patterns, Software Evolution Path

1. INTRODUCTION

Software continually changes due to performance improvements, new requirements, bug fixes, and adaptation to a changing operational environment [1]. Software changes include function body modification, local variable renaming, moving functions from one file to another, and function signature changes [2]. Among these changes, function signature changes have a significant impact on parts of the source code that use the changed functions. Most signature changes cause a signature mismatch problem. Understanding the character and evolution patterns of function signature changes is important to researchers concerned with alleviating the impact of signature changes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR '05, May 17, 2005, Saint Louis, Missouri, USA

Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00.

Others have observed code changes, though none have examined signature changes at the same level of detail. Kung et al. identified kinds of code changes [2] and Counsell et al. discussed the trends of changes in Java code [3]. Both of them identified large granularity change kinds, such as method body changes, method addition, method deletion and whether the signature changed. Their categorization of changes is useful for understanding software changes in overview. Our analysis of signature changes is motivated by the goal of eventually providing automated support for fixing signature mismatches, and for this we need a very fine-grain understanding and characterization of signature changes. Previous work did not examine signature changes at this level of detail, being concerned only with whether the signature did, or did not, change, but not what kind of change.

We focus on fine-grain changes in function signatures, categorizing them based on whether they increase, decrease, or do not modify the data flow between caller and callee. Within these broad categories, change kinds are further refined. We show the properties of function signature change patterns by answering the following research questions: How often do signatures change? What are the common signature change kinds? How often does each kind appear? Do they have a common evolution pattern?

The answers, along with analysis of the results, can be used to predict future signature changes, provide automatic change accommodation algorithms, develop glue code generators, or develop refactoring algorithms.

We analyzed eight prominent open source projects listed in Table 1. These eight open source projects are written in the C programming language. For our analysis, we used Kenyon, a data extraction, preprocessing, and storage backend designed to facilitate software evolution research [4]. Using Kenyon, we checked out all revisions or copied all releases of source code from each project, and extracted function signatures. We grouped signatures by function name, and observed the changes over revisions or releases to find properties of signature changes. We implemented an automatic signature change kind identification tool, but some change patterns are not automatically identifiable, such as concept splitting and merging. We also compared the number of signature changes over all functions to find the frequency of each signature change kind. Finally we looked for sequence patterns in the common evolution paths of function signature changes.

The remaining sections of the paper are as follows: In Section 2, we describe our analysis process with detailed information from the open source projects we analyzed. Sections 3, 4, 5, and 6 provide answers to our research questions. We discuss the limitations of our analysis in Section 7, and conclude in Section 8.

Table 1. Open source projects we analyzed. LOC indicates number of lines in .h and .c source files, including comments. The period shows the project history period for projects for which we directly accessed the SCM repository, otherwise we list release numbers. The number of revisions indicates the number of revisions we extracted or the number of releases we analyzed.

Project	Software type	LOC	SCM	Period/Releases	# of revisions/releases
Apache Portable Runtime (APR)	Portable C library	72,630	Subversion	Jan 1999 ~ Jan 2005	5832 revisions
Apache HTTP 1.3 (Apache 1.3)	HTTP server	116,393	Subversion	Jan 1996 ~ Jan 2005	7508 revisions
Apache HTTP 2.0 (Apache 2)	HTTP server modules	104,417	CVS	Jul 1999 ~ Aug 2003	3877 revisions
Subversion	SCM software	183,740	Subversion	Aug 2001 ~ Feb 2005	5886 revisions
CVS	SCM software	62,415	CVS	Dec 1994 ~ Sep 2003	2873 revisions
Linux Kernel 2.5 (Linux)	Linux OS	5,140,625	N/A	2.5.1 ~ 2.5.75	75 releases
GCC	C/C++ compiler	506,931	N/A	1.35, 1.36, ..., 2.7.2	15 releases
Sendmail	SMTP server	127,733	N/A	8.7.6, 8.8.3, ..., 8.13.3	37 releases

2. ANALYSIS PROCESSES

We analyzed eight open source projects, listed in Table 1, using the Kenyon system. Kenyon checks out all revisions from a SCM repository and invokes a fact extractor we implemented to extract function signatures. The extracted signatures are grouped by function names. The grouped signatures are ordered by revisions and stored in a signature change history file.

For the projects we analyzed, the revision history was stored using either the CVS or Subversion SCM system. An important issue in software evolution research is the extraction of logical transactions from the SCM repository. Since Subversion assigns a revision number per commit, there is no need to recover transactions for Subversion-managed projects [5]. CVS does not keep the original transaction information, usually requiring a process of transaction recovery [6]. Kenyon provides CVS transaction recovery using the Sliding Time Windows algorithm [4, 6]. Recently, the Apache Software Foundation (ASF) changed its SCM repository to Subversion from CVS using the cvs2svn converting tool. We analyzed some ASF projects, including Apache 1.3 and APR, whose repositories were converted. Since the cvs2svn tool uses the fixed time window algorithm [6] to convert CVS data for Subversion, using the converted data won't affect our analysis results.

We manually observed the signature change history file to identify common signature change kinds. After analyzing the signature change history files from various open source projects, we found the common change kinds shown in Table 3. While most of the change patterns can be automatically identified by a static software analysis, some change kinds, such as concept merging/splitting changes are not automatically identifiable, requiring project knowledge concerning the project and parameter concepts. We implemented an automatic signature change kind identifier that reads a signature change history file, and annotates the file based on the identified kinds. After the signature change history file annotation, we calculate the frequency of each change kind. We also examine the sequence of signature change kinds of a given function to see if there was a common pattern in the signature evolution. The results of our analysis are presented in following sections.

3. SIGNATURE CHANGE KINDS

Before presenting our results, we describe our fine-grain taxonomy of signature change kinds. First we define the basic elements of a function signature: *parameter*, *argument*, *return parameter*, and the *signature*.

The *modifier* indicates a data type modifier such as *const*, *register*, and *static*. A *type* is the data type of a parameter, and name

indicates the parameter name. The *array/pointer* is the count of * or [] when a parameter is an array or pointer type. This represents both the array/pointer type and its dimension. Using these basic definitions, we now identify and define signature change kinds. In the remainder of the definitions, we use the subscript _{new} to indicate a later revision and _{old} a previous revision. If we omit the equality of elements, assume the other elements are the same. For example, in Definition 2 we define *N* if the *name_{old}* and *name_{new}* are different. We assume all other elements such as *type* and *modifier* are the same.

Definition 1 (Parameter, Argument, Return parameter, Signature)

Parameter **Param** = {modifier, type, name, array/pointer, order}
Argument **Arg** = a set of zero or more **Param**
Return parameter **R** = {modifier, type, array/pointer}
Signature **S** = {**R**, function name, **Arg**}

Definition 2 (Name change)

Function name change FN = function name_{new} ≠ function name_{old}
Parameter name change N = name_{new} ≠ name_{old}

The name change category has two kinds: function name change and parameter name change. Table 2 shows an example of parameter name changes. A parameter name change does not introduce a signature mismatch problem since the parameter name is used internal to the function. However, parameter name changes may cause semantic errors. For example, as shown in Table 2, if the change of parameter from 'service_name' to 'display_name' indicates a change in parameter meaning, call sites will compile without error, but the software may not work as expected due to the change in meaning.

Table 2. A parameter name change in Apache 1.3, os/win32/service.c file, ValidService function. The old version is on top, the new version is on bottom. Changes between versions are shown in bold.

BOOL ← char *service_name
BOOL ← char * display_name

Definition 3 (Ordering change)

Order = the position of an argument
Ordering change O = order_{new} ≠ order_{old}
Only ordering change o = O and |Arg_{new}| = |Arg_{old}|
Ordering change by addition OA = O and |Arg_{new}| > |Arg_{old}|
Ordering change by deletion OD = O and |Arg_{new}| < |Arg_{old}|

The parameter ordering changes occur when the order of two or more parameters has been changed. The typical motivation behind these changes is parameter order consistency with other function

signatures. Sometimes adding or deleting parameters causes signature ordering changes.

Definition 4 (Parameter modifier change)

Parameter modifier change $M \equiv \text{modifier}_{\text{new}} \neq \text{modifier}_{\text{old}}$

Modifier changes happen when developers alter a modifier without changing the data type. We mostly observed the addition or removal of the ‘const’ modifier in the C programs of our data set.

Table 3. A taxonomy of signature change kinds. The * item indicates that the item is manually identifiable, and hence the frequency is not reported in this paper.

Data flow invariant	*Function name change (MN) Parameter only ordering change (o) Parameter name change (N) Parameter modifier change (M) *Concept merge/splitting change (CM/CS) Array/Pointer operation change (P) *Return type change (R) Primitive type change (T) Complex type name change (CN)
Data flow increasing	Parameter addition (A) Ordering change by addition (OA) *Return type addition (RA) *Complex type inner variable addition (CA)
Data flow decreasing	Parameter deletion (D) Ordering change by deletion (OD) *Return type deletion (RD) *Complex type inner variable deletion (CD)

Definition 5 (Parameter array/pointer change)

Parameter array/pointer change $P \equiv \text{array/pointer}_{\text{new}} \neq \text{array/pointer}_{\text{old}}$

Array/pointer dimension changes occur when developers add or delete dimensions of pointer or array parameters. An example of this change is shown in Table 4.

Table 4. A pointer change example in APR, threadproc/unix/procsup.c file, ap_detach function.

<code>ap_status_t ← ap_proc_t **new, ap_pool_t *cont</code>
<code>ap_status_t ← ap_proc_t *new, ap_pool_t *cont</code>

Definition 6 (Parameter addition/deletion)

Parameter addition $A \equiv p \in \text{Arg}_{\text{new}}$ and $p \notin \text{Arg}_{\text{old}}$

Parameter deletion $D \equiv p \notin \text{Arg}_{\text{new}}$ and $p \in \text{Arg}_{\text{old}}$

The parameter addition and deletion changes are common change kinds; an example is shown in Table 5.

Table 5. Parameter addition changes in the Linux kernel, kernel/sched.c file, try_to_wake_up function. First sync was added, then later the variable state was added.

<code>static int ← task_t * p</code>
<code>static int ← task_t * p, int sync</code>
<code>static int ← task_t * p, unsigned int state, int sync</code>

One of the most interesting change kinds is the concept splitting/merging change defined in Definition 7. Usually concept splitting/merging changes look like parameter addition or deletion changes. But if we observe the changes carefully, the new parameters can be derived from existing or deleted parameters.

For example, suppose a function takes ‘first name’ and ‘last name’ as its arguments. In the next version, the function takes only ‘name’. It seems the ‘first name’ and the ‘last name’ parameters are deleted while the new ‘name’ parameter is added. In fact, the new parameter, ‘name’, is a combination of the deleted parameters, ‘first name’ and ‘last name’. In this case, a derivation function F exists.

Definition 7 (Concept merging/splitting change)

$\text{A}_{\text{sub}} \subseteq \text{Arg}_{\text{old}}$

Concept merging $\text{CM} \equiv A$ and \exists a derivation function F, such that $p_{\text{added}} = F(\text{A}_{\text{sub}})$ and $|\text{A}_{\text{sub}}| > 1$

Concept splitting $\text{CS} \equiv A$ and \exists a derivation function F, such that $p_{\text{added}} = F(\text{A}_{\text{sub}})$ and $|\text{A}_{\text{sub}}| = 1$

The ‘name’ parameter can be derived using a derivation function F: ‘name’ = F(‘first name’, ‘last name’). We define this kind of changes as a concept merging change. If the evolution goes in the opposite direction, we define it as a concept splitting change.

Definition 8 (Primitive types and Complex types)

Primitive type set $\text{PTS} \equiv \{\text{char, int, long, float, double}\}$

Is primitive type $\text{PT}(t) \equiv \text{true iff } t \in \text{PTS}, \text{ else false}$

Is complex type $\text{CT}(t) \equiv \text{true iff } t \notin \text{PTS}, \text{ else false}$

Definition 9 (Primitive type change)

Primitive type change $\equiv \text{type}_{\text{new}} \neq \text{type}_{\text{old}}$ and

$\text{PT}(\text{type}_{\text{new}})$ and $\text{PT}(\text{type}_{\text{old}})$

Definition 10 (Complex type change)

Type variable set $\text{TVS} \equiv$ a set of variables used in a complex type

Complex type name change $\text{CN} \equiv \text{type}_{\text{new}} \neq \text{type}_{\text{old}}$ and $(\text{CT}(\text{type}_{\text{new}}) \text{ or } \text{CT}(\text{type}_{\text{old}}))$

Complex type inner variable addition

$\text{CA} \equiv \text{CT}(\text{type}_{\text{new}})$ and $\text{CT}(\text{type}_{\text{old}})$

and $\text{type}_{\text{new}} = \text{type}_{\text{old}}$ and $|\text{TVS}_{\text{new}}| > |\text{TVS}_{\text{old}}|$

Complex type inner variable deletion

$\text{CD} \equiv \text{CT}(\text{type}_{\text{new}})$ and $\text{CT}(\text{type}_{\text{old}})$

and $\text{type}_{\text{new}} = \text{type}_{\text{old}}$ and $|\text{TVS}_{\text{new}}| < |\text{TVS}_{\text{old}}|$

Definition 11 (Return parameter change)

Return type change $R \equiv \text{modifier}_{\text{new}} \neq \text{modifier}_{\text{old}}$ or

$\text{type}_{\text{new}} \neq \text{type}_{\text{old}}$ or $\text{array/pointer}_{\text{new}} \neq \text{array/pointer}_{\text{old}}$

and $\text{type}_{\text{new}} \neq \text{void}$ and $\text{type}_{\text{old}} \neq \text{void}$

Return type addition $\text{RA} \equiv \text{type}_{\text{new}} \neq \text{type}_{\text{old}}$ and $\text{type}_{\text{old}} = \text{void}$

Return type deletion $\text{RD} \equiv \text{type}_{\text{new}} \neq \text{type}_{\text{old}}$ and $\text{type}_{\text{new}} = \text{void}$

We define primitive type and complex types in Definition 8, and based on this definition we define primitive type and complex type changes.

The primitive type change indicates one of the parameter types has been changed while the parameter name remains unchanged. For example, if a parameter, ‘int age’ is changed to ‘long age’, it is a primitive type change. If the primitive type and the parameter name of an argument change together, it is a parameter addition/deletion change.

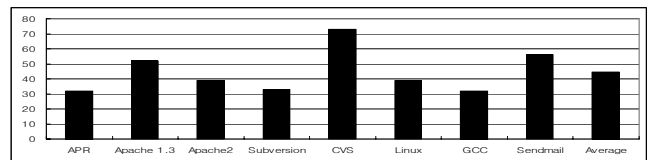


Figure 1. The percentage of the primitive data types used in function signatures of each project.

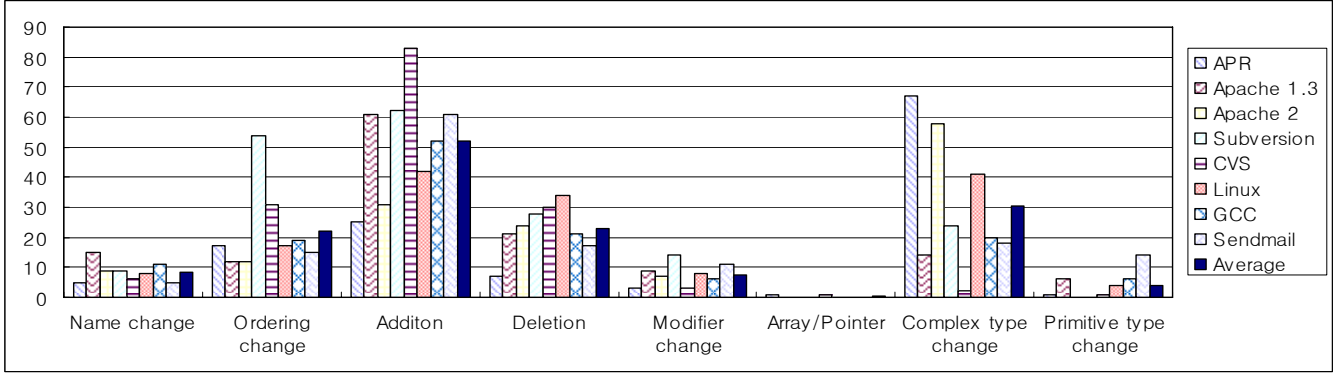


Figure 2. The percentages of each change kind frequency of the eight open source projects and average.

In the open source projects we observed, on average 55% of data types in signatures are complex data types (class, typedef, struct or union); see Figure 1. If one of the complex data types is changed, we define this change as a complex type change. These changes are different from parameter addition or deletion changes in that the old and new data types are related. Usually, when there are major changes in a class or structure, developers change the class/structure name. If there are only minor changes to the structure or class, such as adding a member variable, the structure/class name will not be changed. Since we are analyzing only signatures, we cannot automatically identify changes inside of structures or classes. To identify these changes, we need to monitor the structure/class body for changes in each revision. We may observe this in future work.

To define the major categories of our taxonomy, we use a data flow model between a function and a client. A client calls a function by passing arguments (**Arg**) and expecting returns (**R**) as shown in Figure 3. The total data flow is the union of **Arg** and **R**, defined in Definition 12. Broadly, when parameters or return values are added, there is an increase in the amount of data flowing between caller and callee, while parameter deletion or removal of return values results in reduction of data flow. Modifier changes or parameter name changes have no impact on the data flow.

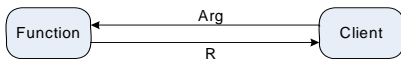


Figure 3. Data flow model.

Definition 12 (Data Flow)

$DF \equiv Arg \cup R$

Data flow invariant $\equiv |DF_{old}| = |DF_{new}|$

Data flow increasing $\equiv |DF_{old}| < |DF_{new}|$

Data flow decreasing $\equiv |DF_{old}| > |DF_{new}|$

4. FREQUENCY OF CHANGE KINDS

After identifying signature change kinds, we computed the frequencies of each kind. Figure 2 shows the signature change kind frequency percentages of each project. To simplify the graph we aggregated ordering changes (Ordering change = o+OA+OD). Figure 2 shows percentages for each change kind; the percentage is calculated by taking the number of observations of a particular change kind, and dividing it by the total number of signature changes observed for that system. For example, in Apache 1.3, we observed 202 parameter additions, and 327 total signature changes, resulting in a frequency percentage of 61%.

Note that one signature change can include more than one change kind. For example a signature change can include parameter addition, parameter deletion, and ordering changes. As a result, the summation of each percentage is greater than 100%. For example, the sum of all the CVS project change kinds is 157%. It means that whenever a function has a signature change in the CVS project, the signature change includes 1.57 different kinds of change, on average. If there is more than one instance of a particular change kind in a signature change, we count the kind only once. For example, if a signature change includes a parameter addition change three times, we count only one parameter addition change.

Figure 2 shows that the most common change kinds are parameter addition (average 52.13%), complex type changes (average 30.5%), and parameter deletion (average 22.75%). The array/pointer and primitive type change are relatively uncommon change kinds.

5. RATIO OF SIGNATURE CHANGES

To show the distribution of signature changes across functions, we counted the number of functions having n signature changes, with n varying from 0 to 16 signature changes (see Figure 4 for the signature change distribution for Subversion). Figure 4 shows that 5466 functions (77%) never changed their signature and 95% of the functions had fewer than three signature changes.

Another interesting ratio of signature changes can be obtained by comparing the number of signature changes and number of function body changes. We may examine this in future work.

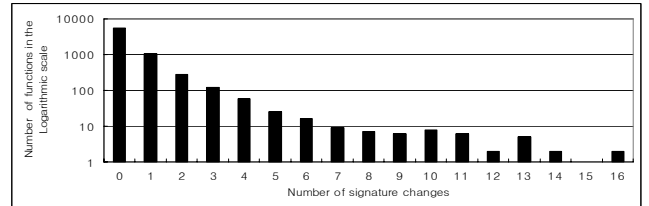


Figure 4. Count of signature changes of functions in the Subversion project. The x-axis indicates the number of signature changes, and the y-axis indicates the number of functions (log scale).

6. SIGNATURE EVOLUTION PATH

We wondered whether common signature evolution paths could be used to predict future software changes. For example, we

might detect that the most common signature changes occurred in this order: parameter addition (A), parameter deletion (D), ordering change (O), return type change (R), and parameter addition (A). In the future, when a known signature change evolution sequence occurred, such as A, D, O and R, we could predict the next signature change is likely to be a parameter addition (A).

To determine whether or not such common evolutionary paths exist, we noted all signature change evolution sequences. For example, when the signature of a function changes in this order: A, D, O, R, and A (See Table 3 for the change pattern abbreviations), we generate a change sequence, 'ADORA'. We examined all signature change sequences whose length is larger than three. We assumed that change sequences with fewer than four changes are rarely associated with common evolution paths.

After having an array of the sequences, we looked for the most common sequence (MCS) patterns using a modified longest common sequence (LCS) search algorithm [7]. Table 6 shows the top five common sequences of the Subversion project and overall eight projects. The occurrence shows how many times we found the change sequence patterns over all patterns, and percentage shows how common each occurrence is as a fraction of all observed occurrences (1,428 for the Subversion project and 2,025 for overall). We need to determine the conditional probabilities of each change kind to see if it depends on previous changes, and that the dependency rate is high enough to predict future change kinds. We weren't able to find predictable evolution paths from common sequences.

Table 6. The top five common function signature change pattern sequences of the Subversion project and across all projects. # means the count of occurrences of the pattern, and % means the percentage of times this sequence occurs.

Subversion Project			Overall projects		
Common Sequence	#	%	Common Sequence	#	%
ACDA	186	13%	AADA	198	9%
AADA	183	12%	ACDA	186	9%
AACD	159	11%	ADDD	171	8%
ADDD	152	10%	AACD	159	7%
ACAA	133	9%	ADAD	141	6%

7. THREATS TO VALIDITY

The results presented in this paper are based on selected eight open source projects. It includes major open source projects, but other open source or commercial software projects may not have the same properties we presented here. We analyzed only projects written in the C programming language; software written in other programming languages may have different signature change patterns. Some open source projects have revisions that are not compilable and contain syntactically invalid source code. In that case, we had to guess at the signatures or skip the invalid parts of the code. We ignored '#ifdef' statements because we cannot determine the real definition value; ignoring '#ifdef' caused us to add some extra signatures which will not be compiled in the real program.

8. CONCLUSIONS AND FUTURE WORK

We have introduced a fine-grain taxonomy of signature change kinds. Among change kinds, the common change kinds are parameter addition (52.13%), complex type change (30.5%) and

parameter deletion (22.75%). In future work we hope to this result can be used to alleviate signature change impact. If we can provide an ontological framework that includes a conceptual meaning for each parameter with its data type, it is possible to accommodate ordering changes and parameter deletion changes by generating glue code that resolves the signature mismatch problem. We found that about 77% of functions never change their signature and another 23% of functions change their signature once or twice.

We used a function name as an identifier to keep track of signature changes. Unfortunately, this means that if a function name changes, we lose its previous history of signature changes. The C++ and Java programming languages allow method overloading – more than one method with the same name but different parameters. When groups of overloaded methods evolve, sometimes ambiguity prevented us from determining which old method changed to which new method. Tu et al. introduced an origin analysis algorithm to find the origins of new procedures or files [8]. Origin analysis helps to find evolution paths when function names are changed or methods are overloaded. However, origin analysis requires heavy computation for entity analysis and dependency analysis. Providing more accurate results using origin analysis remains future work.

About 55% of parameters are complex data types such as structures, unions, or classes. Even though the signature remains unchanged, when a complex data type has changed internally, such as the addition of a member variable, it should be regarded as a signature change. Monitoring changes to each complex data type used in a signature to observe this kind of change remains future work.

Finally, further study is needed to explore the correlations between signature evolution and whole system evolution.

9. ACKNOWLEDGMENTS

Thank you to Mark Slater, and the anonymous reviewers for their valuable feedback on this paper. Work on this project is supported by Samsung Electronics, NSF Grant CCR-01234603, and a Cooperative Agreement with NASA Ames Research Center.

10. REFERENCES

- [1] M. M. Lehman, "Rules and Tools for Software Evolution Planning and Management," *Proc. Int'l Workshop on Feedback and Evolution in Software and Business Processes (FEAST 2000)*, Imperial College, London, July 10-12, 2000.
- [2] D. C. Kung, J. Gao, P. Hsia, F. Wen, Y. Toyoshima, and C. Chen, "Change Impact Identification in Object Oriented Software Maintenance," *Proc. the Int'l Conf. on Software Maintenance*, Victoria, Canada, 1994, pp. 202-211.
- [3] S. Counsell, et al., "Trends in Java code changes: the key to identification of refactorings?" *Proc. 2nd Int'l Conf. on Principles and Practice of Programming in Java*, Kilkenny City, Ireland, 2003, pp. 45 - 48.
- [4] J. Bevan, "Kenyon Project Homepage," 2005 <http://kenyon.dforge.cse.ucsc.edu>
- [5] B. Behlendorf et al., "Subversion Project Homepage," 2005 <http://subversion.tigris.org/>
- [6] T. Zimmermann and P. Weißgerber, "Preprocessing CVS Data for Fine-Grained Analysis," *Proc. MSR 2004*, Edinburgh, Scotland, 2004, pp. 2-6.
- [7] D. S. Hirschberg, "Algorithms for the Longest Common Subsequence Problem," *Journal of the ACM (JACM)*, vol. 24, no. 4, pp. 664 - 675, 1977.
- [8] Q. Tu and M. W. Godfrey, "An Integrated Approach for Studying Architectural Evolution," *Proc. Intl. Workshop on Program Comprehension (IWPC 2002)*, Paris, June, 2002, pp. 127.

Improving Evolvability through Refactoring

Jacek Ratzinger, Michael Fischer
Vienna University of Technology
Institute of Information Systems
A-1040 Vienna, Austria

{ratzinger,fischer}@infosys.tuwien.ac.at

Harald Gall
University of Zurich
Department of Informatics
CH-8057 Zurich, Switzerland

gall@ifi.unizh.ch

ABSTRACT

Refactoring is one means of improving the structure of existing software. Locations for the application of refactoring are often based on subjective perceptions such as "bad smells", which are vague suspicions of design shortcomings. We exploit historical data extracted from repositories such as CVS and focus on change couplings: if some software parts change at the same time very often over several releases, this data can be used to point to candidates for refactoring. We adopt the concept of bad smells and provide additional *change smells*. Such a smell is hardly visible in the code, but easy to spot when viewing the change history. Our approach enables the detection of such smells allowing an engineer to apply refactoring on these parts of the source code to improve the evolvability of the software. For that, we analyzed the history of a large industrial system for a period of 15 months, proposed spots for refactorings based on change couplings, and performed them with the developers. After observing the system for another 15 months we finally analyzed the effectiveness of our approach. Our results support our hypothesis that the combination of change dependency analysis and refactoring is applicable and effective.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Maintenance and Enhancement—*restructuring, reengineering*; D.2.8 [Software Engineering]: Metrics—*complexity measures, evolution measures*

Keywords

software evolution, refactoring, change smells

1. INTRODUCTION

The notion of "bad smells" was introduced by Fowler [4] and describes a vague suspicion that the software contains design deficiencies that should be restructured. Our research question is: Are there data sources other than source code to identify such kinds of smells for refactorings? We address

this question by exploiting change history data of software and analyze them to identify smells and, as a consequence, hot-spots for refactoring. We utilize our visualization approach of change couplings to help software engineers to locate places that deserve design improvements.

Furthermore, we then apply appropriate refactorings to the identified software parts and again observe the evolution for a period of release. Then at some point we again investigate the change history to see whether the initially suggested and implemented refactorings were effective with respect to change couplings. We can positively answer the question of effectiveness, if the refactored software keeps that status over the observed post-refactoring releases. As a result, we derive that, given the refactored structure does not again show high change couplings, these hot-spots were the right places to apply refactorings.

To evaluate our approach, we used a 500 000 lines of code (LOC) industrial Picture Archiving and Communication System (PACS) written in Java and observed it twice for a period of 15 months, with a change coupling driven refactoring between the two observation periods. The results show that change couplings point to highly relevant refactoring candidates in the code and that after refactoring the code has a low change coupling characteristics, which means that the refactorings were successful.

The origins of this work are our previous results described in [5], in which we concentrated on the measuring of software dependencies: Common change behavior of modules to be discovered on a macro level exploiting information such as version numbers and change reports. Source code control systems such as CVS provide necessary information about change requests and usually also about the change implementation, as the developer can use such systems for documentation purpose [3]. Thus, hidden dependencies not evident in the source code can be revealed. Such common change behavior of different parts of the system during the evolution is referred to as *logical or change coupling*. As a result, change couplings often point to structural weaknesses that should be subject to reengineering.

We propose to use refactoring based on change smells detected with the help of mining source code repositories. We extend the concept of "bad smells" introduced by Fowler to change smells, as some structural weaknesses are not evident in the code but in the software history. When developers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MSR'05, May 17, 2005, Saint Louis, Missouri, USA Copyright 2005 ACM 1-59593-123-6/05/0005...5.00

have to change some system part they often work on several files containing source code. Sometimes the dependencies are not easily detectable within the source code, e.g. when similar patterns or clones of source code are used but for different functionality. Nevertheless, in such a situation the engineer has to consider all the involved files to keep the consistency of the entire system. After detecting change smells and its cause, we suggest certain refactorings to improve the software.

The remainder of the paper is organized as follows: In Section 2 we present two of our change smells that are relevant for this paper. Next, in Section 3 we describe our industrial case study. In Section 4, we describe the core contribution of this paper, the change smell guided refactoring using examples from the case study. Section 5 positions our work in relation to other works, and in Section 6 we draw conclusions.

2. CHANGE SMELLS

Software often encloses change smells. These are spots in the system, which do not evolve smoothly but cause changes through a long period in the development process. To improve the development effort, we need decision support where to apply restructuring. Fortunately, most development teams collect historical data about the product's life cycle as they use software configuration management systems such as the Concurrent Versions System (CVS).

Refactoring is a vital technique to improve the design of existing systems by changing a software system in such a way that the external behavior of the code is not changed yet the internal structure is improved. It is an activity that accompanies all phases of the software life cycle. Many different refactorings have already been identified [4]. When applying refactorings on detected change smells we can demonstrate how the evolvability of software improves.

As CVS logs every action, it provides the necessary information about the history of a system. The log-information is pure textual, human readable information and retrieved via standard command line tools, parsed and stored in a relational database. Following the import of the logs, the required evolutionary information is reconstructed in a post processing phase. Log groups L_n are sets of files which were checked-in into the CVS by a single author within a short time-frame—typically a few minutes. The degree of logical coupling between two entities a, b can be determined easily by counting all log groups which both a and b are members of, i.e., $C = \{\langle a, b \rangle | a, b \in L_n\}$ is the set of logical coupling and $|C|$ is the degree of coupling.

We define *change coupling* (or logical coupling) as follows: *Two entities (e.g. files) are logically coupled if modifications affect both entities over a significant number of releases.* An interesting aspect of coupling is the distinction between internal and external. We define internal coupling as a dependency that happens between classes in respective parts of the system; e.g. the relations between classes of a single module and its submodules are defined as internal couplings. The couplings between classes within this module and any other part of the software (i.e. another module or another subsystem) are considered as external couplings.

In addition to Fowler's "bad smells", we investigate two change coupling smells in this paper:

Man-in-the-Middle: A central class evolves together with many others that are scattered over many modules of the system. Thus, we detect change couplings between the central class and the related ones; these related classes often exhibit change couplings among each other as well. A *Man-in-the-Middle* smell hinders the evolution of single modules, because of the strong dependencies to other parts of the system. The central class does not necessarily contain much code. We detected that it is just a class, which represents a kind of a mediator for many other classes or even other modules and has to be changed often if other parts of the software change. Refactorings such as *Move Method* and *Move Field* can repair such a smell. Then the functionality can be pulled to the data and slim interfaces may be introduced.

Data Container: This smell is similar to the data containers defined in the "Move Behavior Close to Data" reengineering pattern of Demeyer et al. [1] that defines data containers as "classes defining mostly public accessor methods and few behavior methods (e.g., the number of methods is approximately two times larger than the number of attributes)". The difference is that in our change smell *Data Container* *two classes* make up the smell instead of a single one. One class holds all the necessary data whereas the second class interacts with other classes implementing functionality related to the data of the first class. This violates the principle of encapsulating data and their related functions. In our case, when two classes have common change patterns we should check for the reason. We detected situations where the change smell of *Data Container* was responsible for the unintended evolution of the software. This smell is detectable within the visualization when we encounter two classes, which have a strong change relation connecting them and additionally a net of other classes surround these two classes with weaker coupling. Usually, both the data container and the class with the interaction methods are related with each of the other classes. Hence, we obtain a lot of triangular relationships. The refactorings *Move Method* and *Extract Method* should be used to enrich the *Data Container* with behavior operating on the data and to combine the two classes into one. The aim of the improvement is that the data is well encapsulated.

3. CASE STUDY

A Picture Archiving and Communication System (PACS) was selected as case study for our approach. The PACS includes a viewing workstation, which supports concurrent displaying of pictures as well as an archive. The images are acquired from different modalities like magnetic resonance, or ultrasound scanning and save in distributed archive storages. The software is implemented in Java. The information of the whole application is maintained with the help of CVS.

All subsystems of the PACS can be viewed as separate projects that encapsulate some aspects of the whole application such as viewing unit, archiving process or extensions to the viewing unit. These extensions add diagnostic features to the viewing application. The case study is composed of 35 subsystems, each containing between one and fourteen modules.

Single classes represent the lowest level of decomposition. The history of the PACS system was inspected over a period 30 months. During this time the software grew from approximately 2000 to more than 5500 classes. At the end it was composed of over 500 000 LOC. Regarding these simple numbers the system seems quite well designed, as each class has less than 100 lines of code on the average.

4. CHANGE SMELL BASED REFACTORING

In this section we present an example from the case study, where we detect change smells and use refactorings to improve the evolvability of the software. During the analysis of the historical data received from CVS we identify a small module (i.e. Java package) with a high changing activity. So we calculate the logical couplings of this module called *jvision/workers*. To get a better understanding of logical couplings for the classes of *jvision/workers* we create a graphical representation (see Figure 1).

In this representation classes are depicted as small ellipses. The ellipses are grouped by their membership to modules. Modules themselves are depicted as bounding ellipses surrounding their classes. This structural information is enriched with historical data. From CVS we extract the evolution of classes and calculate logical coupling between classes. This coupling is depicted in Figure 1 through edges connecting the ellipses whereas the thickness of the edges describes the "strength" of the visualized couplings. The more often a pair of classes is changed at the same time the thicker is the representing edge. This visualization approach has been extended with class based metrics and implemented in EvoLens.

The navigation through the change couplings based on our visualization approach helps to locate the change smell *Man-in-the-middle* for the class *ImageFetcher*. This class has multiple strong logical couplings with other classes. The situation is even worse as it often changes together with classes of different packages. Thus, when a change has to be done by an engineer the editing is scattered over the software.

ImageFetcher is one of the largest classes of the entire system; it contains almost 2000 lines of code. The methods of this class are of exceptional length: Some of them contain more than 100 lines of code. When trying to reveal the reasons for such "spaghetti code", we discover that many methods are similar. Thus, the entire class is internally redundant. The length of the class itself does not automatically lead to the necessity of refactoring, but *ImageFetcher* often changes together with other classes. Thus, each change has to be thoroughly analysed in order not to miss any important change, which may be scattered over a large part of the system. This has a severe impact on the maintenance effort: When a bug is discovered within one of the methods of this class, many other methods have to be changed in a similar way. Often such changes are missed and have to be fixed later when the bug reoccurs. This results in a high changing activity.

Additionally, this class seems to have divergent changes [4], because it changes together with a lot of classes of other

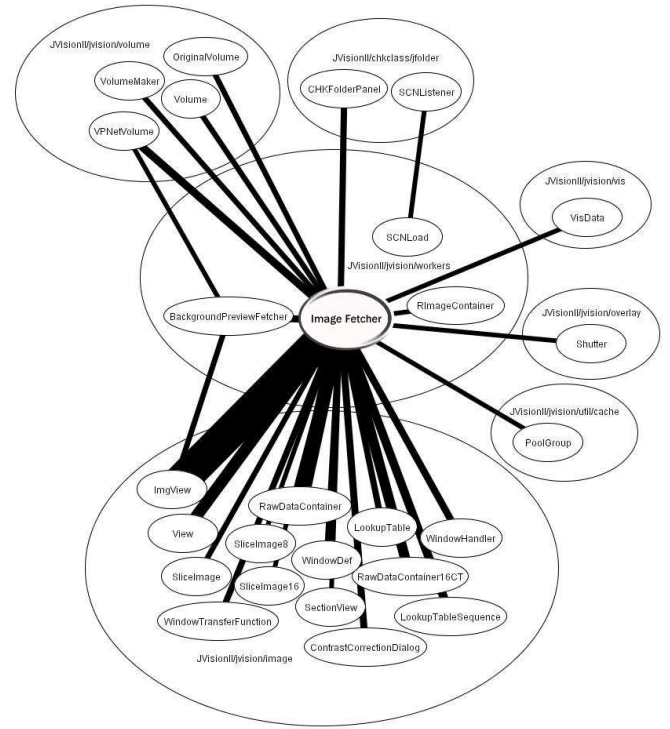


Figure 1: Change smell: Man-in-the-Middle

modules. Thus some methods seem stronger related with classes of a particular module, whereas other methods of *ImageFetcher* have to be changed in conjunction with classes of other modules. When inspecting the source of the *ImageFetcher* we determine that the principle of separation of concerns is violated. This class implements a thread pool, a queue for work items, and logic for loading images altogether. As a result, different classes implementing different functionality are related with *ImageFetcher*.

4.1 Refactoring to improve Evolvability

We apply several refactorings to reduce the weaknesses of this change smell. Then we continue to observe the evolution of the module *jvision/workers* to see if the evolvability has been improved through evolution guided refactoring.

To minimize code duplication, we first extract code clone parts of methods and reuse the newly formed methods where appropriate. For that, we apply the *Extract Method* refactoring that helps to get reusable items. After these improvements the class contains just 1100 lines of code, because of the removal of duplication.

To further improve the evolvability, we split *ImageFetcher* into new classes encapsulating the different concerns. We move the methods and data for image loading into a separate class called *FetchWorker*. The logics for thread pooling and the handling of the work queue are left together in *ImageFetcher*. After the movements we obtain a surprising result: *FetchWorker* contains just one public method called *loadimage()*. This simple interface results in reduced coupling. Also *ImageFetcher* has a simple interface after the

tions are another field where code smells have to be evaluated. jCOSMO [14] was developed to automatically detect code smells such as the ones defined by Fowler [4] and to visualize their distribution over the system.

Complementing to our approach in which we correct hot spots in the evolution of software systems, also the risk of a change to break an already existing feature can be assessed by analyzing software changes [9]. Ostrand et al. [10] predict the quality of certain parts of software. They estimate the number of faults per file for the next release based on a negative binomial regression model using information from previous releases. Additionally to source code repositories several other information sources such as mail messages and defect reports can be explored to get a better understanding how a software product has evolved since its conception [6]. In [7] four different kinds of studies for software evolution are presented and compared. The studies consider long-running observations of growth and evolution as well as fine grained issues like code cloning and software architectures.

6. CONCLUSIONS

We have shown an approach to exploit historical data extracted from repositories such as CVS in terms of change couplings: We adopted the concept of "bad smells" and provided additional *change smells* based on change coupling analysis. Such a smell is hardly visible in the code, but easy to spot when viewing the change history.

Based on these change couplings and the proposed change smells, the developer obtains support where to apply refactorings efficiently. In an industrial case study comprised of 500 000 LOC in Java, we have shown how these change smells can be cured and how refactoring can be based on them. It turned out that after the refactorings had been implemented, the evolution of the system, that we observed for another 15 months, was facilitated and did not lead to the originally strong change couplings or change smells. In talking to the developers, they stated that the directed refactorings were effective for them and the new interfaces and classes were much clearer and easier to use.

From this we conclude, that such an approach can help in improving the maintainability and evolvability of a large software system. The change coupling data itself to get is rather straightforward, as are the two described change smells *Man-in-the-Middle* and *Data Container*.

Our prototype tool EvoLens integrates many of these concepts already but it will be enhanced to better deal with change smells in the future. The next steps will further investigate change smells and their curing with appropriate refactorings.

7. ACKNOWLEDGMENTS

We thank Tiani Medgraph, our industrial partner, which provided the case study and helped us with the interpretation of the results. The work described in this paper was supported by the Austrian Ministry for Infrastructure, Innovation and Technology (BMVIT), The Austrian Industrial Research Promotion Fund (FFF), and the European Commission in terms of the EUREKA 2023/ITEA project FAMILIES (<http://www.infosys.tuwien.ac.at/Cafe/>).

8. REFERENCES

- [1] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-oriented Reengineering Patterns*. Morgan Kaufmann Publishers, An Imprint of Elsevier Science: San Francisco CA, USA, July 2002.
- [2] S. Demeyer and T. Mens. Evolution metrics. *Proc. of the 4th Int. Workshop on Principles of Software Evolution*, pages 83–86, 2001. Session 4A: Principles.
- [3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. *Proc. Int. Conf. on Software Maintenance*, pages 23–32, September 2003.
- [4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, June 1999.
- [5] H. Gall, J. Krajewski, and M. Jazayeri. CVS Release History Data for Detecting Logical Couplings. In *Proc. 6th Int. Workshop on Principles of Software Evolution*, pages 13–23. IEEE Computer Society Press, September 2003.
- [6] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softchange. *Proc. 16th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 336–341, June 2004.
- [7] M. Godfrey, X. Dong, C. Kapser, and L. Zou. Four interesting ways in which history can teach us about software. *Int. Workshop on Mining Software Repositories*, May 2004.
- [8] C. F. Kemerer and S. A. Slaughter. An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, 25(4):493–509, July-August 1999.
- [9] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, April-June 2000.
- [10] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. Where the bugs are. *Proc. on the Int. Symposium on Software Testing and Analysis*, pages 86–96, July 2004.
- [11] J. Ratzinger, M. Fischer, and H. Gall. Evolens: Lens-view visualizations of evolution data. *Technical Report: Vienna University of Technology*, December 2004.
- [12] F. Simon, F. Steinbrückner, and C. Lewernetz. Metrics based refactoring. *Proc. European Conf. on Software Maintenance and Reengineering*, pages 30–38, March 2001.
- [13] W. P. Stevens, G. J. Myers, and L. L. Constantine. Structured design. *IBM Systems Journal*, 13(2):115–139, May 1974.
- [14] E. van Emden and L. Moonen. Java quality assurance by detecting code smells. *Proc. of the 9th Working Conf. on Reverse Engineering*, pages 97–108, October 2002.
- [15] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. *Proc. Int. Conf. on Software Engineering*, pages 563–572, May 2004.

Linear Predictive Coding and Cepstrum coefficients for mining time variant information from software repositories

Giuliano Antoniol
RCOST- University Of Sannio
Via Traiano 1
82100, Benevento (BN), ITALY
+390824305526

antoniol@ieee.org

Vincenzo Fabio Rollo
RCOST- University Of Sannio
Via Traiano 1
82100, Benevento (BN), ITALY
+390824305526

f.rollo@unisannio.it

Gabriele Venturi
RCOST- University Of Sannio
Via Traiano 1
82100, Benevento (BN), ITALY
+390824305526

venturi@unisannio.it

ABSTRACT

This paper presents an approach to recover time variant information from software repositories. It is widely accepted that software evolves due to factors such as defect removal, market opportunity or adding new features. Software evolution details are stored in software repositories which often contain the changes history. On the other hand there is a lack of approaches, technologies and methods to efficiently extract and represent time dependent information. Disciplines such as signal and image processing or speech recognition adopt frequency domain representations to mitigate differences of signals evolving in time. Inspired by time-frequency duality, this paper proposes the use of Linear Predictive Coding (LPC) and Cepstrum coefficients to model time varying software artifact histories. LPC or Cepstrum allow obtaining very compact representations with linear complexity. These representations can be used to highlight components and artifacts evolved in the same way or with very similar evolution patterns. To assess the proposed approach we applied LPC and Cepstral analysis to 211 Linux kernel releases (i.e., from 1.0 to 1.3.100), to identify files with very similar size histories. The approach, the preliminary results and the lesson learned are presented in this paper.

Keywords

Software evolution, data mining.

1. INTRODUCTION

An intrinsic property of software is malleability: Software systems change and evolve at each and every level of abstraction and implementation during their entire life span from inception to phase out. This fact, calls for approaches, methods, and technologies to study evolution of software characteristics during the system life.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

The evolution of a software system is observable as changes in structural information (e.g. modular decomposition and relation between modules), behavioral information (e.g. functionalities, or bugs), and project information (e.g., maintenance effort). As these changes happen in time, software evolution can be modelled and studied as time series. A time series is a collection of measures recorded over time. Time series and time series based approaches have been successfully applied to many disciplines such as speech processing, computer vision, or stock market forecasting. Common to these disciplines is the need to detect the occurrence of similar phenomena evolutions over time. Therefore models and technologies developed to study time series and time dependant phenomena or signals can be applied to software engineering.

In applying time dependant models to software artifacts evolution our goal is the definition of a criterion to establish similarity or dissimilarity of artifact histories. Indeed, similarity is quite a crucial issue: there are several software engineering areas such as software evolution and maintenance, software analysis, software testing, or automatic Web Services composition where the ability to effectively compute a similarity between artifact histories can greatly help researchers and practitioners.

On the other hand, similarity computation is a difficult problem. Often, similarity discovering is hampered by the presence of some distortion in one dimension of data (e.g., time). This distortion can cause dissimilar instances seem similar and the opposite as well.

As an example, effort prediction in software development or maintenance requires both effort prediction and effort distribution forecasting (i.e, schedule) [9]. Traditional approaches focus on effort prediction assuming a relation, often linear [14], between metrics related to complexity and/or size and the effort [1] [2] [3] [10]. Often a simple figure quantifying the effort doesn't suffice. Effort distribution over time is a key issue for project planning and staffing, therefore is an important cost driver and a cause of organizational disruption.

Unfortunately, effort distribution forecasting is more difficult than effort prediction because discovering similarities between past projects effort distributions is hampered by several factors causing 'distortion' in the data if represented as evolving in a linear time.

While the overall effort in past maintenance projects is mainly related to high level software metrics [6][14], the effort distribution is determined by internal system dependencies and organizational issues. Internal system dependencies can easily induce ripple effects imposing constraints between activities, a

component must be changed *after* some other has undergone maintenance. Organizational issues like holidays, staffing decisions, reorganizations, and so on, can cause postponing of activities and impact on the effort distribution in an unpredictable way. Therefore, analysing past effort distribution to determine similarities among time histories can be a difficult task, since similarities among activities are hidden because of these factors, while spurious similarities can emerge for the same reason. In other words, automating similarity computation between artifact histories is a challenging and difficult task. Similar difficulties are present in other software engineering activities such as log file or user behaviour analysis.

The above example outlines the usefulness of robust similarity detection approaches, robust when the original data are distorted in time.

We present an approach to detect similarities between artifacts histories. In particular we aim at devising an approach to detect similarities in evolutions starting from past maintenance and activities effects, notwithstanding their temporal distortions. Theories and technologies to detect similarities in phenomena evolving in time, in a manner that the time rate can change among instances and also during a single instance are present in literature. In this work we applied one of these, namely LPC/Cepstrum, to mine from a repository of Linux kernel modules, files evolved in the same or very similar ways.

The remainder of this paper is organized as follow: first we present the background of the used approach, **Case study and results** section illustrate the application of the approach to the Linux kernel evolution data, and in **Discussion and future works** we debate about our results and indicate our future work guidelines

2. TACKLING TIME RATE CHANGES

Automatic speech recognition and speech synthesis researchers have a long history of wrestling with time distortion. Human beings change the rate of speech when talking (prosody), but humans recognize words also in presence of dramatic changes in pronunciation speed or accent during locution. When machines come into play, it is quite obvious expecting from them at least a similar ability in comprehension. Therefore, a speech recognition system must be robust with respect to time distortion as well as to disturbance (noise).

Among the speech recognition approaches the family based on Linear Predictive Coefficient and Cepstrum (LPC/Cepstrum) is prominent for its performances and its relative simplicity. LPC/Cepstrum, first proposed in [7] and subsequently in [12] and [13], models a time evolving signal as an ordered set of coefficients representing the signal spectral envelope. That is a curve passing close to the peaks of the original signal spectrum. To obtain the LPC/Cepstrum representation the first step is to compute Linear Predictive Coding (LPC) coefficients. These are the coefficients of an auto-regressive model minimizing the difference between linear predictions and actual values in the given time window.

The LPC analysis uses the autocorrelation method of order p .

In matrix form, we have

$$Ra = r$$

where

$$r = [r(1)r(2)..r(p)]^T$$

is the autocorrelation vector,

$$a = [a_1 a_2 \dots a_p]^T$$

is the filter coefficients vector and R is the $p \times p$ Toeplitz autocorrelation matrix, which is nonsingular and gives the solution

$$a = R^{-1}r.$$

Once LPC have been obtained it is possible to compute cepstra from them. Cepstra are the coefficients of the inverse Fourier transform representation of the *log* magnitude of the spectrum. The cepstra series represents a progressive approximation of the 'envelope' of the signal: as for LPC, the more are the cepstra considered the more the envelope adheres to the original spectrum.

Starting from a and r , we have as c_m coefficients (for order p):

$$c_0 = r(0),$$

$$c_m = a_m + \sum_{k=1}^{m-1} \frac{k}{m} c_k a_{m-k},$$

for $1 < m < p$, and

$$c_m = \sum_{k=m-p}^{m-1} \frac{k}{m} c_k a_{m-k},$$

where $m > p$.

In speech recognition LPC/Cepstrum has been proven capturing most of the relevant information contained in the original series. For a sequence of 30-300 points a number of 8-30 coefficients suffice for most application. Therefore, LPC/Cepstrum allows to obtain a very synthetic representation of a time evolving phenomenon. This compact representations can be used to efficiently compare signals, once a suitable distance measure has been defined between LPC or Cepstrum coefficients. Most approaches aiming to assess similarity between time series use the Euclidean distance among the LPC/Cepstrum representations as an indirect similarity measure. Although distance and similarity are different concepts, cepstral distance can be used to assess series similarity: If two cepstra series are "close", the original signals have a similar evolution in time. As an alternative to Cepstrum and Euclidean distance, it is possible to use the Itakura distance (a.k.a. Log Likelihood Ratio LLR) [4] that can be computed directly from LPC.

LPC/Cepstrum has been used also in computer vision and in other research fields [11]. For examples in [15] LPC/Cepstrum is

applied to online signatures verification and Euclidean distance between LPC/Cepstrum has been used as dissimilarity measure to cluster ARIMA series modeling electrocardiogram signals [5].

3. CASE STUDY AND RESULTS

We tested the application of LPC/Cepstrum to the evolution of a real world software system: the Linux kernel. Our goal was to verify if LPC/Cepstrum can be a starting point to produce compact representations of software modules evolution while preserving essential characteristics of the phenomena under study. In other words, if the spectral based representations could be applied to identify artifacts having very similar maintenance evolution histories. Being interested in mining the effect of maintenance on artefact but also in effort we selected a metric that is quite commonly recognized as strongly related to maintenance effort: size measure in LOC. Therefore our initial dataset was composed by the LOC histories, 211 releases, of 1788 files composing the Linux kernel from version 1.1.0 to 1.3.100 for 211 releases.

Over this dataset we performed LPC/Cepstrum analysis where the modules evolution in size was thought of as signals evolving in time. Once obtained LPC/cepstrum coefficients we computed the distance between each pair of module (that is about one million of module pairs). A method to be effective must efficiently produce results, our approach for the 1788 histories requires less than 5 seconds on a Pentium 4 machine at 1.6 GHz. The tools used in each phase are summarized in Table 1. These are all open source software integrated together allowing an almost fully automated analysis.

Table 1: Test case technologies and instruments

Phase	Instruments
Extraction of size modules evolution from CVS repository	Perl scripts
LPC computation	C program
Cepstra computation	C program
Euclidean distance computation	C program
Results classification and graph plotting	Perl script and GNUPlot

To produce useful results, a similarity assessment based on an abstraction and on a distance measure must respond to three minimal requirements:

- It has to discriminate among similar histories, allowing to identify some as similar and some as dissimilar by applying a threshold (such as the more restrictive the threshold the less the pairs deemed similar). The possibility to vary the threshold is important because similarity research often starts with a blur similarity definition gaining sharpness in late phases. Therefore it must be possible to customize similarity detection on the fly.
- It has to be sensible to the relative richness of the information supplied. With less information most items seem similar, increasing the information used we expect

fine grain dissimilarities to emerge. This is important because allows researchers to decide the best abstraction level for the case at hand.

- It has to respond to some intuitive and meaningful notion of similarity. Because similarity is not a value in themselves: similarities discovery is ancillary to other purposes for which a clear understanding of a similarity judgment is fundamental.

To address items a) and b) we calculate the sets of files with indistinguishable time series applying three distance thresholds (Euclidean distances less than $1 \cdot 10^{-3}$, $1 \cdot 10^{-4}$, and $1 \cdot 10^{-5}$) and four cepstra series lengths (12, 16, 20, 32). Since the more cepstra are used the more the envelope representation adhere to the original data, with less cepstra we expect to find more similar pairs and the opposite as well. The size of 8, 12, and 16, for both the LPC coefficients and the subsequent cepstra series, is a rule of thumb in speech coding. However, as this is the first application to software engineering of LPC/Cepstrum spectral representation, we decided to try the sizes from 12 to 32 to allow a richer signal representation. It should be noted that our thresholds are quite tight because the computed distances among software modules were far smaller than the ones among words in speech recognition.

By applying the above defined parameters we obtained Table 2, in which the number of files pairs deemed indistinguishable over a given threshold is shown for each combination of threshold value and cepstra series length.

Table 2. Number of pairs beating the thresholds for cepstra cardinality.

Threshold	Cepstra series cardinality			
	12	16	20	32
$1 \cdot 10^{-3}$	6045	4049	2897	1605
$1 \cdot 10^{-4}$	858	607	440	312
$1 \cdot 10^{-5}$	194	163	144	129

Table 2 responds to a) and b); the cardinality of the pairs considered undistinguishable is sensible to both threshold value and cepstra series length. These effects can be better appreciated in **Figure 1** and **2** reporting the impacts of thresholds and cepstra series length, respectively. Notice that both tables have a logarithmic Y axis thus quite different results are obtained with different configurations.

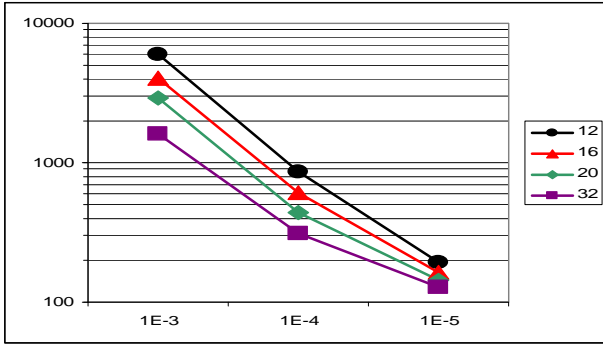


Figure 1. Impact of the threshold over the number of pairs deemed similar (logaritmik).

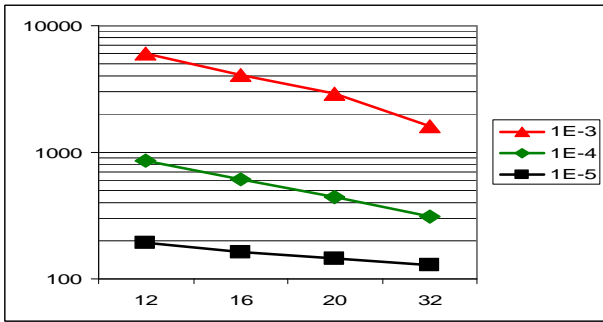


Figure 2. Impact of the cepstra series cardinality over the number of pairs deemed similar (logaritmik).

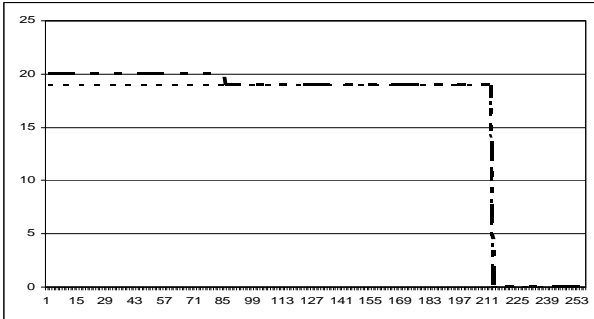


Figure 3. Less similar pair selected with 32 cepstra and a threshold of 10^{-5} .

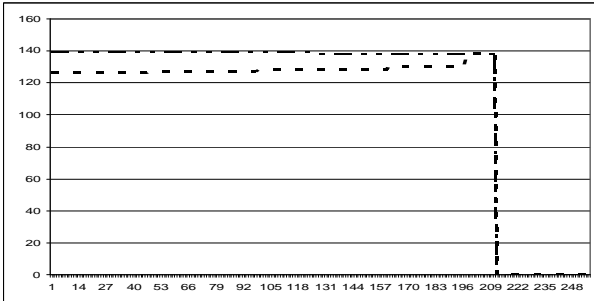


Figure 4. Less similar pair selected with 16 cepstra and a threshold of 10^{-5} .

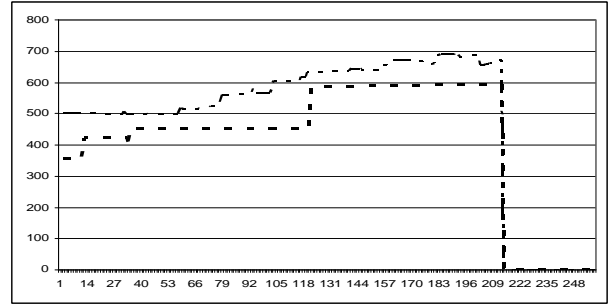


Figure 5. Most similar pair selected with 12 cepstra and a threshold of 10^{-2} that is discarded by more restrictive criteria.

To qualitatively assess whether the results of the automated analysis responds to some intuitive notion of distance and similarity (item c) we plotted the graphs of pairs classified as indistinguishable. Here we report three examples chosen to give an insight of how different configurations impact on distance and similarity. Figure 3, 4, and 5 report plots of the less similar pair selected with 32 cepstra and a threshold of 10^{-5} ; the less similar pair selected with 16 cepstra and a threshold of 10^{-5} ; and the most similar pair selected with 12 cepstra and a threshold of 10^{-3} not included in the sets selected by the other criteria. Notice that the Y axis aren't of the same scale.

The graphs show an appreciable progressive relaxation of the similarity as far as the cepstra series size is reduced and a less stringent threshold is applied.

4. DISCUSSION AND FUTURE WORKS

This work presents a case study assessing the suitability of LPC/Cepstrum to compare software artifacts evolutions. LPC/Cepstrum allows to obtain a compact representation of signals with linear complexity and to perform a robust comparison with respect to signal distortion. Computational efficiency, output compactness, and robustness are appealing characteristics for tools supporting software engineering activities. However, since the approach stems from a different research field, there is the need to assess its suitability. We conducted a first case study comparing evolution histories of 1788 Linux files at LOC level. We also defined three success criteria for the case study. To be deemed interesting for further explorations, the approach must: allow defining similarity thresholds, be sensible to the quantity of information used, and produce results responding to an intuitively understandable notion of similarity.

Indeed, we found that Euclidean distances computed among LPC/Cepstrum representations can be used to assess similarity in a way that is sensible to the richness of the representation and allows to define effective similarity thresholds. By inspecting histories we also verified that the sets of similar pairs selected with our approach respond to an intuitive notion of similar evolution in size. Therefore, the case study results show that LPC/Cepstrum is worth of further exploration by software engineering researchers.

An important theoretical issue is left aside from this case study. Distance measures are often seen and used as indirect similarity measures under the assumption that closeness between items is related to their similarity. This is a keystone of spectral representations use in speech recognition. In this case study we followed this approach as well. Nevertheless it must be pointed out that similarity and closeness remain two different concepts. From a theoretical perspective we believe that a better clarification of similarity between software artifact histories can be of great help in software evolution research.

A first further research step will be to apply the same framework to other software systems. In doing so we aim to gain knowledge about what are the cepstra containing the most relevant information, what are the threshold values most suitable for the various tasks and how the approach performs when metrics other than size are used. This should allow a broader understand of LPC/Cepstrum characteristics when applied in software engineering.

Spectral based representations support also comparisons with metrics other than Euclidean distance (e.g. the Itakura distance), and allow for further abstracting from data distortion in time (e.g. by means of time warping [8]). Exploring these alternatives it is possible to increase the robustness of the approach with respect to distortion and its flexibility with respect to the distance used.

Finally it is remarkable that LPC/Cepstrum has been successfully used also in situations in which data are distorted in dimensions other than time. This suggests the application to software engineering situation in which data are distorted in other dimensions as well (e.g. size or effort).

5. REFERENCES

- [1] Boehm, B.W. *Software Engineering Economics*. Prentice-Hall Inc., Englewood Cliffs, N.J., 1981.
- [2] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering*, vol. 1, 1987, 57-94.
- [3] Hastings, T.E., and Sajeev, A.S.M. A Vector-Based Approach to Software Size Measurement and Effort Estimation. *IEEE Transactions on Software Engineering*, vol. 27, no. 4, 2001, 337-350.
- [4] Itakura F., Minimum prediction residual principle applied to speech recognition, *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 23, pp. 67-72, Feb. 1975
- [5] Kalpakis K., Gada D., and Puttagunta V., "Distance Measures for Effective Clustering of ARIMA Time-Series". In *Proc. of the 2001 IEEE International Conference on Data Mining (ICDM'01)*, San Jose, CA, November 29-December 2, 2001, pp. 273-280.
- [6] Lindvall, M. Monitoring and Measuring the Change-Prediction Process at Different Granularity Levels: An Empirical Study. *Software Process Improvement and Practice*, no. 4, 1998, 3-10.
- [7] Markel, J.D. and Gray Jr, A.H. *Linear Prediction of Speech*. Springer-Verlag, New York, 1976.
- [8] Myers C.S. and Rabiner L.R. A comparative study of several dynamic time-warping algorithms for connected word recognition. *The Bell System Technical Journal*, 60(7):1389-1409, September 1981
- [9] Mockus A., Weiss D.M., Zhang P. Understanding and Predicting effort In Software Projects. *Proc. of the 25th International Conference On Software Engineering*, 2003, 274 - 284
- [10] Nesi, P. Managing Object Oriented Projects Better, *IEEE Software*, vol. 15, no.4. 1998, 50-60.
- [11] Oppenheim A.V and Schafer R.W, "From Frequency to Quefrency: A History of the Cepstrum", *IEEE Signal Processing Magazine*, September 2004.
- [12] Papamichalis, P.E. *Practical Approaches to Speech Coding*. Prentice Hall, Englewood Cliffs, NJ, 1987
- [13] Rabiner, L.R. and Juang B.H. *Fundamentals of Speech Recognition*. Prentice Hall, Englewood Cliffs, NJ, 1993
- [14] Ramil, J.F. Algorithmic Cost Estimation Software Evolution. *Proceeding of Int. Conference on Software Engineering*, Limerick, Ireland, IEEE CS Press, 2000, 701-703.
- [15] Wu, Q.Z., Jou, I.C., Lee, S.Y., Online Signature Verification Using LPC Cepstrum and Neural Networks, *IEEE Transactions on Systems, Man, and Cybernetics* (27), No. 1, February 1997, pp. 148-153.

Process and Collaboration

Repository Mining and Six Sigma for Process Improvement

Michael VanHilst
Dept. of Computer Science & Eng.
Florida Atlantic University
Boca Raton, Florida
1 954 661-1473

vanhilst@fau.edu

Pankaj K. Garg
Zee Source
1684 Nightingale Avenue, Suite 201
Sunnyvale, California
1 408 373-4027

garg@zeesource.net

Christopher Lo
Dept. of Computer Science & Eng.
Florida Atlantic University
Boca Raton, Florida
1 561 346-4749

chrishlo@yahoo.com

ABSTRACT

In this paper, we propose to apply artifact mining in a global development environment to support measurement based process management and improvement, such as SEI/CMMI's GQ(I)M and Six Sigma's DMAIC. CMM has its origins in managing large software projects for the government and emphasizes achieving expected outcomes. In GQM, organizational goals are identified. The appropriate questions with corresponding measurements are defined and collected. Six Sigma has its origins in manufacturing and emphasizes reducing cost and defects. In DMAIC, a major component of a Six Sigma approach, sources of waste are identified. Then changes are made in the process to reduce effort and increase the quality of the product produced. GQM and Six Sigma are complementary. Both approaches rely heavily on the measurement of input and output metrics. Mining development artifacts can provide usable metrics for the application of DMAIC and GQM in the software domain.

Categories and Subject Descriptors

D.2.9 [Software Engineering]: Management – *productivity, software process models.*

General Terms

Management, Measurement, Reliability, Theory.

Keywords

Six Sigma, GQM, Process Improvement, Repositories

1. INTRODUCTION

Six Sigma and CMMI are two different approaches to process improvement that come from different perspectives. The two approaches are complementary. Combining the strengths of each approach yields an approach that focuses strongly on continuous and incremental process improvement while seeking metrics that are appropriate to the reality of software development. Within this perspective, we propose that mining artifacts found in large software repositories can provide useful metrics to support a program of continuous process improvement. Mining artifact repositories provides useful process metrics without adding

overhead to the process being observed. Instrumenting artifacts, rather than people, supports other kinds of process improvement. While we have not yet put our ideas into practice, in this paper we explain our reasoning and place the proposal in the context of recent and historical trends in software and management theory. In future papers we will describe the experience of putting these ideas to use in a large software organization.

2. GQM, DMAIC, and Repository Mining

GQM is a disciplined approach to defining and collecting metrics as part of a software development process improvement program. Originally developed by Basili's group at the University of Maryland, it has since been adopted, slightly modified to GQ(I)M, as part of the guidelines for the SEI's CMMI. GQ(I)M stands for Goal-Question-(Indicator)-Measure. The 10 steps in a GQM process identify business goals, identify the questions to ask related to these goals, and measurements that will help answer them, and create a plan to collect the measurements. The CMMI and GQM focus on measuring and managing the development process to predictably and reliably achieve organizational goals.

Six Sigma is a disciplined approach to continuous process improvement designed to increase customer satisfaction and profits while reducing defects and cost. The name derives from the ideal of 3.4 defects per million opportunities. Organizations with a three sigma level of defects (typical of software) are candidates for improvement. Beyond six sigma, the investment is assumed not to be cost effective. Originally developed at Motorola, it has been popularized by many high profile companies including Honeywell, GE, 3M, Kodak, DuPont, and Allied Signal. Today it is widely applied to manufacturing and service-related processes. A good description of Six Sigma can be found on the SEI web site [21].

The origins of Six Sigma are instructive for software development. In 1985, Bill Smith argued that if a product was found defective and corrected during the production process, other defects were bound to be missed and found later by the customer during use of the product. This raised the question, was the effort to achieve quality really dependent on detecting and fixing defects, or could quality be achieved by preventing defects in the first place through manufacturing controls and product design? Smith's observation echoes the third of Deming's 14 points, not to rely on inspection and testing to achieve quality [3].

Six Sigma is an iterative approach based on undertaking a continuous series of initiatives to improve performance over time. The process improvement model is called DMAIC, an acronym for the following 5 steps.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

- 1) *Define* what is important. What matters to the customer?
- 2) *Measure* performance. How are we doing? What aspects of the process are affecting customer value?
- 3) *Analyze* opportunity. What could we be doing better? What are the variables that affect performance?
- 4) *Improve* the process. Plan a strategy for improvement and test it out.
- 5) *Control* the process. Institutionalize practices to sustain the improvement.

A key concept in Six Sigma is the “big Y”. What is the greatest gain in measurable customer value (measured on the y-axis), that can be achieved by an investment (measured on the x-axis) in process improvement. At the beginning of each initiative iteration the process is analyzed to find threats to customer satisfaction and opportunities for improvement. Traditionally, the measurement part of the process is based on practices of statistical quality control.

A typical example of the application of Six Sigma might involve light bulb manufacturing. The measure phase discovers that recently the variance in the thickness of the glass has been increasing. Continuation of this trend could lead to breakage in shipping and higher costs. The source of the variance is identified (worn machine part, new operator, supplier, etc.) and corrective action is undertaken.

The strengths and weakness of GQM and DMAIC are complementary [9]. Implementations of the CMM are sometimes criticized for emphasizing repeatability over improving productivity. Six Sigma is sometimes criticized for being inappropriate for development processes characterized by the unique intellectual efforts of knowledge workers. DMAIC’s strength is its focus on continuous process improvement and its iterative and incremental approach to achieving it. GQM’s strength is in defining metrics that are appropriate to the business goals and to the process. In this context, the kind of information that can be found in software repositories adds value.

Mining software development repositories can be used to detect weaknesses and identify opportunities to improve the development process. Repository measurements can be collected without adding significant process overhead. In the past, there has been an impediment to using the kind of data that can be collected and inferred from the mining of software repositories because of its perceived lack of methodological and statistical rigor. However, there is an emerging understanding within both the GQM [16] and Six Sigma [15] communities that this kind of data yields real value. “In rapidly changing environments, precise numbers and elaborate statistical analyses are often less valuable than simpler answers to insightful, well-directed questions” [15].

Moreover, recent theories in process management and process improvement place greater value on the kinds of knowledge that can be found by mining development repositories in the pursuit of process improvement. Theories such as Obsolete Theory [13], Lean Management [20], Theory of Constraints [7], and Agile methods teach us to focus more on execution and less on planning, reduce waste, look for bottlenecks, balance reliable measures with measures that show value, and embrace change as a strategic advantage.

The concept of waste in lean manufacturing is attributed to Toyota’s Taichi Ohno and Shigeo Shingo [17]. Waste is defined as any activity that consumes resources but delivers no value to the customer. Defects are a source of waste – once allowed to

occur, they require rework at best, and at worst, lead to less useful or returned products and unhappy customers. Delay is also a source of waste, not only from increased development cost, but also from opportunities missed in the marketplace and in resources not available to produce more value. In Six Sigma, Black Belt practitioners achieve their rating through training and proven experience, where proven experience comes from achieving measurable reductions in waste.

3. Software Development as Production

Software development in large organizations can often be viewed as a production process. A typical team develops multiple variations (possibly variations over time) of a core product. In [23] software product line development is compared to manufacturing cars, where the basic car can be varied in terms of engine, seats, upholstery, etc. (In fact, part of Toyota’s Lean Manufacturing is the SMED, Single Minute Exchange of Dies, concept of process retargeting for major variations.) When software development is viewed as production, features can be viewed as inventory. In this light, the Extreme Programming principle of “build the simplest thing” can be seen as a correlate of Ohno’s concept of Kanban or Just-In-Time inventory. (Test-first and pair programming correspond to Shingo’s Poka-Yoke or mistake-proofing, and source inspection, respectively.)

When software development is viewed as a production process, a valid question becomes, where are the bottlenecks? In software product line development, bottlenecks can be caused by poor architecture and code rot, problems with requirements, or linkages and dependencies between project elements. Inspecting development artifacts can be an effective aid to identify and measure potential bottlenecks.

If a project’s change history shows a pattern of recent changes affecting more than the usual number of sites, an architecture problem might be indicated. Recent additions could be of a type that the architecture does not well support. Decreasing localization of change could also be a sign of code rot – repeated change over time tends to make code progressively more brittle to additional modification. In either case, the area of modification could be a candidate for refactoring.

Analysis of the email or SMS archives could reveal a volume of messages between developers and the internal customer prior to progress being made on specific features or requests. This pattern could indicate a problem with the process of requirements capture or specification. Further analysis of the types of features involved and the nature of the misunderstanding would be warranted.

Standard product line domain analysis practices, e.g. [12], are facilitated by the analysis of artifacts. A pattern of a high frequency of modification on the same pieces of code across multiple variants could indicate an opportunity to save effort by building a code generator to handle the differences [23]. Two pieces of code that often change together might indicate high affinity or coupling, while code artifacts that seldom changes together exhibit the opposite. Code sections that rarely see change are good candidates for inclusion in the core domain architecture. Analyses of these types can help build effective architectures that better support product line and model driven development.

Frequent use of manuals or searching the web may reveal an opportunity for training on the issues in question. Similarly a comparison of artifacts between two teams, where one team is

consistently more productive than the other, might reveal types of training that would best aid the less-performing team.

Reducing defects can also be improved through inspection of process artifacts. Correlating defect reports with prior activities may indicate opportunities to reduce defects through process change. By analyzing sequences of behavior, it might be possible to identify where development shortcuts have been taken. Leveson's STAMP model for reliably safe systems assigns the root cause of system failures to failures in constraints on the process. Using this model, artifact evaluation could identify patterns of violating the constraints before they lead to defects in the product.

As software development organizations mature to CMMI levels 3, 4, and 5, their process artifacts contain more keys for correlation. Change events refer to change requests, and communications more often reference specific features, requests, and code. It is likely that as organizations use and find value in artifact analysis, properties of the artifacts that enhance their value for analysis will improve. The process we propose corresponds the CMM level 5 Technology Change Management, but adds specific measurement practices to drive the process.

4. MINING GLOBAL SOFTWARE DEVELOPMENT ENVIRONMENTS

In the past, approaches such as DMAIC and GQM have advocated putting measurement practices in place that collect measurements to feed the overall method. We think that such instrumentation approaches suffer from two main drawbacks: (1) they introduce measurement overheads in the process that can slow the process, and, more seriously, (2) they reify measurements and their instrumentations, affecting the behavior of the process and its participants. In contrast, we advocate that appropriate measurements be mined from the existing process and product data.

Fortunately, the existence of global software development environments (GDE), like SourceForge [22], and Corporate Source [4,5] and its successor SourceShare [24], provide ample opportunities to collect appropriate data. A GDE provides a repository for multiple projects in an organization to store all project information in a single place [11]. Participants create a new project in a GDE, and subsequently all project communication (through email or discussion forums), version control data, and problem report workflows are captured and maintained in the GDE. We propose that GDEs can be extended with a DMAIC dashboard to interactively provide required metrics and analyses.

Since we do not have practical experience with this approach yet, we give some hypothetical examples of analyses and measurements that could be usefully mined from GDEs. One of the main tenets of Six Sigma is to reduce the number of defects per million opportunities in a product. In the case of software development, the opportunities for introducing defects are numerous, ranging from the abstract (error in understanding a requirement) to concrete (error in a program statement). Therefore, one category of charts that will be useful addition to GDE would be running charts of open defects per opportunity, e.g., open defects per thousand lines of source code. As the lines of code progress over time, and the defects are opened and closed, these charts can give a sense of how the process is maturing over time.

In the spirit of Open Source, a GDE advocates that users (or customers) of a software project have early and continuous visibility of the process. Hence, potential users participate in the email lists for discussions on feature requests and design changes. These discussions can provide a useful measurement of how involved are the users in the process? One can measure the number of emails coming from users versus developers over time.

5. RELATED WORK

5.1 Effort Estimation

Previous works on effort estimation have been focused on the metrics from the development of an entire system. The AMEffMo [10] project has shown that it is possible to estimate the amount of effort that went into four separate projects using the metrics that was gathered from each project. It should stand to reason that effort estimations for individual components of a project are also possible. In an evaluation study performed by Mockus and Graves [2] they set forth an algorithm that is able to estimate effort based on the size of a modification request as well as the type of change requested. It was even stated that if the effort for each change was known, then the size of the change would be known. However, the reliability of developer recorded efforts per module is questionable [8]. Therefore, effort was divided among all the changes performed within a given period.

Four variables were found to be significant in affecting the effort estimation model. The number of changes per modification request, individual developer productivity, the nature of the changes, and the time difference between the detection of decay and the request for the change [19]. In addition to these four main variables, other metrics can be used to measure effort such as the requirements or specification documents. [14] These were the major factors in this particular project and may be used as a starting point for investigating the cause of bottlenecks in a development process.

5.2 Communication Gap

As email is a viable platform for communication among developers of a system, these messages may become important information in understanding the difficulties of developing certain features or modules of a system. The storing of these email messages into a database and later mining their contents has been proven to be possible in the Apache Web server project. Since these email messages follow a relatively structured format with information regarding the sender, receiver, date, and subject, these attributes have been shown to be useable as search variables in a database query. Furthermore, the dates of these messages may be matched to the development timeframe of a particular feature in order to analyze bottlenecks and causes of increased effort during development. The number of developers who participated in changes or development can also be found through these techniques.

In addition to email messages, pools of information are located in the change logs of a CVS repository. In a study of the CVS repositories of an open source project, Mozilla and Bugzilla, [6] the large scale and ongoing nature of the project did not affect the mining. All that was needed was a time frame for which to analyze the data. This time frame restriction might also help narrow down changes performed at the same time as the development period for a feature or module that is being analyzed. Usually associated with each ChangeLog is a Bugzilla bug report which is free formed text written by the developer. These might

also indicate where time was spent and what difficulties were encountered.

6. CONCLUSION

Large repositories of software development artifacts contain a potential wealth of information about the behavior and performance of software development processes. This data is available without adding overhead to the process in question. Using this knowledge effectively requires an organizational commitment to change, and a context for asking the right questions. We believe that the combination of Six Sigma's DMAIC and CMMI's GQ(I)M, provides such a framework. We have explained the rationale and discussed recent trends in project management theory that add support to our view. As we are only now beginning to apply our ideas in an industrial setting, reports on our experience are left to future publication.

7. REFERENCES

- [1] Alonso, O., Gertz, M., and Devanbu, P. "Database Techniques for the Analysis and Exploration of Software Repositories" *MSR '04: International Workshop on Mining Software Repositories*, Edinburgh, UK, 2004.
<http://www.cs.ucdavis.edu/~devanbu/msr04.pdf>
- [2] Atkins, D., Ball, T., Graves, T., and Mockus, A. "Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor." *IEEE Transactions on Software Engineering*, 28(7), July 2002, 625-637.
<http://www.research.avayalabs.com/user/audris/papers/vedraft.pdf>
- [3] Deming, W.E., *Out of the Crisis*, MIT Press, Cambridge, MA, 1986
- [4] Dinkelacker, J., Garg, P.K., Miller, R., and Nelson, D. "Progressive Open Source." In *Proceedings of the International Conference on Software Engineering (ICSE'02)*. Orlando: ACM Press, 2002, 177-184.
<http://lib.hpl.hp.com/techpubs/2001/HPL-2001-233.pdf>
- [5] Garg, P.K. and Dinkelacker, J. "Applying Open Source Concepts Within A Corporation." *1st ICSE International Workshop on Open Source Software Engineering*, Toronto, Canada, May, 2001.
http://sunarcher.org/jamie/pubs/OpenSourceInCorpEnvs_2001.pdf
- [6] German, D.M. "Mining CVS Repositories: The SoftChange Experience." In *1st International Workshop on Mining Software Repositories*. May 2004, 17-21.
<http://turingmachine.org/files/papers/2004/dmgmining2004.pdf>
- [7] Goldratt, E.M. *The Goal: A Process of Ongoing Improvement*, 2nd rev. ed. North River Press, 1992.
- [8] Graves, T.L. and Mockus, A., "Inferring Change Effort from Configuration Management Data." In *Metrics 98: Fifth International Symposium on Software Metrics*, Bethesda, Maryland, November 1998, 267-273.
<http://www.research.avayalabs.com/user/audris/papers/effort>
- [9] Hong, G.Y. and Goh, T.N. "A Comparison of Six Sigma and GQM Approaches in Software Development." *Journal of Six Sigma and Competitive Advantage*, 1(1), 2004,
<http://www.inderscience.com/storage/f125119371042861.pdf>
- [10] Huffman Hayes, J., Patel, S., and Zhao, L., "A Metrics-Based Software Maintenance Effort Model" In *Proceedings of the 8th European Conference on Software Maintenance and Reengineering*, Tampere, Finland, March 2004. pp. 254-258.
http://selab.netlab.uky.edu/Homepage/csmr_ameffimo_hayes_2004%5Eas_published.doc
- [11] Inoue, K., Garg, P.K., Iida, H., Matsumoto, K. and Torii, K.. "Mega Software Engineering." Accepted for *PROFES 2005*, Finland, June 2005
- [12] Jacobson, I., Griss, M.K., and Jonsson, P. *Software Reuse: Architecture, Process, and Organization for Business Success*. Addison-Wesley, Reading, MA, 1997
- [13] Koskela, L., and Howell, G. "The Underlying Theory of Project Management is Obsolete." In *Proceedings of the PMI Research Conference*, 2002, 293-302.
<http://www.leanconstruction.org/pdf/ObsoleteTheory.pdf>
- [14] Lehman, M.M., Perry, D.E., and Ramil, J.F. "Implications of Evolution Metrics on Software Maintenance." *ICSM'98*, November 1998.
<http://www.ece.utexas.edu/~perry/work/papers/feast2.pdf>
- [15] Martin, R. "Validity vs. Reliability: Implications for Management." *Rotman Magazine*, Winter 2005.
<http://www.rotman.utoronto.ca/integrativethinking/ValidityVSReliability.pdf>
- [16] Morasca, S., Briand, L.C., Basili, V.R., Weyuker, E.J. and Zelkowitz, M.V. "Comments on 'Towards a Framework for Software Measurement Validation'." *IEEE Transactions on Software Engineering*, 23(3), March 1997, 187-188
- [17] Ohno, T. *The Toyota Production System: Beyond Large-Scale Production*. Productivity Press, 1988.
- [18] Park, R.E., Goethert, W.B., and Florac, W.A. *Goal-Driven Software Measurement — A Guidebook*, Software Engineering Institute, 1996.
<http://www.sei.cmu.edu/pub/documents/96.reports/pdf/hb002.96.pdf>
- [19] Perpich, J.M., Perry, D.E., Porter, A.A., Votta L.G., and Wade, M.W. "Anywhere, Anytime Code Inspections: Using the Web to Remove Inspection Bottlenecks in Large-Scale Software Development." *1997 International Software Engineering Conference (ICSE97)*, Boston Mass, May 1997.
<http://www.ece.utexas.edu/~perry/work/papers/icse97.pdf>
- [20] Poppendieck, M. and Poppendieck, T. *Lean Software Development: An Agile Toolkit*. Addison-Wesley, Reading MA, 2003.
- [21] Sivi, J. "Six Sigma." Software Engineering Institute, 2001.
http://www.sei.cmu.edu/str/descriptions/sigma6_body.html
- [22] <http://www.sourceforge.net>
- [23] Weiss, D. and Lai, C.T.R. *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wesley, Boston, MA, 1999

Mining Version Histories to Verify the Learning Process of Legitimate Peripheral Participants

Shih-Kun Huang^{1,2}
skhuang@csie.nctu.edu.tw

Kang-min Liu¹
gugod@gugod.org

¹Department of Computer Science and Information Engineering
National Chiao Tung University, Hsinchu, Taiwan

²Institute of Information Science, Academia Sinica, Taipei, Taiwan

ABSTRACT

Since code revisions reflect the extent of human involvement in the software development process, revision histories reveal the interactions and interfaces between developers and modules.

We therefore divide developers and modules into groups according to the revision histories of the open source software repository, for example, `sourceforge.net`. To describe the interactions in the open source development process, we use a representative model, Legitimate Peripheral Participation (LPP) [6], to divide developers into groups such as core and peripheral teams, based on the evolutionary process of learning behavior.

With the conventional module relationship, we divide modules into kernel and non-kernel types (such as UI). In the past, groups of developers and modules have been partitioned naturally with informal criteria. In this work, however, we propose a developer-module relationship model to analyze the grouping structures between developers and modules. Our results show some process cases of relative importance on the constructed graph of project development. The graph reveals certain subtle relationships in the interactions between core and non-core team developers, and the interfaces between kernel and non-kernel modules.

Keywords: Legitimate Peripheral Participants(LPP), Open Boundary, Open Source Software Development Process.

1. INTRODUCTION

Because of the success of Linux, GNU, Apache, and tens of thousands of open source development (OSD) projects in `sourceforge.net`, we review the process of OSD and compare it with conventional approaches to proprietary software development. Many researchers have explored and tried to explain the differences between the software processes of OSD and conventional approaches. Among them, Eric S.

Raymond was the first to publish his findings in the noted *Cathedral and Bazaar* [11] model.

Ye and Kishida also proposed an open source software(OSS) development process model [15]. It is based on the evolving nature of a community with projects and a learning theory – *Legitimate Peripheral Participation (LPP)*, proposed by Lave and Wenger [6]. In [15], an OSS project may be associated with a virtual community, and developers may play certain roles in both the community and the project. During the learning process, the role of each member of the virtual community co-evolves in both the project and the community.

Few of the criteria of conventional software engineering methods, which are concerned with process models and control of schedules, can be applied in open source project development. In OSD, developers of a project may work together without knowing each other and build a successful system with millions of users worldwide. Although OSD does not appear to allow complete control and scheduling over software, it works well in reality. Besides, OSD projects often release new versions of software that are comparable to high quality proprietary software with similar functions. Such sustainable nature of the OSD process is worth exploring. However, although OSD has low initial deployment costs, there may be higher long-term costs.

In our experience, many open source developers do not contribute a great deal to OSD. They only do relatively minor work, such as fixing non-critical bugs, and do not make major contributions to the development process. Even so, although such minor contributors form weak links in developer networks, they are often a major driving force behind a project growing larger. This is similar to the small-world phenomenon.

The project-community evolutionary model, proposed by Ye and Kishida [15], states that any change of roles in the community maps to a change of roles in the project. The model also lists eight possible project roles and states that users, or peripheral developers, change their roles by learning about the project in detail, and are therefore central to the project.

In this paper, we propose a quantitative approach to analyzing the data of open source project development in order to evaluate the role changes of developers in a project.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MSR'05, May 17, 2005, Saint Louis, Missouri, USA Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

Through this analysis, we verify the learning process of LPP and provide a quantitative measurement for open source development models. The major advantage of our approach is that it is fully automatic; thus, manual verification is not required in the middle of the data mining process.

We believe that, in each open source project, there is a large amount of source code that does not need to be opened; that is, the success of an open source project depends on only a small proportion of its code. This would allow commercial developers of software to work with peripheral teams in the development of products without losing control of their source codes.

2. METHODS

We use a similar approach to that of Luis et al [7]. For each target project, we perform network analysis of its version control repository. Our main source of data is `sourceforge.net`, which provides a full CVS repository archive.

From revision histories, we can construct social network graphs that represent the relations between developers of different parts of a project. The evolutionary pattern of a social network reflects some process features and anomalies during a project's evolution. With network analysis methods, we can measure the relative importance of each developer, and classify each one's role.

For each path, p , found in the revision log, we define a developer set, D_p , for path p (such paths refer to directories specified in the revision log.) Formally, we define a developer network graph as follows.

$$D_p = \{d | \text{developer } d \text{ has modified path } p\}.$$

Then, we can define a symmetric developer graph, G_d , as:

$$\begin{aligned} G_d &= \{V_d, E_d\} \\ V_d &= \{d | d \text{ is a developer}\} \\ E_d &= \{(d_1, d_2) | \exists \text{ path } p \text{ s.t. } d_1 \in D_p \text{ and } d_2 \in D_p\}. \end{aligned}$$

In [7], the affiliation graph group is associated with the source code modules, while our group is associated with the directory. Our approach requires relatively less prior knowledge about the source code itself, and is more independent in terms of programming language; hence, it does not require human involvement to decide the affiliation group, as every step can be processed automatically.

We use the following definition in our analysis.

Distance Centrality (D_c) [12] : also called *closeness centrality*. The higher the value of D_c , the closer the vertices are to each other. Given a vertex, v , and a graph, G , D_c is defined as:

$$D_c(v) = \frac{1}{\sum_{t \in G} d_G(v, t)}. \quad (1)$$

For each project, we first generate the developer social network, then compute the distance centrality of each node.

From the distribution of the centrality values, we can discover the properties of different stages in the project development process.

3. RESULTS AND DISCUSSIONS

Figure 1 shows the developer social network for the project **awstats** [3]. Although this is a typical small project with only three developers, it has been very active according to `sourceforge.net`'s records. Its social network is fully connected, which means that all developers co-develop at least one directory. Project **phpmyadmin** [9] also has this kind of developer social network (Figure 2). In such a network, it is impossible to determine the importance of each developer, because they all have exactly the same attributes. Hence, we say that each developer plays the same role in the development process.

The above network pattern may reflect a possible flaw in our analytical method, because grouping developers based on directories is not detailed enough. However, it is also possible that the design of the software lacks proper modularity so that developers cannot modify a feature without modifying many directories in the source code.

The results of project **moodle** [8] (Figure 3) demonstrate another extreme case of social network patterns. A vertex's color represents its distance centrality value; the darker the color, the higher the centrality value. Nodes with the highest centrality values are rectangular in shape. The central portion of a node has only one vertex and all other vertices connect directly to that vertex. There are very few connections between non-central vertices. Project **filezilla** [5] (Figure 4) is another example of this kind of pattern.

Projects with this pattern start with a few developers deciding to work together, and they keep control of the source code as the project grows bigger. Non-central developers only make relatively minor contributions.

Nearly all projects with more than 10 developers have the same social network pattern as project **gallery** [1] (Figure 5). In such a pattern, only a small group of developers have a relatively high distance centrality, i.e., they are the center of the developer relationships; other developers play peripheral or intermediate roles. Project **bzflag** [13] (Figure 6) is another example of this kind of pattern.

Such social networks have many distance centrality values, which reflect many different kinds of project roles. Developers with high centrality values play important roles (core members or active developers), while those with lower values play peripheral roles (peripheral developers or bug fixers.)

Ye and Kishida [15] propose a project-community co-evolution process model, and define eight different roles in an open source project: **Project Leader**, **Core Member**, **Active Developer**, **Peripheral Developer**, **Bug Fixer**, **Bug Reporter**, **Reader**, and **Passive User**. Although, from the repository mining process, we are unable to associate each developer with a certain role, we can at least group developers into two large categories: active developer and above; and peripheral developer and below.

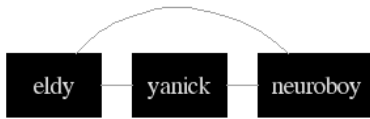


Figure 1: The awstats developer social network

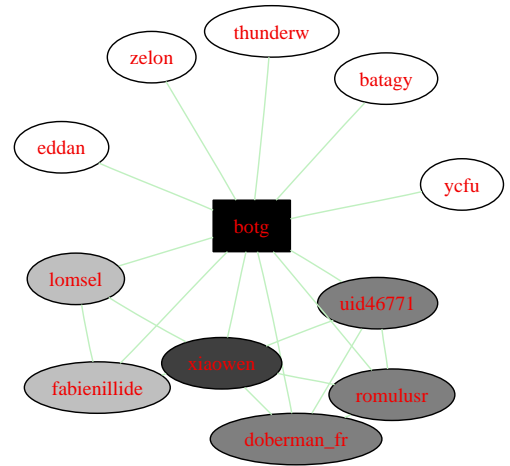


Figure 4: The filezilla developer social network

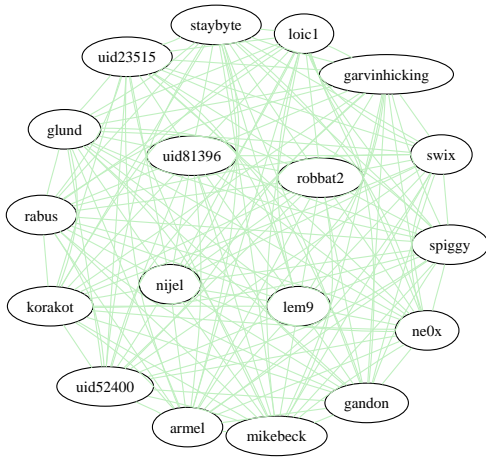


Figure 2: The phpmyadmin developer social network

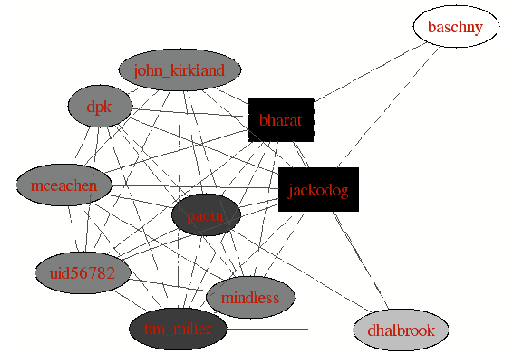


Figure 5: The gallery developer social network

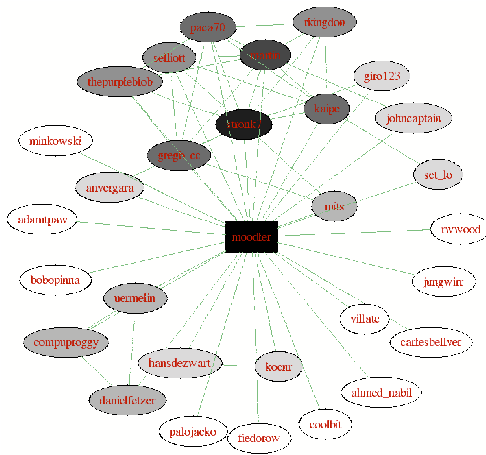
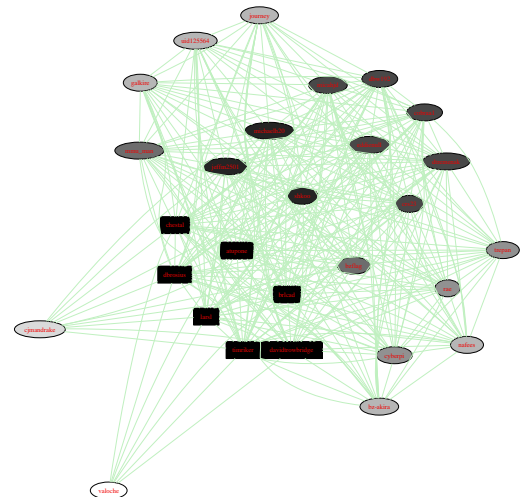


Figure 3: The moodle developer social network



4. RELATED WORK

In 1999, Eric Steven Raymond proposed the community-based development model in his famous work *Cathedral and Bazaar* [11]. In this work, he takes the development process of the `fetchmail` project as an example and proposes the bazaar process development model.

Ye and Kishida [15], state that, in an open source project, “Every user is a potential developer,” and propose a role hierarchy to show that participation in a project is actually a learning process for both peripheral users and core developers.

Project Bloof [10] gives a statistical revision log analysis for the source code evolution of a software project. The aim of Bloof is to help people comprehend software systems and the underlying development processes.

Project CVSMonitor [4] provides a more comprehensive presentation of revision analysis of the CVS repository, a version control system that has been widely used in the last ten years.

Zimmermann et al [16] recently proposed that mining version control histories can be helpful during the project development process, as they give programmers information about all the changes of a given revision.

White and Smyth [14] discuss several methods for analyzing large and complex network structures. In their experiments, they evaluated the different properties of many algorithms on toy graphs and demonstrated how their approach can be used to study the relative importance of nodes in real-world networks, including a network of interactions among the September 11th terrorists, a network of collaborative research in biotechnology among companies and universities, and a network of co-authorship relationships among computer science researchers.

Scacchi and Jensen [2] use techniques that exploit advances in artificial intelligence to discover the development processes of publicly available open source software development repositories. Their goal is to facilitate process discovery in ways that use less cumbersome empirical techniques and offer a more holistic, task-oriented process than current automated systems provide.

5. CONCLUSION

In this work, we use social network analysis methods to analyze the developer social network of a project created from the project’s revision history.

We then try to verify the LPP process in Ye and Kishida’s work [15]. Although this is not very accurate, we can at least split project developers into two groups: core and peripheral. This supports our conjecture that even in an open source project, there is a part of the source code that can be retained by core members only. With further graph-based network analysis, we believe that it would be possible to achieve more accurate results.

Developers involved in the revision process reveal their skill and familiarity with the source modules by different degrees

of interfacing and interaction with core members. From the revision histories, we build a link structure between developers and code modules and analyze the relationships between these structures to determine their level of involvement with core teams and kernel modules. The extent of developers’ involvement can be ranked. From the ranking results, we can verify the LPP learning process and propose a potential boundary between conceptual kernel and non-kernel modules. This boundary gives a clear indication of the degree of source code openness in joint development projects involving core and non-core teams of developers. The weak links around the boundary may significantly affect the ability of external peripherals to maintain the project’s vitality and popularity. Our preliminary results reveal a few such process cases of relative importance on the constructed graphs that could affect a project’s development.

6. REFERENCES

- [1] Chris Smith Bharat Mediratta. Gallery. a slick, intuitive web based photo gallery with authenticated users and privileged albums, 2000. <http://sourceforge.net/projects/gallery/>.
- [2] Walt Scacchi Chris Jensen. Data mining for software process discovery in open source software development communities. In *Proc. Workshop on Mining Software Repositories*, page 96, 2004.
- [3] Laurent Destailleur. Awstats is a free powerful and featureful server logfile analyzer, 2000. <http://sourceforge.net/projects/awstats/>.
- [4] Adam Kennedy. Project cvsmonitor. cvsmonitor is a cgi application for looking at cvs repositories in a much more useful and productive way, 2002. <http://ali.as/devel/cvsmonitor/>.
- [5] Tim Kosse. Filezilla is a fast ftp and sftp client for windows with a lot of features. filezilla server is a reliable ftp server, 2001. <http://sourceforge.net/projects/filezilla/>.
- [6] J. Lave and E. Wenger. *Situated Learning: Legitimate Peripheral Participation*. Cambridge university Press, Cambridge, 1991.
- [7] Jesus M. Gonzales-Barahona Luis Lopez-Fernandez, Gergorio Robles. Applying social network analysis to the information in cvs repositories. In *MSR2004*, 2004.
- [8] Eloy Lafuente Martin Dougiamas. Moodle is php courseware aiming to make quality online courses (eg distance education) easy to develop and conduct., 2001. <http://sourceforge.net/projects/moodle/>.
- [9] Loïc Chapeux Oliver Müller, Marc Delisle. phpmyadmin is a tool written in php intended to handle the administration of mysql over the web. <http://sourceforge.net/projects/phpmyadmin/>.
- [10] Lukasz Pekacki. Project bloof. bloof is an infrastructure for analytical processing of version control data, 2003. <http://sourceforge.net/projects/bloof/>.

- [11] Eric Steven Raymond. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly, 1999.
- [12] Gert Sabidussi. *The centrality index of a graph*, volume 31, pages 581–603. Psychometrika, 1966.
- [13] David Trowbridge Tim Riker. Opensource opengl multiplayer multiplatform battle zone capture the flag. 3d first person tank simulation, 2000.
<http://sourceforge.net/projects/bzflag/>.
- [14] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 266–275. ACM Press, 2003.
- [15] Yunwen Ye and Kouichi Kishida. Toward an understanding of the motivation open source software developers. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 419–429. IEEE Computer Society, 2003.
- [16] T. Zimmermann, P. Weigerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *26th International Conference on Software Engineering (ICSE 2004.)*, 2004.

Taxonomies & Formal Representations

Towards a Taxonomy of Approaches for Mining of Source Code Repositories

Huzefa Kagdi, Michael L. Collard, Jonathan I. Maletic
Department of Computer Science
Kent State University
Kent Ohio 44242
{hkagdi, collard, jmaletic}@cs.kent.edu

ABSTRACT

Source code version repositories provide a treasure of information encompassing the changes introduced in the system throughout its evolution. These repositories are typically managed by tools such as CVS. However, these tools identify and express changes in terms of physical attributes i.e., file and line numbers. Recently, to help support the mining of software repositories (MSR), researchers have proposed methods to derive and express changes from source code repositories in a more source-code “aware” manner (i.e., syntax and semantic). Here, we discuss these MSR techniques in light of what changes are identified, how they are expressed, the adopted methodology, evaluation, and results. This work forms the basis for a taxonomic description of MSR approaches.

Categories and Subject Descriptors

D.2.7. [Software Engineering]: Distribution, Maintenance, and Enhancement – *documentation, enhancement, extensibility, version control*

General Terms

Management, Experimentation

Keywords

Mining Software Repositories, Taxonomy, Survey

1. INTRODUCTION

Software version history repositories are currently being extensively investigated under the umbrella term *Mining of Software Repositories* (MSR). Many of the repositories being examined are managed by CVS (Concurrent Versions System). In addition to storing difference information across document(s) versions, CVS annotates code commits, saves user-ids, timestamps, and other similar information. However, the

differences between documents are expressed in terms of physical entities (file and line numbers). Moreover, CVS does not identify/maintain/provide any change-control information such as grouping several changes in multiple files as a single logical change. Neither does it provide high-level semantics of the nature of corrective maintenance (e.g., bug-fixes).

Researchers have identified the need to discover and/or uncover relationships and trends at a syntactic-entity level of granularity and further associate high-level semantics from the information available in the repositories. Recently, a wide array of approaches emerged to extract pertinent information from the repositories, analyze this information, and derive conclusions within the context of a particular interest.

Here, we present our analyses showing the similarities and variations among six recently published works on MSR techniques. These examples represent a wide spectrum of current MSR approaches. Our focus is on comparing these works with regards to the following three dimensions:

- Entity type and granularity
- How changes are expressed and defined
- Type of MSR question.

Further, we define notation to describe MSR in an attempt to facilitate a taxonomic description of MSR approaches. Finally, we outline the MSR process in terms of the underlying entities, changes, and information required to answer a high-level MSR question. We believe this work provides a better insight of the current research in the MSR community and provides groundwork for future direction in building efficient and effective MSR tools.

The remainder of the paper is organized as follows: section 2 discusses the various MSR approaches, section 3 gives a formal definition of MSR, section 4 outlines the MSR process and requirements, and finally we draw our conclusions.

2. APPROACHES TO MSR

A number of approaches for performing MSR are proposed in the literature. Here, we discuss these techniques with regards to the identified entities, questions addressed, evaluation, and results.

2.1. MSR via CVS Annotations

One approach is to utilize CVS annotation information. In the work presented by Gall et al [2, 3], common semantic (logical and hidden) dependencies between classes on account of addition or modification of a particular class are detected, based on the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

version history of the source code. A sequence of release numbers for each class in which it changed are recorded (e.g., class A =<1, 3, 7, 9>). The classes that changed in the same release are compared in order to identify common change patterns based on the author name and time stamp from CVS annotations. Classes that changed with the same time stamp (in a 4 minute window) and author name are inferred to have dependencies. In summary, this work seeks answers to the following representative questions:

- Which classes change together?
- How many times was a particular class changed?
- How many class changes occurred in a subsystem (files in a particular directory)?
- How many class changes occurred across subsystems?

This technique is applied on 28 releases of an industrial system written in Java with half a million LOCS. The authors reported that the logical couplings were revealed with a reasonable recall when verified manually with the subsequent release. The authors suggest that logical coupling can be strengthened by additional information such as the number of lines changed and the CVS comments.

In another study, the file-level changes in mature software (the email client *Evolution*) are studied by German [4]. The CVS annotations are utilized to group subsequent changes into what is termed a modification request (*MR*). Here, the focus is on studying bug-*MRs* and comment-*MRs* to address the following questions:

- Do *MRs* add new functionality or fix different bugs?
- Are *MRs* different in different stages of evolution?
- Do files tend to be modified by the same developer?

Further effort was on investigating the hypotheses that bug-*MRs* involve few files whereas comment-*MRs* involve large number of files.

2.2. MSR via Data Mining

Data mining provides a variety of techniques with potential application to MSR. Association rule mining is one such technique. As an example, the recent work by Zimmerman et al [8] aims to identify co-occurring changes in a software system. For example, when a particular source-code entity (e.g., function with name *A*) is modified what other entities are also modified (e.g., functions with names *B* and *C*). This is akin to market-basket analysis in Data Mining. The presented tool, *ROSE*, parses the (C++, Java, Python) source code to map the line numbers to the syntactic or physical-level entities. These derived entities are represented as a triple (*filename, type, identifier*). The subsequent entity changes in the repository are grouped as a transaction. An association rule mining technique is employed to determine rules of the form $B, C \Rightarrow A$. Examples of deriving association rules such as a particular “*type*” definition change leads to changes in instances of variables of that “*type*” and coupling between interface and implementation is demonstrated. This technique is applied on eight open-source projects with a goal of utilizing earlier versions to predict the changes in the later versions. Although performed at a function and variable granularity, the best precision reported was 26% at the file-level granularity.

In summary, their technique brings forward various capabilities:

- Ability to identify addition, modification, and deletion of syntactic entities without utilizing any other external information (e.g., AST).
- Handles various programming languages and HTML documents.
- Detection of hidden dependencies that cannot be identified by source-code analysis.

2.3. MSR via Heuristics

CVS annotation analysis can be extended by applying heuristics that include information from source code or source-code models. A variety of heuristics, such as developer-based, history-based, call/use/define relation, and code-layout-based (file-based), are proposed and used by Hassan et al [5] to predict the entities that are candidates for a change on account of a given entity being changed. CVS annotations are lexically analyzed to derive the set of changed entities from the source-code repositories. The following assumptions were used: changes in one record are considered related; changes are symmetric; and the order of modification of entities in a change set is unimportant. The authors briefly state that they have developed techniques to map line-based changes to syntactic entities such as functions and variables, but it was not completely clear the extent to which this is automated.

These heuristics are applied to five open-source projects written in C. General maintenance records (e.g., copyright changes, pretty printing, etc) and records that add new entities are discarded. The best *average* precision and recall reported in table 3 of [5] was 12% (file-based) and 87% (history) respectively. The call/use/define heuristics gave a 2% and 42% value for precision and recall respectively while the hybrid heuristics did better.

The research in both [8] and [5] use source-code version history to identify and predict software changes. The questions that they answered are quite interesting with respect to testing and impact analysis.

2.4. MSR via Differencing

Source-code repositories contain differences between versions of source code. Therefore, MSR can be performed by analyzing the actual source-code differences. Such an approach is taken by the tool *Dex*, presented by Raghavan et al [7], for detecting syntactic and semantic changes from a version history of C code. All the changes in a patch are considered to be part of a single higher level change, e.g., bug-fix. Each version is converted to an abstract semantic graph (ASG) representation. A top-down or bottom-up heuristics-based differencing algorithm is applied to each pair of in-memory ASGs specialized with *Datrix* semantics. The differencing algorithm produces an edit script describing the nodes that are added, deleted, modified, or moved in order to achieve one ASG from another. The edit scripts produced for each pair of ASGs are analyzed to answer questions from entity-level changes such as how many functions and function calls are inserted, added or modified to specific changes such as how many *if* statement conditions are changed. *Dex* supports 398 such statistics.

This technique was applied to version histories of GCC and Apache. Only bug-fix patches were considered (deduced from the CVS annotations), 71 for GCC and 39 for Apache respectively.

The differencing algorithm takes polynomial time to the number of nodes. Average time of 60 seconds and 5 minutes per file were reported for Apache and GCC respectively on a 1.8 Ghz Pentium IV Xeon 1GB RAM machine. The six frequently occurring bug-fix changes as a percentage of patches in which they appear are reported. *Dex* reported 378 out of 398 statistics always correct with an average rate of 1.1 incorrect results per patch.

In an approach by Collard et al [1, 6] a syntactic-differencing approach called *meta-differencing* is introduced. The approach allows you to ask syntax-specific questions about differences. This is supported by encoding AST information directly into the source code via an XML format, namely srcML, and then using *diff* to compute the added, deleted, or modified syntactic elements. The types and prevalence of syntactic changes are then easily computed. The approach supports queries such as:

- Are new methods added to an existing class?
- Are there changes to pre-processor directives?
- Was the condition in an if-statement modified?

While no extensive MSR case study has been carried out using meta-differencing, it does support the functionality necessary to address a range of these problems. Additionally, the method is fairly efficient and usable with run times for translation similar to that of compiling and computation of the meta-difference is around five times that of *diff*.

3. A DEFINITION OF MSR

The investigations described in the previous section have a number of common characteristics. They all are working on version release histories (changes), all work at some level of change granularity (software entity), and most of them ask a very similar (MSR) question. We also see that the MSR process is to extract pertinent information from repositories, analyze this information, and derive conclusions within the context of software evolution. From these examples we further define MSR by identifying some fundamental representational issues and defining the terminology so we can contrast the different approaches. However, first we discuss the types of questions asked.

3.1. MSR Questions & Results

What types of questions can be answered by MSR? In the examples described in section 2 we see two basic classes of questions. The first is a type of market-basket question and the other deals with the prevalence, or lack of, a particular type of change. The market-basket¹ type question is formulated as: If *A* happens then what else happens on a regular basis? The answer to such a question is a set of rules or guidelines describing situations of trends or relationships. That is, if *A* happens then *B* and *C* happen *X* amount of the time.

This type of question often addresses finding hidden dependencies or relationships and could be very important for impact analysis. MSR identifies (or attempts to identify) the actual impact set after the fact (i.e., after an actual change). However, MSR oftentimes gives a “best-guess” for the change. The change may not be explicitly documented and as such must sometimes be inferred.

¹ The term market-basket analysis is widely used in describing data mining problems. The famous example about the analysis of grocery store data is that “people who bought diapers often times bought beer”.

This is an interesting trade-off and is reflected in the results described in Hassan et al [5] and Zimmerman et al [8].

The other type of question addressed in the examples discussed concerns the characteristics of common changes. The work by Raghavan et al [7] asks the question: What is the most common type of change in a bug-fix? This also has implications to impact analysis but not directly.

To even begin to answer these types of high-level questions we need to address the practical aspects of extracting facts and information from source-code repositories.

3.2. Underlying Representation

Repositories consist of text documents containing source code (e.g., *routine.h*, *routine.cpp*). The representation of differences between versions may also contain source code (e.g., output of *diff*). If the mining process uses the source code in its original document form than fact extractors are limited to using a light-weight approach, such as regular expressions as an API to the source code. The source code can also be represented in a data view, such as an AST (Abstract Syntax Tree). The AST view allows an API that is based on the abstract syntax of the source code.

The choice of representation is very important. Using a textual document view allows access to all parts of the document including comments, white space, and particular ordering information. Tools such as *diff* also work on text files. However, this textual view creates difficulty in determining the contents of a particular version. On the other hand, using an AST view of the source code does not easily allow access to white space, comments, etc.

The representation of the differences between source-code documents is an extension of the source-code representation. Textual representations can use tools such as *diff*. Regions of lines that are deleted or added are recorded, along with additional lines of text of the added lines. The ASG tree/graph-based representations of a program allow for changes to be represented as tree/graph changes and can include syntactic information easily.

As information is extracted for the purpose of mining it must be stored. Because of the large amounts of source code involved the extraction result is often chosen to produce as compact a result as possible. For example, if the purpose is to take a single measurement of each source-code document then only this single result is required.

The higher the abstraction of the extraction result the more specific the purpose of the extraction. This makes methods and tools for extracting results unusable for other, even closely related, applications.

Note that the desire to not store all of the original documents is partially based on the source-code representation chosen. An AST representation can be hundreds of times larger than the original document. There is too much to store in memory simultaneously, so an external representation format must be used.

These differences in representation and the level of syntactic information extracted often makes methods and tools for extracting results unusable for other, even closely related, MSR applications.

3.3. Definitions of Terms

With respect to MSR the basic concepts involve the level of granularity of what type of software entity is being investigated, the changes, and the underlying nature of a change. We present the definitions of these concepts in an attempt to form a terminology for what a change is and how it can be expressed within the context of MSR. If a need arises, these definitions will be refined to accommodate the future MSR approaches as they emerge.

Definition: An *entity*, e , is a physical, textual, or syntactic element in software. Example: file, line, function, class, comment, if-statement, white-space, etc.

Definition: A *change*, δ , is a modification, addition, or deletion to, or of, an entity. Additionally, this change defines a mapping from the original entity to the new entity as in $\delta(e) \rightarrow e'$, $\delta(\emptyset) \rightarrow e'$ is addition, and $\delta(e) \rightarrow \emptyset$ is deletion. A change describes which entities are changed and where the change occurs.

Definition: The *syntax* of a change is a concise and specific description of the syntactic changes to the entity. This description is based on the grammar of the language(s) of entities. We classify δ in the context of e as having some specific syntactic type (if-statement), change type (add, remove), location, etc. For example: a condition was added to an if-statement; a parameter was renamed; an assignment statement was added inside a loop; etc. The notation for deriving the syntax of a change is as follows: $\text{syntax}(e, \delta) = (d_1, d_2, \dots, d_n)$ where each d_i is some descriptor of the syntax.

Definition: The *semantics* of a change – is a high-level, yet concise, description of the change in the entity's semantics or feature set. This may be the result of multiple syntactic changes that is, $\Delta = \delta_1 \circ \delta_2 \circ \dots \circ \delta_n$. For example: a class interface change; a bug fix; a new feature was added to a GUI; etc. So we can now define notation for the semantics of a change as: $\text{semantics}(e, \Delta) = (d_1, d_2, \dots, d_n)$ where each d_i is some descriptor of the semantics.

4. INFORMATIONAL REQUIREMENTS

Mining of Software Repositories (MSR) is operationalized by the dimensions of the problem and types of information that must be extracted to support the high-level question. We feel the following are key dimensions to categorize MSR approaches:

- Entity type and granularity used (e.g., file, function, statement, etc.);
- How changes are expressed and defined (e.g., modification, Addition, Deletion, location, etc.);
- Type of question (e.g., market-basket, frequency of a type of change, etc.).

We have already addressed the type of questions in section 3.1. We now need to focus on the information necessary to answer these questions. From the discussion in section 3, we see that two types of pertinent information need to be extracted to answer MSR questions, namely entity-level information and information about the nature of change of an entity. We now describe each category and the specific types of fact extraction associated with each.

4.1. Entity-Level Information

The entity-level category addresses which entities changed, the location of the changed entities, and how many were changed. For example if functions represent our entities then we want to be able to answer queries such as:

- Which functions were added?
- Was the function A modified?
- How many functions were deleted?

The first query is a discovery or fact extraction activity regarding functions that were added between given source-code versions (e.g., a list of functions $[f_1, f_2, \dots, f_n]$). Similar questions can be defined for the deletion, modification, or movement of an entity. MSR approaches need this information for addressing questions such as identifying relations between functions that were added i.e., did a addition/deletion of a particular function lead to the addition/deletion of other functions?

The second query regards a particular function of interest. The research discussed in section 2.1 (CVS Annotations) needs this type of support to determine whether a particular class was modified in a given version.

The last query is an aggregate count that is useful for identification of higher level semantic changes such as those in the techniques discussed in section 2.4 (Differencing).

4.2. Change Information

Determining the nature of a change in an entity is the next step in the process. This kind of change can be syntactic or semantic. This specific change information can enhance the research presented in section 2.2 (Data Mining) by enabling the restricted application of the association rules and thus cutting down the list of affected entities that are reported. For example, consider a case where the change to an existing if-statement is only in the condition. The rule $\{if-condition\ change\} A \Rightarrow B, C$ would report B and C as affected entities only when the precondition $\{if-condition\ change\}$ in entity A is satisfied. Augmenting these rules with the exact nature of change further reduces the number of affected components and applicable association rules; thus avoiding false positives. Also, to determine a semantic change, such as identifying interface changes, this type of knowledge is needed:

- Are the modifications in function A only in if-statements?
- Was the conditional in the 2nd if-statement deleted in function A ?

The higher-level semantic information such as identification of conditional bugs addressed by research discussed in section 2.4 (Differencing) needs lower-level facts as reported by the above questions:

- Were only comments changed in function A ?
- Was the header comment of function A modified?
- Is there a change in the code layout in an entity A ?

These questions enable analysis to utilize or discard such textual changes. The research discussed in section 2.3 (Heuristics) analyzed CVS message annotations to discard header comment changes and proposed heuristics on predicating change propagation based on developer name and code layout. This

approach can be augmented with the facts gathered by the above questions.

Table 1. A taxonomy of MSR approaches.

	Entity	Change	Question
Annotation Analysis			
Gall et al	class	syntax and semantic - hidden dependencies	market basket and prevalence
German	file & comment	syntax and semantic – file coupling	market basket and prevalence
Heuristic			
Hassan et al	function & variable	syntax and semantic - dependencies	market basket
Data Mining			
Zimmerman et al	class & method	syntax and semantic - association rules	market basket
Differencing			
Raghavan et al	logical statement	syntax and semantic – move	prevalence
Collard et al	Logical statement	syntax – add, delete, modify	prevalence

5. CONCLUSIONS

In Table 1 we present an overview of the discussed approaches along with their MSR characteristics. We've categorized them generally into four groups (along the left). Then for each, we identify what granularity of entities they deal with, what types of changes they express (as defined in section 3.3), and what general class of question they are trying to address.

There is a large difference in the level to which these approaches understand the programming language syntax. Most of the approaches work with a fairly high-level entity. The two differencing approaches however can work as low as primitive logical programming language statements (if, while, class, or function).

Further investigation is necessary to discern between how changes are expressed. Also, there is very different semantic information

being used in the approaches. The notation we defined fits in well here but the domains must be further studied to support a more descriptive taxonomy. It is interesting to notice that both classes of questions are represented in this survey.

6. ACKNOWLEDGEMENTS

This work was supported in part by a grant from the National Science Foundation C-CR 02-04175.

7. REFERENCES

- [1] Collard, M. L. Meta-Differencing: An Infrastructure for Source Code Difference Analysis. Kent State University, Kent, Ohio USA, Ph.D. Dissertation Thesis, 2004.
- [2] Gall, H., Hajek, K., and Jazayeri, M. Detection of Logical Coupling Based on Product Release History in Proceedings of 14th IEEE International Conference on Software Maintenance (ICSM'98) (Bethesda, Maryland, March 16 - 19, 1998), 190-198.
- [3] Gall, H., Jazayeri, M., and Krajewski, J. CVS Release History Data for Detecting Logical Couplings in Proceedings of Sixth International Workshop on Principles of Software Evolution (IWPSE'03) (Helsinki, Finland, September 01 - 02, 2003), 13-23.
- [4] German, D. M. An Empirical Study of Fine-Grained Software Modifications in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 316-325.
- [5] Hassan, A. E. and Holt, R. C. Predicting Change Propagation in Software Systems in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 284-293.
- [6] Maletic, J. I. and Collard, M. L. Supporting Source Code Difference Analysis in Proceedings of IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11-17, 2004), 210-219.
- [7] Raghavan, S., Rohana, R., Podgurski, A., and Augustine, V. Dex: A Semantic-Graph Differencing Tool for Studying Changes in Large Code Bases in Proceedings of 20th IEEE International Conference on Software Maintenance (ICSM'04) (Chicago, Illinois, September 11 - 14, 2004), 188-197.
- [8] Zimmermann, T., Weißgerber, P., Diehl, S., and Zeller, A. Mining Version Histories to Guide Software Changes in Proceedings of 26th International Conference on Software Engineering (ICSE'04) (Edinburgh, Scotland, United Kingdom, May 23 - 28, 2004), 563-572.

A Framework for Describing and Understanding Mining Tools in Software Development

Daniel M. German

Davor Čubranić

Margaret-Anne D. Storey

Software Engineering Group, Dept. of Computer Science
University of Victoria, Box 3055 STN CSC, Victoria BC
Canada V8W 3P6

{dmg, mstorey, cubranic}@uvic.ca

ABSTRACT

We propose a framework for describing, comparing and understanding tools for the mining of software repositories. The fundamental premise of this framework is that mining should be done by considering the specific needs of the users and the tasks to be supported by the mined information. First, different types of users have distinct needs, and these needs should be taken into account by tool designers. Second, the data sources available, and mined, will determine if those needs can be satisfied. Our framework is based upon three main principles: the type of user, the objective of the user, and the mined information. This framework has the following purposes: to help tool designers in the understanding and comparison of different tools, to assist users in the assessment of a potential tool; and to identify new research areas. We use this framework to describe several mining tools and to suggest future research directions.

1. INTRODUCTION

Understanding how programs evolve or how they continue to change is a key requirement before undertaking any task in software engineering or software maintenance. Software engineering is a highly collaborative activity and hence *awareness* is an important factor in being informed of what has changed and what is currently being changed.

Software teams consist of many different stakeholders with distinct roles in their projects. A developer is interested in knowing how related artifacts changed in the past and why these changes occurred. A reengineer wants to consider how a system has evolved so that they can learn from prior experiences before redesigning the system. A manager is interested in understanding ongoing development and a programmer's previous work before assigning new work. A researcher wants to study how large projects have evolved so that the lessons learned can be applied to new projects. And a tester wishes to know which parts of the program to test, and who to talk to if they have questions or problems to report. Some of the many questions these various stakeholders ask of a software project can often be answered by other team members. In some

cases the relevant team members may no longer be available or they may not remember important details adequately. Therefore, answering these questions requires the extraction of information from a project's history to answer a particular stakeholder's questions. Unfortunately, these questions often do not have a simple answer. Details concerning concrete changes can be extracted from a source code repository, but the intent behind these changes is not easy to infer without considering other information sources and doing some sort of deeper analysis.

During the past few years, many researchers have started to investigate how software repositories and other information sources can be mined to help answer interesting questions which will inform software engineering projects and processes. Most of these research projects originate from trying to solve particular problems that satisfy different user needs.

In a recent paper [15], we presented a framework to describe how awareness tools in software development use visual techniques to present relevant information to different stakeholders. We used this framework to provide a survey of visualization tools that provide awareness of human activities in software engineering. The framework considered the intent behind these tools, their presentation and interaction style, the information they presented, as well as preliminary information on their effectiveness.

We noted in this earlier survey that the visualization tools are limited in their effectiveness by the information available to display. For example, if a tool only extracts information about software releases, the tool will not be able to reveal who made the changes, no matter how sophisticated the visualization technique may be. Extracting information from most information sources is relatively straightforward. But many questions can only be answered by correlating information from multiple sources. The difficulty of successfully mining pertinent information arises during this analysis. It is challenging to know which questions to ask and how best to answer the questions given that some of the information may be incomplete or vague. An example is relating an email message to a particular change in the source code when trying to discover intent. Another problem is that such information repositories although rich, are often very large and contain many details that are not relevant to the problem at hand. It is also important to know how to filter the information so that the user is not overwhelmed by a deluge of data.

The goal of this paper, therefore, is to complement our visualization framework by exploring and analyzing the issues related to the mining aspects of software tools. In our previous work we stud-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, Saint Louis, Missouri, USA Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

ied the issues related to the presentation of information to the user, while in this paper we focus on the data available and its extraction. A framework for mining software repositories should enable us and other researchers to understand how these diverse mining tools are positioned within a broader research context. It should provide a mechanism for tool researchers and designers to evaluate and compare their work with other efforts, as well as illuminate new research areas which could benefit software engineering.

In the first part of this paper, we summarize the different user roles and the specific tasks that can be supported by mining software repositories. We then explore, in depth, the different types of information that can be beneficial to these user roles while considering what kinds of analyses are needed to discover pertinent information. Finally, we demonstrate the benefits of this framework by comparing three diverse research tools that were independently developed by the three authors. Each of these three tools mines or extracts information from software repositories to support software engineering tasks. The framework helps us understand how these tools may be improved and highlights the need for more analysis of combined information sources.

2. A FRAMEWORK FOR COMPARISON

The framework for comparing software visualization tools of human activities is described in detail in [15]. Here we focus on just three of its dimensions, where each attempts to describe a different aspect of a repository mining tool. “Intent” explains who are the expected users of the tool, and its main objective. “Information” describes the specific sources that the tool mines and the type of analysis made by the tool. This dimension is elaborated in more detail as it is most relevant to mining software repositories. We provide some examples of tools to strengthen the descriptions of information extraction where necessary. Finally, the “infrastructure” addresses any special needs that the tool has.

2.1 Intent

We describe this dimension in detail in our other paper [15], but summarize it here.

Role. This dimension identifies who will use the tool. Roles include *developers*, and whether they are a part of a *team* that is co-located or distributed. Other development roles include *maintainers*, *reverse engineers* and *reengineers*. *Managers*, *testers* and *documenters* can also improve their effectiveness by knowing about human activities in the project. And finally *researchers* may wish to explore human activities to make recommendations for improved tools and processes on future projects.

Time. Some tools provide information about activities occurring in the distant or near *past*, while other tools focus on presenting information about the *present*. Other tools try to forecast the *future* and predict which parts of the system are more prone to be modified in the future.

Cognitive support. Cognitive support describes how a tool can help improve human cognition [16]. In order to provide cognitive support, it is essential to know which tasks require extra tool support. Specifically we need to know which questions are likely to be asked during these tasks and how the questions can be answered. The questions that the various roles can ask about developer activities can be roughly classified into four categories: *authorship*, *rationale*, *chronology*, and *artifacts*. Consequently, we consider

how mining tools can provide information according to these four categories.

2.2 Information

As we mentioned previously, this dimension is thoroughly explored as it is the most relevant to mining software repositories. To help clarify the discussion when necessary, we give specific examples of tools.

Change management. *Configuration management* tools provide support for building systems by selecting specific versions of software artifacts [7]. *Version control* tools contribute to software projects in the following ways: software artifact management, change management and team work support [18]. Change management is an important data source because it provides *traceability*: it records who performed a given change, and when it was performed. The capabilities of the change management system will determine the type of information that can be extracted. For instance, CVS does not record when a given commit is a branch-merge and it does not support transactional commits. Several heuristics have been created to overcome these problems [4, 5].

Program code. We classify these tools into two categories. In the first category we place those tools that treat the file as a unit, and make no effort to understand its contents; we call these tools **programming-language-agnostic**. On the other hand, tools are **programming-language-aware** if they attempt to do some fact extraction from the source code. We can further classify programming-language-aware tools based upon:

- **The language supported.** Given the differences in syntax and grammar, tools that are language-specific can only understand a fixed set of programming languages.
- **Syntactic analysis.** In this type of analysis the extractor does not need to understand what the code does, only its syntax. Examples of this analysis are the removal of comments from the source code (to be able to distinguish if the changes affected actual source code or only its documentation), and extraction of the main entities of the code (such as packages, classes, methods, functions, etc.).
- **Semantic analysis.** This analysis requires an understanding of the intent of the source code and can be done *dynamically* (by running the software under well defined test-cases) or *statically* (by processing the source code). The generation of a call graph, or the tracing of the execution of a program are examples of this type of analysis.

Defect tracking. Many larger software projects rely on tracking tools to help with the management of *defects* and *change requests*. Such systems often store metadata about who is assigned a task and track the task’s completion. In some cases a defect management tool is also used as a way to track activities and changes in requirements. For example, Bugzilla includes a category for a defect report called “improvement”, which is used by its users to submit a change in requirements.

Correlated information. We have observed that the type of analysis and correlation can be classified into two broad categories:

- **Within the data source.** This type of analysis uses data from one data source only and attempts to correlate different data

entities within it. In some cases this analysis strives to reconstruct relationships that were lost because they were not explicitly recorded (such as grouping file revisions into commits in CVS). In other cases, the new information is computed from the data available in the source (for example, extracting the functions that were modified in a given change). Some sources are very rich in the amount of information that can be extracted and correlated from them (version control systems are one example).

- **Between the data sources.** In some cases, there is explicit information that allows a tool to correlate entities from two different data sources. For example, it is not uncommon for open source developers to record the corresponding Bugzilla defect number in the log of the CVS commit that resolves such defect, allowing a tool to correlate file revisions with a defect. Frequently, there is no explicit information that correlates information from different sources, and heuristics are required to build these relationships. For example, which email messages are relevant to a particular bug fix?

Informal communication. Email is undoubtedly the most widely used form of computer-mediated communication, and it is not surprising that distributed software development projects rely on it extensively. In the early days of open-source software, a project mailing list used to be one of the first, and often the only, communication and coordination mechanism used by development teams [2]. Today, specialized tools like Bugzilla have taken over some of its functionality, but email remains an essential component of distributed development process. For example, open-source projects typically document all decisions on the project mailing list, even when the original decision was reached in a different medium, or such as face-to-face [8].

In recognition of the mailing list's importance to a project, it is usually archived and available on the web. However, messages in the archive are typically organized chronologically or at best by conversation thread. Even when text search of an archive is available, finding specific information can be difficult. For example, if a developer wants to know why certain a function was added to the project, then the challenge is to find all the messages that relate to the decision to add that function. The limited structure and metadata of archived email mean that this source of information is rarely mined. However, in their study of developer communication in open-source projects, Gutwin et al. found that developers would like to see improved access to email archives [8].

Various forms of text chat, such as IRC and IM, have become increasingly important channels of communication in open-source projects. For example, in 2000, neither Apache nor Mozilla projects had official IRC channels used by the development team, and today both do. Text chat is rarely archived (and when it is, it is usually in another form, such as email messages, or as part of a Web page), but this is likely to change as its importance is recognized. However, chat has even less structure than email, so it may be considerably more difficult to mine effectively.

Advances in computing technology are making it possible to archive communication that used to be unarchivable. For example, Richter et al. have demonstrated a system for automated capture of team meetings [11]. Their system provides automated transcript of the spoken content, which the attendees can annotate on-the-fly with a set of keywords from a predefined list.

Local history. Many local interactions are not captured by a project's repositories. However, a developer's local history is a rich resource for understanding human activities and how they relate to the software under development. Recently, several researchers have been investigating how mining this information source can assist in navigation and program comprehension.

Two tools that address navigation support are Mylar and NavTracks. Mylar [9] provides a degree-of-interest model for the Eclipse software development environment. As a program artifact is selected, its value increases while the value of other artifacts decrease. Therefore, elements of more recent interest have a higher degree of interest value. Mylar filters artifacts from the Package Explorer in Eclipse that are below a certain threshold and thus helps a developer focus on the artifacts in the workspace that are relevant to the current activity. Navtracks [14] is a tool to support browsing through software spaces. It provides recommendations of files that should be of higher relevance to the user given the currently selected file. It keeps track of the navigation history of a software developer, forming associations between related files. Associations are created when short cycles are detected between file navigation steps. There are also several projects in the human interaction research community that investigate how tracking interaction histories can support future interactions [17, 1].

Schneider et al. describe how local interaction histories can be mined to support team awareness [13]. They propose that sharing local interactions among team members can benefit the following activities: coordinating team member activities such as undo, identifying refactoring patterns and coordinating refactoring operations, mining browsing patterns to identify expertise, and project management. They describe a tool called Project Watcher and are currently evaluating the benefits it brings to developers.

2.3 Infrastructure

This category addresses the environment needed to support the tool.

Required infrastructure. This category lists any requirement the tools have, such as a given operating system, an IDE such as Eclipse, a Web server and client, a database management system, etc.

Offline/Online. Tools can be classified depending upon whether the software repository is required during its operation. For instance, some tools mine the software repository ahead of time, while others query the repository as a result of a user request.

Storage backend. If the tool operates offline, this category is used to describe how it stores its required data. For example, some tools use a SQL backend, other use XML or a proprietary format.

3. A COMPARISON OF MINING

We now use this framework to help us understand the intent and mining capabilities of three tools designed by the authors.

3.1 softChange

Intent: The main goal of softChange is to help programmers, their managers and software evolution researchers in understanding how a software product has evolved since its conception [6]. With respect to *time*, softChange concentrates only on the past. In terms of *cognitive support*, it allows one to query who made a given change to a software project (*authorship*), when (*chronology*) and, whenever available, the reason for the change (*rationale*). The *artifacts*

that softChange tracks are files, and some types of entities in the source code (such as functions, classes, and methods).

Information: softChange extracts and correlates three main sources of information: the version control system (CVS), the defect tracking system (Bugzilla), and the software releases. softChange reconstructs some of the information that is never recorded by CVS (such as recreating commits), and it does syntactic analysis of the source code. The analysis is static and it supports C/C++ and Java. softChange also attempts to correlate information between CVS and Bugzilla using defect numbers.

Infrastructure: softChange is an offline tool that uses an SQL database for its storage needs. Its mining is done without any special requirements beyond access to the software repository. One particular problem with the type of mining that softChange does is that it can retrieve a very large amount of data, and for that reason, it is recommended that it operate on a local copy of the repositories (rather than query the repositories using the Internet, consuming their bandwidth and computer resources). softChange has two different front ends: one is Web based, and the other a Java application.

3.2 Hipikat

Intent: Hipikat can be viewed as a recommender system for software developers that draws its recommendations from a project's development history [3]. The tool is in particular intended to help newcomers to a software project. Therefore, in terms of the *time dimension*, it is concentrated on the past. *Cognitive support* is largely limited to answering questions about *rationale* and *artifacts*. In terms of user *roles*, Hipikat is targeted almost exclusively at developers and maintainers.

Information: Hipikat is designed to draw on as many information sources as possible and identify relationships between documents both of same and different types. The information sources that are currently supported in Hipikat are: version control system (CVS), issue tracking system (Bugzilla), newsgroups and archives of mailing lists, and the project Web site. All four of these sources are typically present in large open-source software projects.

Hipikat is programming language-agnostic. The only information that it collects from files in the version control system is versioning data, such as author, time of creation, and check-in comment.

Hipikat correlates information across sources using a set of heuristics, such as matching for bug id in version check-in comment to link file revisions in CVS and bug reports in Bugzilla. These heuristics are based on observations of development practices in open-source projects like Mozilla. Another method that Hipikat uses to find documents that are related is by textual similarity.

Infrastructure: Repository mining in Hipikat works in offline mode: Hipikat periodically checks project repositories for recent changes and updates its model. The model is stored in an SQL database. The front end is an Eclipse plug-in, although in principle it could be implemented for other environments, as long as it follows the communication protocol with the Hipikat server.

3.3 Xia/Creole

Intent: The main goal of the Xia [18] tool is to help *developers* understand version control activities by visualizing architectural dif-

ferences between two versions. Therefore, within the *time dimension*, it focuses on the past, both near and distant. Xia provides *cognitive support* for developers when they need answers to questions concerning *authorship*, *chronology*, and *artifacts*. Several of the visual techniques from Xia have been subsequently integrated into the Creole visualization plug-in for Eclipse [10]. The purpose of the Creole tool is to provide both high-level visualizations of the architecture of a program as well as detailed views of dependencies between software artifacts. Combining views from Xia with Creole means that information concerning version control activities can be viewed in concert with the dependency views in Creole.

Information: Creole represents software using a graph where nodes in the graph correspond to software artifacts such as packages, classes, methods, fields, etc., and edges correspond to relationships such as “created by,” “calls,” and “accesses data”. Creole extracts information from the CVS version control system and tags nodes in the graph with the following information: authorship (author of first commit, last commit, and the author with the most number of commits); time (time of first commit and most recent commit) and the total number of commits. This information can then be used in tooltips for the artifacts in the repository, or to filter nodes from the view or to highlight them using a colour scale.

Infrastructure: Creole and Xia both work in *online* mode and directly access the CVS repositories. Creole extracts dependencies from the source code using the Feat data extractor [12]. For large projects, CVS queries can be very slow. Creole and Xia have both been integrated with Eclipse as plugins. Creole is available for download from www.thechiselgroup.org/creole.

4. DISCUSSION AND CONCLUSIONS

Tools need to be created around the needs of the developer. To our knowledge, very little has been done in terms of asking developers what types of requirements they have, and few tools have been formally evaluated to determine if they are useful to their expected users. Many of the tools are created around the needs of the researcher (somebody who is interested in understanding how a software system has evolved). This is a natural phenomenon because many of these tools are built by researchers to satisfy their own requirements. We could argue that by coincidence some of the requirements of the managers are the same as those of the researcher. Developers, however, have a different type of questions that need answers. Researchers and managers are frequently satisfied with trends and aggregated data; the developer, on the other hand, requires precise answers most of the time. Once the needs of the potential users are better understood (the **intent**), then one can determine what information should be mined and how it can be analyzed (the **information**).

Some data sources are very rich, and others have been barely exploited. The more data retrieved, the more difficult it will be to find relevant information for a given query (high recall) with little noise (high precision). One can argue that the act of “mining” is not the important problem that tools are trying to solve. Instead, these tools are attempting to answer valid questions that their users have by taking advantage of the historical information available. Tapping into new sources of data should be done with relevance in mind. How can this data be used to help answer a question? Who is the potential user? What questions can it help answer?

The less structured or organized the historical information is, the

more difficult it is to use it effectively. We conjecture that the reason why few tool use email messages (and other informal forms of communication) is because they are difficult to correlate to other types of information, and to answer questions posed by the user. However, the informal forms of communication are being recorded, and in the future, they could prove to be an important source of valuable information.

We hope that this paper prompts discussion towards a common nomenclature, and potentially, an ontology that can be used to describe tools that mine software repositories. Another area that we believe should be considered is the selection of a set of applications that can serve as test cases or benchmarks (this has been already suggested during the previous Workshop in Mining Software Repositories in 2004). It would then be possible to create a corpus with copies of the software repositories, that can be shared among the researchers; this will reduce the stress posed to the servers of the projects that are to be mined.

Having a common set of benchmarks will also help to address another problem in the area of mining software repositories. The actual task of retrieving “facts” from the repository is not considered to be an important research issue. Often, the act of mining involves reverse engineering of the formats in which the data is stored, scraping information from the Web, or trying to find some regularity in the output of tools that access the repositories. In this case (such as the syntactic and semantic analysis of source code) it involves the use of tools created by other communities (such as the program analysis and comprehension communities); sometimes the problem is getting the tools to work with the information retrieved from a particular repository. The act of mining for facts is tedious and error-prone. If the community agrees on a set of test cases, the fact extraction can be done only once, and the resulting data shared along with the copies of the repositories. This will allow researchers more time to concentrate on the more important problems related to the analysis and, correlation of this information always keeping in mind the needs of the potential user.

5. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their helpful comments.

6. REFERENCES

- [1] M. Chalmers, K. Rodden, and D. Brodbeck. The order of things: Activity-centred information access. In *Proceedings of 7th Intl. Conf. on the World Wide Web (WWW7)*, 1998.
- [2] D. Čubranić and K. S. Booth. Coordinating open-source software development. In *Eighth IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, pages 61–65, 1999.
- [3] D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: A case study for software development. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work*, pages 82–91, 2004.
- [4] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance*, pages 23–32. IEEE Computer Society Press, September 2003.
- [5] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories*, 2004.
- [6] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softChange. In *Proc. of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE 2004)*, pages 336–341, 2004.
- [7] J. C. Grundy. Software architecture modeling, analysis and implementation with SoftArch. In *the Proceedings of the 25th Hawaii International Conference on System Sciences*, page 9051, 2001.
- [8] C. Gutwin, R. Penner, and K. Schneider. Group awareness in distributed software development. In *Proc. of the 2004 ACM conference on Computer supported cooperative work*, pages 72–81, 2004.
- [9] M. Kersten and G. Murphy. Mylar: A degree-of-interest model for IDEs. In *Proceedings of Aspect Oriented Software Development*, March 2005.
- [10] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu. Plugging-in visualization: experiences integrating a visualization tool with Eclipse. In *Proc. of the 2003 ACM Symposium on Software Visualization*, pages 47–56, 2003.
- [11] H. Richter, G. D. Abowd, C. Miller, and H. Funk. Tagging knowledge acquisition to facilitate knowledge traceability. *International Journal on Software Engineering and Knowledge Engineering*, 14(1):3–19, Feb. 2004.
- [12] M. Robillard and G. Murphy. Feat: A tool for locating, describing, and analyzing concerns in source code. In *Proceedings of 25th International Conference on Software Engineering*, May 2003.
- [13] K. Schneider, C. Gutwin, R. Penner, and D. Paquette. Mining a software developer’s local interaction history. In *Proceedings of 1st International Workshop on Mining Software Repositories*, 2004.
- [14] J. Singer, R. Elves, and M.-A. Storey. Navtracks: Supporting navigation in software space. In *International Workshop on Program Comprehension*, 2005. To be presented.
- [15] M.-A. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: a survey and a framework. In *Proceedings of the 2nd ACM Symposium on Software Visualization*, 2005. To be presented.
- [16] A. Walenstein. Observing and measuring cognitive support: Steps toward systematic tool evaluation and engineering. In *Proc. of the 11th International Workshop on Program Comprehension (IWPC’03)*, pages 185–195, 2003.
- [17] A. Wexelblat. Communities through time: Using history for social navigation. In T. Ishida, editor, *Lecture Notes in Computer Science*, volume 1519, pages 281–298. Springer Verlag, 1998.
- [18] X. Wu, A. Murray, M.-A. Storey, and R. Lintern. A reverse engineering approach to support software maintenance: Version control knowledge extraction. In *Proc. 11th Working Conference on Reverse Engineering*, pages 90–99, 2004.

SCQL: A formal model and a query language for source control repositories

Abram Hindle
Software Engineering Group
Department of Computer Science
University of Victoria
abez@uvic.ca

Daniel M. German
Software Engineering Group
Department of Computer Science
University of Victoria
dmg@uvic.ca

ABSTRACT

Source Control Repositories are used in most software projects to store revisions to source code files. These repositories operate at the file level and support multiple users. A generalized formal model of source control repositories is described herein. The model is a graph in which the different entities stored in the repository become vertices and their relationships become edges. We then define SCQL, a first order, and temporal logic based query language for source control repositories. We demonstrate how SCQL can be used to specify some questions and then evaluate them using the source control repositories of five different large software projects.

Categories and Subject Descriptors

D.2.7 [Software Engineering]: Distribution, Maintenance, and Enhancement—*Version Control*; D.2.8 [Software Engineering]: Metrics—*Process metrics*

1. INTRODUCTION

A configuration management system, and more specifically, a source control system (SCS) keeps track of the modification history of a software project. A SCS keeps a record of who modifies what part of the system, when and what the change was.

Typically a tool that wants to use this historical information starts by doing some type of fact extraction. These facts are processed in order to create new *information* such as metrics [9, 3] or predictors of future events [6, 7]. In some cases, this information is queried or visualized [5, 10]. Some projects store the extracted facts into a relational database ([8, 5, 2]), and then use SQL queries to analyze the data. Others prefer to use plain text files, and create small programs to answer specific questions [9], or query the SCS repository every time [10]. One of the main disadvantages of these approaches is that querying this history becomes difficult. A query has

to be translated from the domain of the SCS history to the data model or schema used to store this information. Also, questions regarding the temporal aspects of the data are difficult to express. Furthermore, there is no standard for the storage or the querying of the data, making it difficult for a project to share its data or its analysis methods with another one.

When a developer completes a task it usually means that she has modified one or more files. The developer then submits these changes to the SCS, in what we call a *modification request*, or MR (this process has also been called a transaction). A MR is, therefore, atomic (conceptually the MR is atomic, even though it might not be implemented as such by the SCS system). Once the change is accepted by the SCS, it creates a new *revision* for each file present in the MR. Thus an MR is a set of one or more file revisions, committed by one developer. The SCS allows its users to retrieve any given revision of a file, or for a given date, determine what is the latest revision for every file under its control.

There are many SCSs available on the market. They can be divided into two types: centralized repositories (like CVS) and Peer-to-Peer repositories (such as BitKeeper, Darcs, Arch). Even though they differ strongly in the way they operate and store the tracked changes, they all track files and their revisions. We will focus on CVS because there is a large number of CVS repositories available to researchers. We will, therefore, use the CVS nomenclature in this paper. It is important to mention that our model and SCQL can be applied to any SCS.

This paper is divided as follow: first we present an abstract model to describe version control systems; second, we define a query language, called SCQL, we end demonstrating how it can be used to pose questions related to the source control history in several mature, large projects.

2. MODEL

In order to create a language for the querying of a SCS we first need to be able to describe its data model. This data model will be used to formally describe the data available in the SCS and to provide a uniform representation of the information available across multiple SCSs. One of the requirements of this model that is “time aware” and it is able to represent the temporal relationships (“before”, “after”) of the different entities stored in the SCS.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR’05, May 17, 2005, Saint Louis, Missouri, USA
Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00

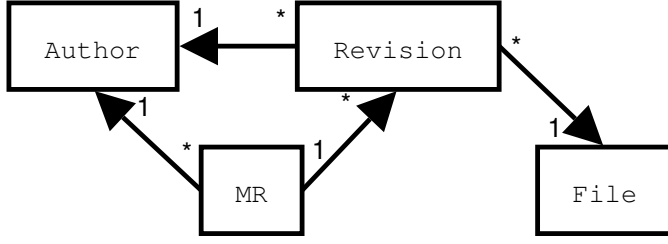


Figure 1: Cardinality and Directions of Edges in the Model

2.1 Characteristic Graph of a Source Code Repository

We represent an instance of a SCS as a directed graph. Entities such as MRs, Revisions, Files and Authors are vertices, while their relationships are represented by edges. It is important that for any given instance of a SCS, there exists a corresponding characteristic graph, and that given a query, this query can be translated into an equivalent graph query on its characteristic graph. As a consequence, the original query will be answered by solving the graph query.

2.2 Entities

The model for SCQL contains four different types of entities: MRs, Revisions, Files and Authors. See figure 1.

MRs model modification requests and correspond to the set \mathbb{MR} in the graph instance. MRs have attributes such as log comments, timestamp, and a unique ID. We assume that the timestamp of an MR is unique (derived from its earliest revision), and that an MR is an atomic operation. There exists an edge from each MR to the next MR in time (if one exists). One edge extends from the MR to the author of its revisions, and one edge is also created from the MR to each of its revisions (an MR is not connected to more than one revision of the same file).

Revisions correspond to the set of file revisions and are denoted by $\mathbb{Revision}$. Revisions are atomic in time with respect to other revisions, thus they have unique timestamps and they are assigned unique identifiers. They have attributes such as the *diff* of the change, and the lines added and removed. An edge extends from the revision to its author, and another one to the corresponding file. Revisions are also connected to each other. An edge is created from any given revision to each of its successor (the revision which modified it), thus one revision can have multiple children (or branches). Revision subgraphs are characterized as acyclic stream-like graph which springs up from a single node. If a revision merges a branch from another branch (or the main development trunk), an edge will be created from the “predecessor” revisions on both branch to the merged revision.

Files are represented as the subset of vertices \mathbb{File} in the graph. Files are the springs from which streams of revisions flow. Files have attributes such as path, filename, directory, and a unique full path name. Time-wise, files have unique timestamps associated with the first revision made of a file (this records the moment the file first appears in the graph). Files are connected to by revisions as described above.

Table 1: Model Primitives

$isaMR(\phi)$	is ϕ an MR?
$isaRevision(\phi)$	is ϕ a Revision?
$isaFile(\phi)$	is ϕ a File?
$isaAuthor(\phi)$	is ϕ an Author?
$numberToStr(i)$	Represent i as a string.
$length(\phi)$	Length of the string ϕ
$substr\phi, k, l$	Return a substring of ϕ of length l at k .
$eq(\phi, \theta)$	are ϕ and θ equivalent strings?
$match(\phi, \theta)$	is θ a substring of ϕ ?
$isEdge(\phi, \theta)$	is there an edge from ϕ to θ ?
$count(S)$	counts the elements in a subset.
$isAuthorOf(\psi, \phi)$	is ψ an author of ϕ ?
$isFileOf(\tau, \phi)$	is τ an File of ϕ ?
$ifMROf(\phi, \phi)$	is ϕ is an mr of ϕ ?
$isRevisionOf(\theta, \phi)$	is θ is a revision of ϕ ?
$revBefore(\theta, \theta_2)$	is there is a revision path from θ to θ_2 ?
$revAfter(\theta, \theta_2)$	is there is a revision path from θ_2 to θ ?

Authors are represented by the subset \mathbb{Author} in the graph. Authors have attributes such as user ID, name and email. Time wise authors are associated to their first revision implying their entry into the project. There is only one author per MR and per Revision.

2.3 Formalizing the characteristic graph

Formally we define the characteristic graph G of a SCS as a directed graph of $G = (V, E)$ where

$$V = \mathbb{MR} \cup \mathbb{File} \cup \mathbb{Author} \cup \mathbb{Revision}$$

$$\begin{aligned}
E = & (v_1 \in \mathbb{MR}, v_2 \in \mathbb{MR}) \cup (v_1 \in \mathbb{MR}, v_2 \in \mathbb{Revision}) \\
& \cup (v_1 \in \mathbb{MR}, v_2 \in \mathbb{Author}) \cup (v_1 \in \mathbb{Revision}, v_2 \in \mathbb{Revision}) \\
& \cup (v_1 \in \mathbb{Revision}, v_2 \in \mathbb{Author}) \cup (v_1 \in \mathbb{Revision}, v_2 \in \mathbb{File})
\end{aligned}$$

There are 6 data types in our model: Vertices representing entities; edges representing relationships; sets of entities which abstract edges; numbers used for numerical questions; strings are needed since much of the data in the repository is string data; and Booleans which are necessary to prove invariants exist. Table 1 provides a description of some of the primitives that operate on these types.

We implement attributes using maps. Attributes can map from entities to subsets, strings, numerics or Booleans. Another assumption is that the output of a mapping is only valid if a node or edge of a correct type is used as an index to the map. More attributes can be added at any time but the attributed mentioned in section 2.2 are the expected attributes. Attributes which are expected to return one entity still return a subset. The motivation is to maintain uniform access to entities while providing a method of abstracting edge traversal. Since sets are returned we use plural function names. Attributes that are subsets of entities (edge traversals) are described in table 2.

2.4 Extraction and Creation

The general algorithm for extracting and creating a graph from a SCS is:

Table 2: Sub-domain Attributes

$authors(\phi \in \mathbb{MR})$	the author of the MR
$revisions(\phi \in \mathbb{MR})$	the revisions of the MR
$files(\phi \in \mathbb{MR})$	the files of the revisions of the MR
$nextMRs(\phi \in \mathbb{MR})$	next MR in time
$prevMRs(\phi \in \mathbb{MR})$	previous MR in time
$mrs(\theta \in \mathbb{Revision})$	MR related of the Revision
$authors(\theta \in \mathbb{Revision})$	the author of the revision.
$files(\theta \in \mathbb{Revision})$	the files of a the revision
$nextRevs(\theta \in \mathbb{Revision})$	Next revisions version-wise.
$prevRevs(\theta \in \mathbb{Revision})$	Previous revisions version-wise.
$mrs(\tau \in \mathbb{File})$	MRs of the Revisions of the file
$revisions(\tau \in \mathbb{File})$	Revisions of the file
$authors(\tau \in \mathbb{File})$	Authors of the revisions of the file.
$mrs(\psi \in \mathbb{Author})$	MRs of the author.
$revisions(\psi \in \mathbb{Author})$	Revisions of the author
$files(\psi \in \mathbb{Author})$	Files of the revisions of the author

- Each file becomes a vertex in \mathbb{File} .
- Each author becomes a vertex in \mathbb{Author} .
- Each revision becomes a vertex in $\mathbb{Revision}$. Assign revisions unique timestamps and connect each revision its corresponding author and file.
- Create vertices for each MR. The MR inherits the timestamp from its first file revision. Associate MR to its author MR.
- Each MR is then connected to the next MR (according to their timestamp), if it exists.
- For each file, connect each revision to the next revision of the file, version-wise. If branching is taken into account, only revisions in the same branch are connected in this manner, and then branching and merging points are connected.

When this algorithm terminates, the result is a characteristic graph of the instance of SCS.

CVS does not record branch merges or modification requests, but some heuristics have been developed to recover both [2, 4, 11]. Branch-merge and MR recovery in CVS are not accurate, and therefore the extracted SCS graph is an interpretation rather than an exact representation of the SCS.

An example of the SCS graph is depicted in figures 2 and 3. The vertices corresponding to the revisions in 2 and 3 are the same and they are shown in two figures to avoid clutter.

3. QUERY LANGUAGE

The rationale for our model is to provide a basis for a query language for a SCS. We are interested in a language that has the following properties:

- It is based on primitives that correspond to the actual data and relationships stored in a SCS. We want a language that directly models files, authors, revisions, etc.

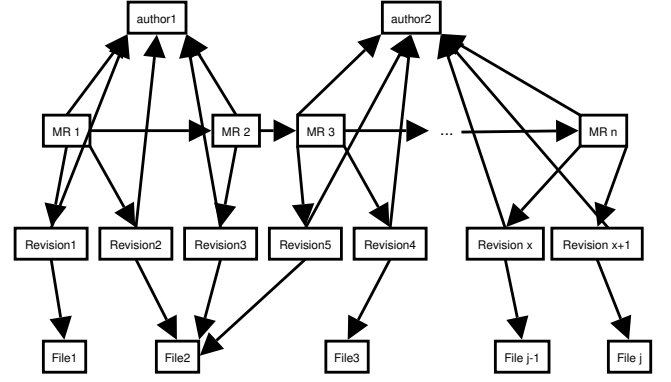


Figure 2: Example Model Subgraph

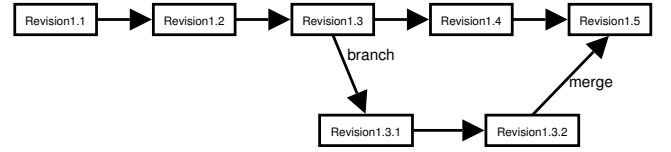


Figure 3: Example Revision Subgraph

- It has the ability to take advantage of the time dimension. We want to be able to pose questions that include predicates such as “previous”, “after”, “last time”, “always”, “never”. For example, “has this file *always* been modified by this author?”, “find all MRs do not include the following file”, “find the file revision *after* this other one”, “find the *last* revisions by a given author”, etc.
- It is computable. We need confidence that if a query is posed, it can be evaluated.
- It is expressive. We are interested in a language that is able to express a wide range of queries.

The characteristic graph of a source code repository is the basis for this language. Thus our language is built such that any query expressed in it can be translated to a query of the characteristic graph.

First order predicate logic will serve as a basis for our query language, as it can handle both graph semantics and “before and after” aspects of temporal logic [1]. The language is designed to query the model, not to provide a general purpose programming language. We have focused in evaluating decision queries with this language (those which answer is either yes or not), but we also support other types of queries that return other types of data (such as the id of an author, the number of files modified, or a set of files).

The language has a rich syntax, but due to a lack of space we only summarize its main features in table 3.

Identifiers are unbound variables that reference entities. Using a variable, one can access the attributes of the referenced

entities ($x.attribute$). Identifiers are only created by a scoping operator such as an Anchor, Universal Quantifiers, Existential Quantifiers or Selection Scope. These scopes iterate over elements in a subset by applying a predicate to each element.

Existential and Universal scopes iterate through an entire subset until a preposition returns either true or false. For empty subsets universal scopes return true and existential scopes return false.

Subset/Select based scopes effectively iterate through all the elements in set of entities such as `MR`, `Revision`, `File`, `Author` selecting entities to form a subset. A subset can only be the same size or smaller than the set it is testing. These subsets may only have 1 type of entity. Anchor scopes are like select- based scopes, but are meant to access a single element in constant time. Scope operators that are “before” or “after” scopes iterate through their respective subsets in sequential order from first to last.

3.1 Example Queries

We now present three different queries and show how they are expressed in SCQL.

Example 1: Is there an author a who only modifies files that author b has already modified? This query can be formally expressed as:

$$\begin{aligned} &\exists a, b \in \text{Author} \text{ s.t. } a \neq b \wedge \\ &\forall r \in \text{Revision} \text{ s.t. } \text{isAuthor}(a, r) \implies \\ &\quad \exists r_b \in \text{Revision} \text{ s.t. } \text{before}(r_b, r) \wedge \\ &\quad \text{isAuthor}(b, r_b) \wedge r.file = r_b.file \end{aligned}$$

We are trying to find two different authors such that for all revisions of one author, there exists a previous revision (by the second author) to the same file. The SCQL query first finds two authors and makes sure they are different. Then it iterates through all the revisions of author a . Per each revision, it checks if the file of that revision has another previous revision that belongs to author b . `a.revisions` gets all the revisions related to the author a while `isAuthorOf(b, r2)` tests if b is the author of the revision of the file f .

```
E(a, Author) {
  E(b, Author) {
    a!=b &&
    A(r, a.revisions) {
      A(f, r.file) {
        Ebefore( r2, f.revisions, r) {
          isAuthorOf( b, r2)
        }
      }
    }
  }
}
```

Example 2: Compute the proportion of MRs that have a unique set of files which have never appeared as part of another MR before. With this query we are want to find out how variable are the sets of files modified in MRs. We

hypothesize that an old, stable project will have a small proportion, while a project that is still growing, and continues to have structural changes will have a larger proportion. This query can be easily expressed directly in SCQL as:

```
1 - (Count(mr,MR) {
  Ebefore(a,MR,mr)) {
    A(f,mr.files) {
      isFileOf(f,a)
    }
  }
} / count(MR)
```

It iterates over the set of all MRs, counting only those that have a previous MR that modifies all its files too. Then it counts all MRs, and computes the desired proportion.

Example 3: Is there an Author whose changes stay within one directory?

$$\begin{aligned} &\exists a \in \text{Author} \text{ s.t.} \\ &\forall f \in \text{File} \text{ s.t. } \text{isAuthorOf}(a, f) \implies \\ &\forall f_2 \in \text{Files} \text{ s.t. } \text{isAuthorOf}(a, f_2) \implies \\ &\quad \text{directory}(f) = \text{directory}(f_2) \end{aligned}$$

In this case we want to know if there exists an author such that for all pairs of files modified by this author, they are both in the same directory. This query can be written in SCQL as:

```
E(a, Author) {
  A(f, author.files) {
    A(f2, author.files) {
      eq(f.directory, f2.directory)
    }
  }
}
```

4. EVALUATION

We have built an implementation for SCQL. In order to demonstrate the effectiveness of SCQL we ran the 3 example queries against five different projects: Evolution (an Email Application), Gnumeric (a spreadsheet), OpenSSL (A Secure Socket Layer library), Samba (Linux support for Win32 network file systems), and modperl (a module for Apache that acts like a Perl Application server). The table 4 provides the output of the 3 example queries for each of these projects. We include the size of the `MR` set (number of MRs) and the `File` set too.

Table 4: Evaluation of the 3 example queries

	evolution	gnumeric	openssl	samba	modperl
Ex 1	true	true	false	false	true
Ex 2	0.002	0.004	0.003	0.002	0.015
Ex 3	false	false	false	false	true
File	4748	3685	3698	4246	300
MR	18573	11337	10847	27413	1398

Table 3: Language Description

Name	Language	Explanation
\mathbb{MR}	MR	Set of Modification Requests
Revision	Revision	Set of Revisions
Author	Author	Set of Authors
File	File	Set of Files
Universal	$A(\phi, \delta)\{P(\phi)\}$	For all ϕ in the set δ is the predicate $P(\phi)$ true?
Existential	$E(\phi, \delta)\{P(\phi)\}$	Does ϕ exist in set δ where predicate $P(\phi)$ is true?
Attribute	$\phi.\zeta$	Given an entity ϕ return its attribute ζ
Function	$\gamma(P)$	Evaluate the function γ with P as the parameter
Universal Before	$Abefore(\phi, \delta, \theta)\{P(\phi, \theta)\}$	For all ϕ in δ before θ is the binary predicate $P(\phi, \theta)$ true?
Universal After	$Aafter(\phi, \delta, \theta)\{P(\phi, \theta)\}$	For all ϕ in δ after θ is $P(\phi, \theta)$ true?
Existential Before	$Ebefore(\phi, \delta, \theta)\{P(\phi, \theta)\}$	Does ϕ exist in δ before θ where $P(\phi, \theta)$ is true?
Existential After	$Eafter(\phi, \delta, \theta)\{P(\phi, \theta)\}$	Does ϕ exist in δ after θ where $P(\phi, \theta)$ is true?
Subset	$S(\phi, \delta)\{P(\phi)\}$	Create a subset of δ , such that for each element ϕ in that subset, $P(\phi)$ is true.
Universal From Subset	$A(\theta, S(\phi, \delta)\{P(\phi)\})\{Q(\theta)\}$	For each elements θ in the set δ for which $P(\phi)$ is true, $Q(\theta)$ is also true
Anchor Select	$Anchor(\phi, MR, "mrid")P(\phi)$	Evaluate $P(\phi)$ on the entity of type \mathbb{MR} with id "mrid"
count	$count(\delta)$	Count the number of elements of the subsets δ
Sum	$Sum(\phi, \delta)\{P(\phi)\}$	Summate the predicate $P(\phi)$ for all ϕ in δ
Average	$Avg(\phi, \delta)\{P(\phi)\}$	Get the average of the predicate $P(\phi)$ for all ϕ in δ
Count	$Count(\phi, \delta)\{P(\phi)\}$	Count the number of elements ϕ in δ where $P(\phi)$ is true.

5. SUMMARY

This paper presents a formal model to describe SCSs. This model is then used to define a query language, SCQL, that can be used to pose queries on the SCSs. The objective of SCQL is to be domain specific and to support temporal logic operators in those queries. We have demonstrated the use of SCQL with example queries, and demonstrated their effectiveness by running those queries against the SCS of 5 different large, mature software projects.

While it is possible to use other query languages to investigate SCSs (such as SQL and XQuery) we believe that SCQL has 2 important properties that these languages are do not. First, it is domain specific: the queries refer to entities in the repository, and second, it supports temporal logic operators. While it is possible to implement temporal logic operations in SQL or XQuery, it might result in overly complex expressions.

We expect to use SCQL in the exploration of the evolution of software and to help us compute metrics on SCS repositories.

6. REFERENCES

- [1] S. Abiteboul, L. Herr, and J. Van den Bussche. Temporal versus first-order logic to query temporal databases. pages 49–57, 1996.
- [2] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the International Conference on Software Maintenance (ICSM 2003)*, pages 23–32, Sept. 2003.
- [3] D. German. An empirical study of fine-grained software modifications. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, Sept 2004.
- [4] D. M. German. Mining CVS repositories, the softChange experience. In *1st International Workshop on Mining Software Repositories*, pages 17–21, May 2004.
- [5] D. M. German, A. Hindle, and N. Jordan. Visualizing the evolution of software using softchange. In *Proceedings SEKE 2004 The 16th International Conference on Software Engineering and Knowledge Engineering*, pages 336–341, 3420 Main St. Skokie IL 60076, USA, June 2004. Knowledge Systems Institute.
- [6] T. Girba, S. Ducasse, and M. Lanza. Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, Sept 2004.
- [7] A. E. Hassan and R. C. Holt. Predicting change propagation in software systems. pages 284–293, September 2004.
- [8] Y. Liu and E. Stroulia. Reverse Engineering the Process of Small Novice Software Teams. In *Proc. 10th Working Conference on Reverse Engineering*, pages 102–112. IEEE Press, November 2003.
- [9] A. Mockus, R. T. Fielding, and J. Herbsleb. Two Case Studies of Open Source Software Development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):1–38, July 2002.
- [10] X. Wu. Visualization of version control information. Master's thesis, University of Victoria, 2003.
- [11] T. Zimmermann and P. Weisgerber. Preprocessing cvs data for fine-grained analysis. In *1st International Workshop on Mining Software Repositories*, May 2004.

Integration and Collaboration

Developer identification methods for integrated data from various sources

Gregorio Robles, Jesus M. Gonzalez-Barahona

{grex, jgb}@gsyc.escet.urjc.es
Grupo de Sistemas y Comunicaciones
Universidad Rey Juan Carlos
Madrid, Spain

ABSTRACT

Studying a software project by mining data from a single repository has been a very active research field in software engineering during the last years. However, few efforts have been devoted to perform studies by integrating data from various repositories, with different kinds of information, which would, for instance, track the different activities of developers. One of the main problems of these multi-repository studies is the different identities that developers use when they interact with different tools in different contexts. This makes them appear as different entities when data is mined from different repositories (and in some cases, even from a single one). In this paper we propose an approach, based on the application of heuristics, to identify the many identities of developers in such cases, and a data structure for allowing both the anonymized distribution of information, and the tracking of identities for verification purposes. The methodology will be presented in general, and applied to the GNOME project as a case example. Privacy issues and partial merging with new data sources will also be considered and discussed.

1. INTRODUCTION

Most research in the area of mining software repositories has been performed on a single source of data. The reason for this is that tools are usually targeted towards accessing a specific kind of data, which can be retrieved and analyzed uniformly. Data mining for control versioning systems [11], bug-tracking systems, mailing lists and other sources is currently state of the art. The focus of these studies is more on the analysis than in the data extraction process, which can be automated, as has already been discussed [2, 9].

However, there is a wide interest in considering data from several sources and integrating them into a single database, getting richer evidence from the observed matter [5]. The data gathered following this approach can be used for studying several kinds of artifacts relevant to the software develop-

ment process, such as source code files or, as we will discuss in this paper, developers.

As an example of the usefulness of this approximation, let's consider collaboration in libre software¹ projects, which is an active research field. Libre software is produced in part (in many cases a large part) by volunteers, which makes it difficult to predict the future evolution. However, it has at least in some cases produced high-quality software, used by millions of persons around the world. It has been shown that this collaboration follows a Pareto law for commits [11], source code contributions [4], bug reports [8] or mailing list posts [6]; i.e. a small amount of developers of around 20% is responsible for a huge amount of the produced artifacts (around 80%). But although this research on different sources coincide in results, there is still no evidence of coherence. In other words, although it is known that the Pareto distribution appears in several data sources for a given project, are the most active actors for each of those sources (mailing lists, code repositories, bug report systems, etc.) the same ones?

In the specific case of merging information about developers from different repositories, the main difficulty is caused by the many identities that they use from repository to repository, and even for the same one, making tracking difficult. That is the reason why we need methods and tools that can find the different identities of a given developer. These methods, and the data they produce, should be designed to be sharable among research groups, not only for validation purposes but also for enabling the merging of partial data obtained by different teams from different sources.

In general, any study considering individuals in libre software projects, even when using a single data source, is sensible to identity variety. Before performing any analysis on the data set, it is necessary to merge the identities corresponding to the same person. This is for instance the case in the promising case of clustering [3] and social network analysis [7], which are trying to get insight in the structure of libre software projects.

The structure of this paper is as follows. The next section deals with the kinds of identities which are usually found in software-related repositories. The third section is devoted to the extraction of data, its structure and verification. Section

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR'05, May 17, 2005, St. Louis, Missouri, USA.

Copyright 2005 ACM 1-59593-123-6/05/0005 ...\$5.00.

¹Through this paper we will use the term "libre software" to refer to any code that conforms either to the definition of "free software" (according to the Free Software Foundation) or "open source software" (according to the Open Source Initiative).

four deals with heuristics for matching identities. Handling data about developers raises some privacy concerns, which are discussed in the fifth section, including some suggestions and solutions for sharing data without violating anonymity. We finish the paper with a section on conclusions and further work. We also include two appendixes, one with some results of applying the methodology to some GNOME repositories, and the other to post-matches analysis.

2. IDENTITIES IN SOFTWARE REPOSITORIES

Libre software developers, or more broadly, participants in the creation of libre software (from now on actors) usually interact with one or more Internet-based systems related to the software production and maintenance, some of which are depicted in Figure 1. These systems usually require every actor to adopt an identity to interact with them. This identity is usually different for every system, and in some cases a given author can have more than one identity for the same system, sometimes successive in time, sometimes even contemporary.

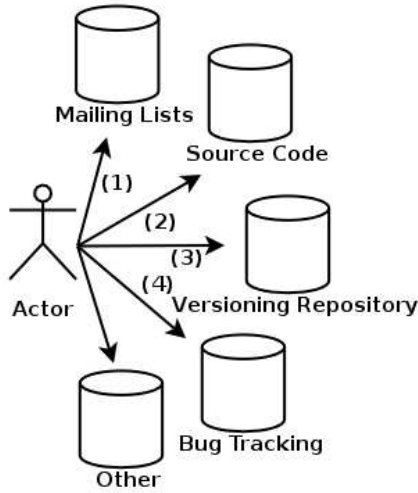


Figure 1: Different systems with which an actor may interact.

Some kinds of identities are the following (summarized in Table 1):

- An actor may post on mailing lists with one or more e-mail addresses (some times linked to a real life name).
- In a source file, an actor can appear with many identities: real life names (such as in copyright notices), e-mail addresses, RCS-type identifiers (such as those automatically maintained by CVS), etc.
- The interaction with the versioning repository occurs through an account in the server machine, which appears in the logs of the system.
- Bug tracking systems require usually to have an account with an associated e-mail address.

Other sources may include entries in weblogs, forums, blogs, etc. Although they are not considered in this study, the approach proposed could easily include them.

Type	Data Source	Primary Identities
(1)	Mailing lists	username@example.com
(1)	Mailing lists	Name Surname
(2)	Source Code	(c) Name Surname
(2)	Source Code	(c) username@example.com
(2)	Source Code	\$id: username\$
(3)	Versioning System	username
(4)	Bug Tracking	username@example.com

Table 1: Identities that can be found for each data source.

Given the various identities linking an actor to his actions on a repository, our goal is to determine all which correspond to the same real person. Basically we can classify these identities in two types: primary and secondary.

- Primary are mandatory. For instance, actors need an e-mail address to post a message to a mailing list. Mailing lists, versioning system and bug tracking system require to have at least a mandatory identity in order to participate (although in some exceptional cases this can be done anonymously). Source code does not have primary identities, except in some special projects where the copyright notice or some other authorship information is mandatory.
- Secondary are redundant. For instance, actors may provide their real-life name in the e-mails they send, but this is not required. Secondary identities usually appear together with primary identities, and may help in the identification process of actors.

Note that the relationships between actors and repositories have not to be unique: an actor could have one or more different identities in any repository. Even in cases such as CVS repositories, a given actor may change the username of his account, and of course the same actor could have different usernames in different CVS repositories.

3. DATA FETCHING, STRUCTURE AND VERIFICATION

Figure 2 shows a glimpse of the data structures used to learn the identities that correspond to the same person in several data sources.

All the identities are introduced into the database in the Identities table. This table is filled by directly extracting identities (using heuristics to locate them) from software-related repositories. Besides the identity itself, this table stores identifiers for the repository (data source) where it was found, which could be of value not only in the latter matching process, but also for validation and track-back purposes. The kind of identity (login, email address, “real name”) is also stored, to ease the automatic processing. Hashes of identities are added to provide a mechanism which can be used to deal with privacy issues, as will be described in a later subsection.

When extracting identities, sometimes relationships among them can be inferred. For instance, a real name can be next

to an e-mail address in a From field in a message. Those relationships are captured as entries in the Matches table, which will be the center of the matching (identification of identities of the same person) process. The ‘evidence’ field in this table provides insight about every identified match. As the process we are performing is mostly automatic, the value of ‘evidence’ will contain the name of the heuristic that has been used. This will include automatic heuristics, but also human inspection and verification. Sometimes, the information is not enough to ensure that the match is true for sure, and that is the reason why a field showing the estimated probability has been added. Fields that have been verified by humans with absolute certainty will be assigned a probability of 1.

With the information stored in Identities and Matches, the identification process may begin. Unique actors are identified with information in Matches, filling the Identifications table, and choosing unique person identifiers. Other information in the Persons table can be filled directly with data from the repositories or from other sources.

4. MATCHING IDENTITIES IN MORE DETAIL

We will usually have many identities for every actor. For instance, we can have name(s), username(s) and e-mail address(es). Every actor considered will have at least one of them, although possibly he may be identified with several, as is shown in Figure 3.

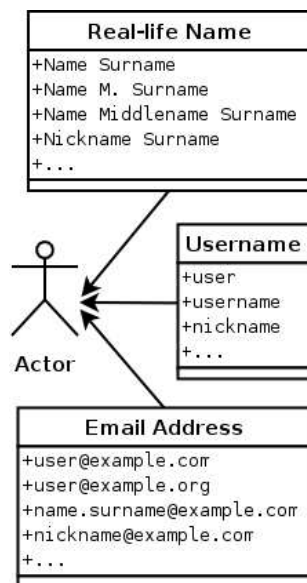


Figure 3: An actor with three different kinds of identities

Our problem is how to match all the identities that correspond to the same actor. In other words, we want to fill the Matches table with as much information as possible (and as accurate as possible). As already mentioned this is done using heuristics. Let’s expose some of them with some detail:

- In many cases it is common to find a secondary identity associated to a primary one. This happens often in

mailing lists, source code authorship assignments and bug tracking system. In all these cases, the primary identity (usually an e-mail address) may have a ‘real life’ name associated to it. Consider, for instance, *Example User* <username@example.com>, which implies that *Example User* and <username@example.com> correspond to the same actor. GPG key rings can also be a useful source of matches. A GPG key contain a list of e-mail addresses that a given person may use for encryption and authentication purposes. GPG is very popular in the libre software community and there exist GPG servers that store GPG keys with all these information.

- Sometimes an identity can be built from another one. For instance, the ‘real life’ name can be extracted in some cases from the e-mail username. Many e-mail addresses follow a given structure, such as *nsurname@example.com*, *name.surname@example.com* or *name.surname@example.com*. We can easily infer in those cases the ‘real life’ name of the actor.
- In many cases one identity is a part of some other. For instance, it is common that the username obtained from CVS is the same as the username part of the e-mail address. This can be matched automatically, and later verified by other means. This is one of the more error-prone heuristics, and is of course not useful for very popular usernames like ‘joe’. But despite these facts, it has proven to be very useful.
- Some projects or repositories maintain specific information that can be used for matching (for instance, because a list of contributors is maintained). As an example, the KDE project maintains a file which lists, for every person with write access to the CVS, his ‘real life’ name, his username for CVS and an e-mail address. Other similar case are developers registered in the SourceForge.net platform, who have a personal page where they may include their ‘real life’ name.

Of course this is not an exhaustive list, and combinations of the described heuristics can be used. For instance, a mixed approach could benefit from the data in Changelog files [1] for finding identity matches.

Usually, the fraction of false positives for matches can be minimized by taking into account the project from which the data was obtained. If we have a ‘joe’ entry as username for the CVS repository in an specific project, and in that same project we find somebody whose e-mail address is joe@example.com (and no other e-mail address that could be suspicious of being from a ‘joe’) then there is a high probability that both are identities of the same actor.

In any case, the fraction of false positives will never be zero for large quantities of identities. Therefore, some heuristics are specifically designed for cleaning the Matches table (eliminating those entries which are not correct, despite being found by an heuristic) and verification, including human verification. In some cases, the help from an expert that knows about the membership of a project, for instance, should be of great help.

But even after cleaning and verification, some matches will be false, and some will be missing, which can cause problems. However, since we are interested in using the

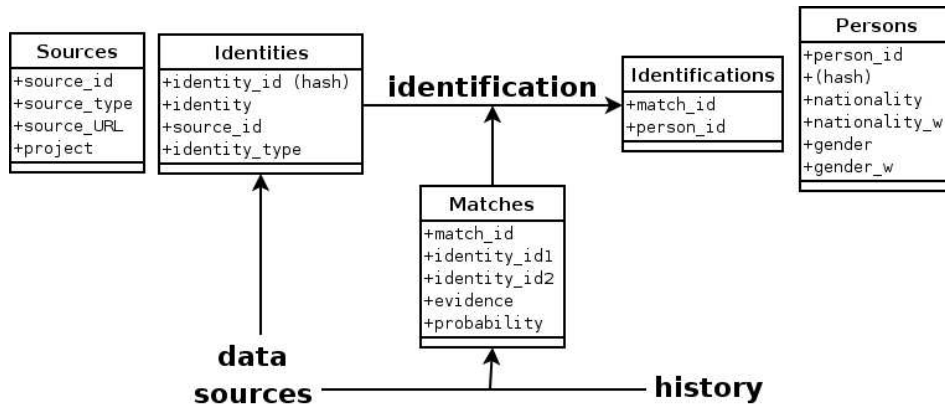


Figure 2: Main tables involved in the matching process and identification of unique actors

collected data for statistical purposes, this is not a big issue provided the error rate is small enough.

5. PRIVACY ISSUES

Privacy is of course an important concern when dealing with sensible data like this. Although all the information used is public, and it hardly contains any private data, the quantity and detail of the information available for any single developer after processing may cause privacy problems. Therefore, we have devised a data schema which allows both for the careful control of who has access to linking data to identified real persons, and for the distribution of information preserving anonymity. In the latter case, the information can be distributed in such a way that real persons are not directly identifiable, but new data sets can be, however, combined with the distributed one. This will for sure allow for a safe exchange of information between research groups.

For this purpose, the hashes of identities serve as a firewall. They are easy to compute from real identities, but are not useful for recovering them when only the hashes are available. Therefore, the Matches, Identifications and Persons tables can be distributed without compromising the real identities of developers as a whole. However, new data sets can be combined. Assuming a research group has a similar schema, with some identities found, the corresponding hash can be calculated for any of them and it may be looked up in the Matches table. Of course this will not be useful in many cases for finding new matches, but it would always allow to link an identity (and the data associated with it) to an actor in the Persons table. Therefore, any development data distributed using hash identities instead of developer names can be safely shared (but see below).

Although hashes will make it impossible to track real persons from the distributed data, it is still possible to look for certain persons in the data set. By hashing the usual identifiers of those persons, they can be found in the Matches table, and their identity is thus discovered. That is the reason why although distributing hashes to other research groups under reasonable ethical agreements is acceptable, probably it is not to do the same for anyone.

To avoid this problem, our schema has still a second level of privacy firewall: the person identifier in the Persons table. This identifier is given in such a way that it cannot be used

in any way to infer the identities of an actor without having access to the Identifications table. Therefore it is enough to key all development data with this person identifiers, and distributing only the Persons table in addition to that data to ensure the full privacy of the involved developers.

Of course, even in this latter case somebody could go to the software repositories used to obtain the data, and try to match the results with the distributed information. But this is an unavoidable problem: a third party can always milk the same repositories, and obtain exactly the same data, including real identities. In fact, this is the basis of the reproducibility of the studies.

6. CONCLUSIONS AND FURTHER WORK

Actors in libre software projects may use many different identities when interacting with different systems related to the development (and even with just a single one). When studying repositories related to libre software development it is very important to find those corresponding to the same person, so that actions can be assigned to the corresponding actor.

In this paper we have presented a design for dealing with this problem, and a methodology, based on heuristics, to identify as accurately as possible the different identities of the involved actors. For that, we use information stored in the repositories, and rely on some properties of the identifiers. This information can also be used to infer some personal information, such as the gender or the nationality (as is shown in appendix).

We have also discussed how privacy issues can be dealt with in our schema, including the distribution of anonymized information about the development, and have presented some results of performing the described study on some repositories of the GNOME project (in appendix).

We are currently testing our approach with larger data sets from several projects at once, and also starting to use it for sharing development data with other research groups. In the future, we are planning to include the functionality described in our GlueTheos tool [10], and to use it widely to obtain estimations of the number of people involved in libre software development, and their activities. We expect to use this data in combination with data from surveys and other sources to get a more complete view of the libre software

development landscape.

7. ACKNOWLEDGEMENTS

This work has been funded in part by the European Commission, under the CALIBRE CA, IST program, contract number 004337, by the Universidad Rey Juan Carlos under project PPR-2004-42 and by the Spanish CICyT under project TIN2004-07296.

8. REFERENCES

- [1] A. Capiluppi, P. Lago, and M. Morisio. Evidences in the evolution of os projects through changelog analyses. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, 2003.
- [2] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, Portland, USA, 2003.
- [3] R. A. Ghosh. Clustering and dependencies in free/open source software development: Methodology and tools. *First Monday*, 8(4), Apr. 2003.
- [4] R. A. Ghosh and V. V. Prakash. The orbiten free software survey. *First Monday*, 7(5), May 2002.
- [5] J. M. Gonzalez-Barahona and G. Robles. Getting the global picture. In *Proceedings of the Oxford Workshop on Libre Software 2004*, Oxford, UK, June 2004.
- [6] S. Koch and G. Schneider. Effort, cooperation and coordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27–42, 2002.
- [7] L. Lopez, J. M. Gonzalez-Barahona, and G. Robles. Applying social network analysis to the information in cvs repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburg, UK, 2004.
- [8] A. Mockus, R. T. Fielding, and J. D. Herbsleb. Two case studies of Open Source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology*, 11(3):309–346, 2002.
- [9] G. Robles, J. M. Gonzalez-Barahona, J. Centeno, V. Matellan, and L. Roderio. Studying the evolution of libre software projects using publicly available data. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 111–115, Portland, USA, 2003.
- [10] G. Robles, J. M. Gonzalez-Barahona, and R. A. Ghosh. Gluetheos: Automating the retrieval and analysis of data from publicly available software repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, Edinburg, Scotland, UK, 2004.
- [11] G. Robles, S. Koch, and J. M. Gonzalez-Barahona. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, Edinburg, Scotland, UK, 2004.

APPENDIX

A. A CASE STUDY: GNOME

To debug and complete our methodology, we have applied it to the data from several real libre software repositories. One of the most complete studies we have performed to date has been on the GNOME project, retrieving data from mailing lists, bug tracking system (including bug reports and comments) and from the CVS repository. Next, we offer some results from this study:

- 464,953 messages from 36,399 distinct e-mail addresses have been fetched and analyzed.
- 123,739 bug reports, from 41,835 reporters, and 382,271 comments from 10,257 posters have been retrieved from the bug tracking system.
- Around 2,000,000 commits, made by 1,067 different committers have been found in the CVS repository.
- From these data, 108,170 distinct identities have been identified.
- For those distinct identities, 47,262 matches have been found, of which 40,003 were distinct (therefore, our Matches table contains that number of entries).
- Using the information in the Matches table, we have been able of finding 34,648 unique persons.

This process has been statistically verified by selecting a sample of identities, looking by hand for matches and comparing the results to the corresponding entries in the Matches table. Currently we are completing the Persons table, and performing gender and nationality analysis.

B. AUTOMATIC (POST-IDENTIFICATION) ANALYSIS

The reader has probably noted that the Persons table in Figure 2 includes some fields with personal information. We have devised some heuristics to infer some of them from data in the repositories, usually from the structure of identities. For instance, nationality can be guessed by several means:

- Analyzing the top level domain (TLD) of the various e-mail addresses found in the identities could be a first possibility. The algorithm in this case consists of listing all e-mail addresses, extracting the TLD from them, rejecting those TLD that cannot be directly assigned to a country (.com, .net, .org, etc.) or those who are from “fake” countries (.nu, etc.), and finally looking at the remaining TLDs and count how often they occur. The TLD that is more frequent gives a hint about the nationality of the person. Of course this heuristic is specially bad for US-based actors (since they are not likely to use the US TLD), and for those using .org or .com addresses, quite common in libre software projects.
- Another approach is to use whois data for the second level domain in e-mail address, considering that the whois contact information (which includes a physical mail address) is valid as an estimator of the country of the actor. Of course, this is not always the case.

Other case example of information which can be obtained from identities is the gender. Usually we can infer the gender from the name of the person. However, in some cases it depends on the nationality, since some names may be assigned to males in one country and to females in another. This is for instance the case for Andrea, which in Italy is a male name while in Germany, Spain and other countries is usually for females.

Accelerating Cross-Project Knowledge Collaboration Using Collaborative Filtering and Social Networks

Masao Ohira Naoki Ohsugi Tetsuya Ohoka Ken-ichi Matsumoto

Graduate School of Information Science
Nara Institute of Science and Technology
8916-5, Takayama, Ikoma, Nara, JAPAN 630-0192
tel.+81(743)-72-5318 fax.+81(743)-72-5319

{masao, naoki-o, tetsuy-o, matumoto}@is.naist.jp

ABSTRACT

Vast numbers of free/open source software (F/OSS) development projects use hosting sites such as Java.net and SourceForge.net. These sites provide each project with a variety of software repositories (e.g. repositories for source code sharing, bug tracking, discussions, etc.) as a media for communication and collaboration. They tend to focus on supporting rich collaboration among members in each project. However, a majority of hosted projects are relatively small projects consisting of few developers and often need more resources for solving problems. In order to support cross-project knowledge collaboration in F/OSS development, we have been developing tools to collect data of projects and developers at SourceForge, and to visualize the relationship among them using the techniques of collaborative filtering and social networks. The tools help a developer identify “who should I ask?” and “what can I ask?” and so on. In this paper, we report a case study of applying the tools to F/OSS projects data collected from SourceForge and how effective the tools can be used for helping cross-project knowledge collaboration.

Categories and Subject Descriptors

H.5 [Information Interfaces and Presentation]: Group and Organization Interfaces—*Collaborative computing, Computer-supported cooperative work, Organization Design, Web-based interaction*

General Terms

Management, Measurement, Human Factors

Keywords

Knowledge Collaboration, Social Networks, Collaborative Filtering, Visualization Tool

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. MSR'05, May 17, 2005, Saint Louis, Missouri, USA

Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00.

1. INTRODUCTION

Vast numbers of free/open source software (F/OSS) development projects use hosting sites such as Java.net and SourceForge.net. These sites provide each project with a variety of software repositories (e.g. repositories for source code sharing, bug tracking, discussions, etc.) which can be seen as knowledge repositories for software development in the aggregate. Many researchers focus on exploiting such the repositories for supporting software development nowadays [4, 9].

While each project can accumulate its own knowledge through software development into the repositories easily, the “freely accessible” knowledge across projects is not supported sufficiently. In order to help cross-project knowledge collaboration in F/OSS development, we have been developing tools to collect data of projects and developers at SourceForge, and to visualize the relationship among them using the techniques of collaborative filtering and social networks. The tools help a developer identify “who should I ask?” and “what can I ask?” and so on.

In what follows, we first discuss the need for supporting cross-projects knowledge collaboration based on our analysis of SourceForge. Then we describe the procedure of mining software repositories at SourceForge using our tools. In the next section, we report a case study of applying Graphmania to the F/OSS projects data collected from SourceForge and illustrate how effective the tool can be used for helping cross-project knowledge collaboration.

2. NEED FOR KNOWLEDGE COLLABORATION ACROSS PROJECTS

Recent studies on F/OSS communities revealed that F/OSS communities needed further developers and people's contribution to software development. For instance, [8] reported only 4% of developers in the Apache community created 88% of new code and fixed 66% of defects. From a total of 196 developers in the Ximian project, 5 developers account for 47% of the modification requests (MRs), while 20 account for 81% of the MRs, and 55 have done 95% of them [3]. 4% of members account for 50 percent of answers on a user-to-user help site [5].

Projects with a large proportion of non-contributors have difficulty providing needed services such as bug fixes and software enhancements to all members [1]. The existence of highly motivated members would be the key success factor of a F/OSS project[2]. As an approach to motivate members

Table 1: Number of Projects with n Developers

NUMBER OF DEVELOPERS	NUMBER OF PROJECTS	(%)
0	278	0.3
1	60665	66.7
2	14151	15.6
3	5854	6.4
4	3222	3.5
Over 5	6732	7.4
TOTAL	90902	100

Table 2: Number of Developers on p Projects

NUMBER OF PARTICIPATING PROJECTS	NUMBER OF DEVELOPERS	(%)
1	100408	77.3
2	18753	14.4
3	5980	4.6
4	2350	1.8
Over 5	2406	1.8
TOTAL	129897	100

of online communities, some theories such as social capital (e.g. ExpertsExchange¹) and social networks [6, 12] have attracted attention recently.

Relatively small projects registered at hosting sites are confronted with more difficulties than such the large projects mentioned above (e.g. the Apache project), because (1) those projects consist of few developers and contributors generally and (2) the hosting sites provide a variety of tools for rich communication and collaboration among members in each project but do not provide them with tools for exchanging or sharing problem-solving knowledge across projects directly.

To confirm the issue related to (1), we collected and analyzed the data of over 90,000 projects and about 130,000 developers² at SourceForge in February 2005. Table 1 shows the number of projects with n developers. 66.7% of overall projects at SourceForge had only one developer. The maximum of number of developers in one project was 272. Table 2 shows the number of developers on p projects. 77.3% of overall developers at SourceForge belonged to one project. The maximum of number of projects a developer joined was 51.

These results are very similar to the results of the social networks analysis in SourceForge in February 2002 [7]. As Madey et al. mentioned in [7], these results indicate that a small number of developers at SourceForge have rich links to others (i.e. the “rich-get-richer” effect) but a majority of developers does not have sufficient links to ask other projects’ developers to help them solve problems which happened in their own projects. We believe that it is useful to give developers in small projects means to access other developers and projects that possess the information

¹<http://www.experts-exchange.com>

²The total number of registered users at SourceForge.net are over 1,000,000. We collected the data of members who are participating in projects actually.

Table 3: Project Information (e.g. the phpMyAdmin Project at SourceForge.net)

ATTRIBUTES	EXAMPLE
project name	phpMyAdmin
description	phpMyAdmin is a tool written in PHP intended to handle the administration of MySQL ...
num. of developers	8
keywords	php, databases, ...
program lang.	PHP
operating system	OS Independent
license	GPL
status	5 – Production/Stable
registered	2001/3/18 02:07
intended audience	Developers, End Users/Desktop, System Administrators
user interface	Web-based
topics	Front-Ends, Dynamic Content, Systems Administration

Table 4: Developer Information (e.g. One of authors registered at SourceForge.jp)

ATTRIBUTES	EXAMPLE
login name	Ohsugi
public name	Naoki Ohsugi
email address	ohsugi at users.sourceforge.jp
site member since	2002/6/10 22:16
group member of	NAIST Collaborative Filtering Engines, Game-R, Bullfrog
skill inventory	C/C++: Competent: 5 yr–10 yr Perl: Competent: 5 yr–10 yr Java: Want to Learn: 2 yr–5 yr

or knowledge relevant to solving problems. The next section describes the procedure of mining software repositories at SourceForge using our tools and then illustrates how the tools works.

3. GRAPHMANIA: A TOOL FOR HELPING KNOWLEDGE COLLABORATION

In order to support cross-project knowledge collaboration, we have been developing Graphmania, a tool for visualizing the relationship among developers and projects using the techniques of collaborative filtering and social networks. The tool helps a developer identify “who should I ask?” and “what can I ask?” and so on.

3.1 Data Collection

Using an autopilot tool for SourceForge.net³ written in Ruby, we have collected two data sets; about 90,000 projects’ information (table 3) and 130,000 developers’ information (table 4). In what follows, for simplicity, we suppose that the data we use in this paper is **project name** and **num. of developers** from the project info. , and **login name** and **group members of** from the developer info. only (other attributes will be used in the near future).

³available from the third author upon your requests

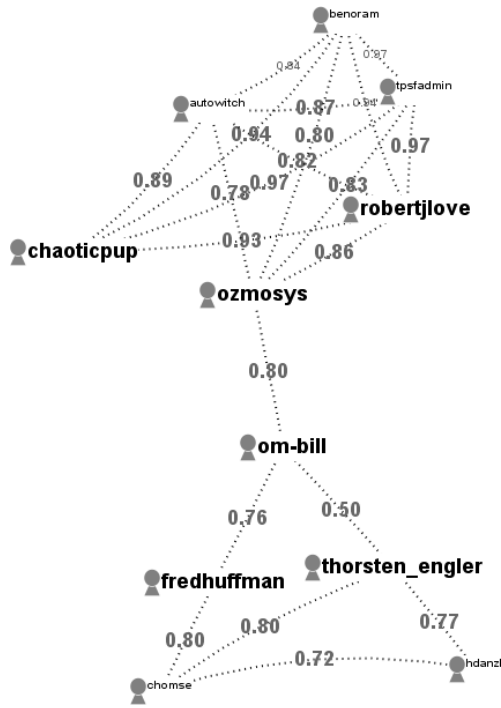


Figure 2: Developer networks (The two developers play a role of a linchpin between two social networks)

consists of two different types of nodes and three different types of edges. A dotted edge connects two developers who are working together for same projects. A dash-dotted edge connects two projects that have same developers. Each black line edge represents the relationship among projects and developers (i.e. who is working for which projects and which projects have whom).

Graphmania uses HyperGraph⁵ to provide users with hyperbolic views for visualizations and with interactivity to visualized results. Hyperbolic visualizations help users understand information in detail while keeping an overview of information (such the technique is called “focus + context”). Since each node can have an URL to a website as a function of HyperGraph, users are able to access to the site (developers’ information pages or projects’ HP) as soon as users can find an interesting node.

4. A CASE STUDY

This section describes a case study of applying Graphmania to F/OSS projects data collected from SourceForge and how effective it can be used. As a condition of the similarity calculation, we selected nodes with maximum 5 edges. This means we used only a few percent of over 90,000 projects data for reducing the amount of the similarity calculation.

Developer networks:

Figure 2 represents a part of developers networks snipped. If you have maximum 5 edges, you can find the strength of each edge comparing with similarities because it implies shared history of participating in same projects. Even if

⁵<http://hypergraph.sourceforge.net/>

you have only a few edges, it would be also helpful to notice important developers who play a role of a linchpin in the cluster because you have possibilities to contact others via the linchpin. For contacting others, just click the node you are interested in and then you will be able to reach a developers’ information page. Developer networks are basically same ones as social networks in common online communities.

Project networks:

Figure 3 represents a part of project networks snipped. You can notice that similar name projects organize one cluster because this indicates that projects that share specific purposes or goals tend to have similar names (e.g. a project related to TurboPower have “tp” at the head of a project name). If you are a member of a project in such the cluster, you might find interesting projects related to software you create and might be able to obtain the useful information for your software development from the members of the project. Project networks are a good example of taking advantage of collaborative filtering by a common practice in which similar projects have similar project names.

Developer–project networks:

Using developer–project networks (figure 1), you can easily notice your neighborhoods who are joining similar projects with the red nodes and edges. You are easy to ask something to these neighborhoods because they are likely acquaintances and seem to have similar interests of F/OSS technologies. You can also recognize the projects’ neighborhoods spend much effort from number of developers. These projects might have ideas, technologies and solutions for problems, which you need. Developer–project networks are a bit complex but useful for finding developers and projects related to your own.

5. CONCLUSION AND FUTURE WORK

In this paper, we described the issues on motivating F/OSS (online) projects and needs for supporting knowledge collaboration across projects. We introduced Graphmania, a tool for visualizing the relationship among developers and projects using the techniques of collaborative filtering and social networks.

In the near future, we have a plan to use other attributes of the collected data listed in Table 3 and Table 4 for more effective visualizations based on NCFE. We would also like to extend the tool according to the Dynamic Collaboration (DynC) framework [13] because the current tool cannot help user control the amount of communication so that “rich” developers or projects can prevent taking a lot of questions and requests from “poor” developers or projects. Then we would like to evaluate the tool through actual uses of F/OSS developers.

6. ACKNOWLEDGMENTS

This work is supported by the EASE (Empirical Approach to Software Engineering) project⁶ and supported by Grant 15103 of the Open Competition for the Development of Innovative Technology program, the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology of Japan.

⁶<http://www.empirical.jp/>

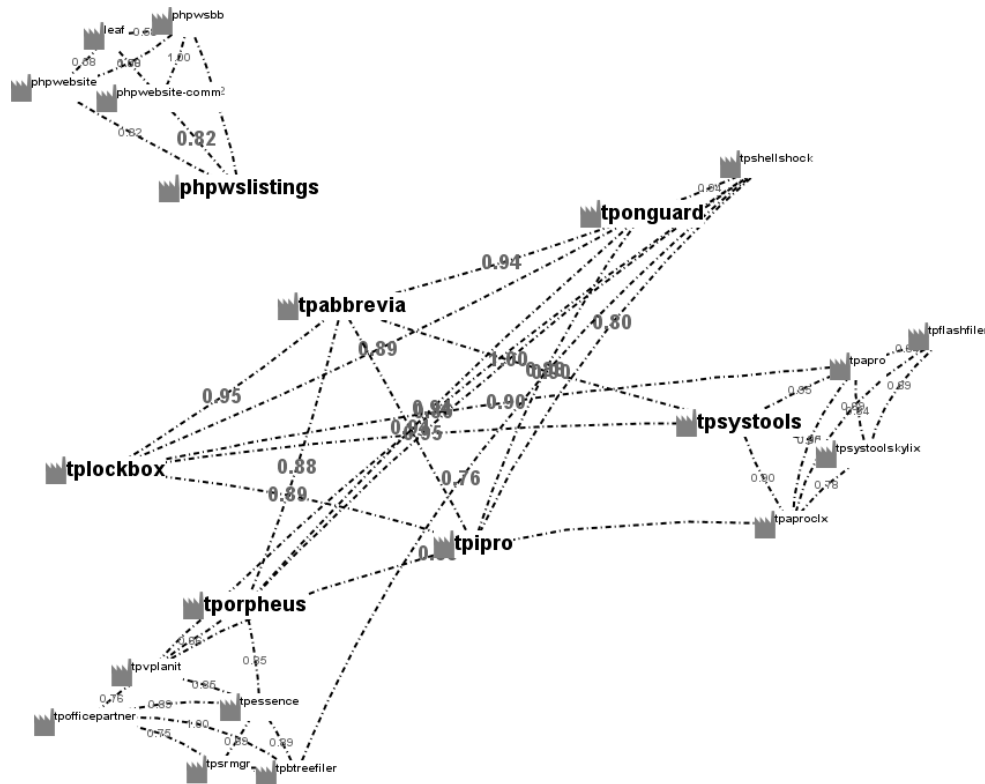


Figure 3: Project networks (The large cluster in center consists of projects related to TurboPower. The isolated small cluster on the upper left consists of projects related to Linux.)

7. REFERENCES

- [1] G. Beenen, K. Ling, X. Wang, K. Chang, D. Frankowski, P. Resnick, and R. E. Kraut. Using social psychology to motivate contributions to online communities. In *Proc. of the 2004 ACM conf. on Computer Supported Cooperative Work (CSCW'04)*, pages 212–221, 2004.
- [2] J. Feller and B. Fitzgerald. *Understanding Open Source Software Development*. Addison-Wesley, 2002.
- [3] D. German and A. Mockus. Automating the measurement of open source projects. In *Proc. of the 3rd Workshop on Open Source Software Engineering*, pages 63–67, 2003.
- [4] A. E. Hassan, R. C. Holt, and A. Mockus, editors. *Proc. of 1st Intl. Workshop on Mining Software Repositories (MSR2004)*, 2004.
- [5] K. R. Lakhani and E. von Hippel. How open source software works: “free” user-to-user assistance. *Research Policy*, 32(6):923–943, 2003.
- [6] L. Lopez-Fernande, G. Robles, and J. M. Gonzalez-Barahona. Applying social network analysis to the information in CVS repositories. In *Proc. of 1st Intl. Workshop on Mining Software Repositories (MSR2004)*, pages 101–105, 2004.
- [7] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In *Americas conf. on Information Systems (AMCIS2002)*, pages 1806–1813, 2002.
- [8] A. Mockus, R. Fielding, and J. D. Herbsleb. Two case studies of open source software development: Apache and mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002.
- [9] M. Ohira, R. Yokomori, M. Sakai, K. Matsumoto, K. Inoue, and K. Torii. Empirical project monitor: A tool for mining multiple project data. In *Proc. of 1st Intl. Workshop on Mining Software Repositories (MSR2004)*, pages 42–46, 2004.
- [10] N. Ohsugi. *A Framework for Software Function Recommendation Based on Collaborative Filtering*. NAIST-IS-DT0361006, Graduate School of Information Science, Nara Institute of Science and Technology, 2004.
- [11] N. Ohsugi, A. Monden, and K. Matsumoto. A recommendation system for software function discovery. In *Proc. of 9th Asia-Pacific Software Engineering conf. (APSEC'02)*, pages 248–257, 2002.
- [12] Y. Ye and K. Kishida. Toward an understanding of the motivation open source software developers. In *Proc. of the 25th Intl. conf. on Software Engineering (ICSE'03)*, pages 419–429, 2003.
- [13] Y. Ye, Y. Yamamoto, and K. Kishida. Dynamic community: A new conceptual framework for supporting knowledge collaboration in software development. In *Proc. of 11th Asia-Pacific Software Engineering conf. (APSEC'04)*, pages 472–481, 2004.

Collaboration Using OSSmole: a repository of FLOSS data and analyses

Megan Conklin
Elon University
Department of Computing Sciences
Elon, NC 27244
1(336)229-4362

mconklin@elon.edu

James Howison
Syracuse University
School of Information Studies
Syracuse, NY 13210
1(315)395-4056

jhowison@syr.edu

Kevin Crowston
Syracuse University
School of Information Studies
Syracuse, NY 13210
1(315)380-3923

crowston@syr.edu

ABSTRACT

This paper introduces a collaborative project *OSSmole* which collects, shares, and stores comparable data and analyses of free, libre and open source software (FLOSS) development for research purposes. The project is a clearinghouse for data from the ongoing collection and analysis efforts of many disparate research groups. A collaborative data repository reduces duplication and promote compatibility both across sources of FLOSS data and across research groups and analyses. The primary objective of *OSSmole* is to mine FLOSS source code repositories and provide the resulting data and summary analyses as open source products. However, the *OSSmole* data model additionally supports donated raw and summary data from a variety of open source researchers and other software repositories. The paper first outlines current difficulties with the typical quantitative FLOSS research process and uses these to develop requirements for such a collaborative data repository. Finally, the design of the *OSSmole* system is presented, as well as examples of current research and analyses using *OSSmole*.

Categories and Subject Descriptors

D.2.8 [Software Engineering]: Metrics – *complexity measures, process metrics, product metrics*.

General Terms

Measurement, Human Factors.

Keywords

Open source software, free software, libre software, data mining, data analysis, data repository, source control, defect tracking, project metrics.

1. INTRODUCTION

OSSmole is a collaborative project designed to gather, share and store comparable data and analyses of free and open source software development for academic research. The project draws on the ongoing collection and analysis efforts of many research groups, reducing duplication, and promoting compatibility both across sources of online FLOSS data and across research groups and analyses.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MSR '05, May 17, 2005, St. Louis, Missouri, USA.

Copyright 2005 ACM 1-59593-123-6/05/0005...\$5.00.

Creating a collaborative repository for FLOSS data is important because research should be as reproducible, extensible, and comparable as possible. Research with these characteristics creates the opportunity to employ meta-analyses ("analyses of analyses") which exploit the diversity of existing research by comparing and contrasting existing results to expand knowledge. Unfortunately, the typical FLOSS research project usually proceeds in a way that does not necessarily achieve these goals. Reproducing, extending, and comparing research project results requires detailed communal knowledge of the many choices made throughout a given research project. Traditional publication methods prioritize results but mask or discard much of the information needed to understand and exploit the differences in the data collection and analysis methodologies of different research groups. *OSSmole* is designed to provide resources and support to academics seeking to prepare the next generation of FLOSS research.

2. BACKGROUND AND METHOD

Obtaining data on FLOSS projects is both easy and difficult. It is easy because FLOSS development utilizes computer-mediated communications heavily for both development team interactions and for storing artifacts such as code and documentation. As many authors have pointed out, this process leaves a freely available and, in theory at least, highly accessible trail of data upon which many academics have built interesting analyses. Yet, despite this presumed plethora of data, researchers often face significant practical challenges in using this data in a deliberative research discourse.

2.1. Data Selection

The first step in collecting online FLOSS data is selecting which projects and which attributes to study. Two techniques often used in estimation and selection are census and sampling. (Case studies are also used but these will not be discussed in this paper.)

Conducting a census means to examine all cases of a phenomena, taking the measures of interest to build up an entire accurate picture. Taking a census is difficult in FLOSS for a number of reasons. First, it is hard to know how many FLOSS projects there are 'out there' and hard to know which projects are actually in or out. For example, are corporate-sponsored projects part of the phenomenon or not? Do single person projects count? What about school projects?

Second, projects, and the records they leave, are scattered across a surprisingly large number of locations. It is true that many are located in the major general repositories, such as Sourceforge and GNU Savannah. It is also true, however, that there are a quickly growing number of other repositories of varying sizes and focuses (e.g. CodeHaus, GridForge, CPAN (the perl

repository) ...) and that many projects, including the well-known and well-studied Apache and Linux projects, prefer to "roll their own" tools. This locational diversity obscures many FLOSS projects from attempts at census. Even if a full listing of projects and their locations could be collated, there is also the practical difficulty of dealing with the huge amount of data—sometimes years and years of email conversations, source control data, and defect tracking data—required to conduct comprehensive analyses.

These difficulties suggest sampling, or the random selection of a small, and thus manageable, sub-group of projects which is then analyzed to represent the whole. While sampling could solve the manageability problem presented in census-taking, there is still another difficulty with both processes: the total population from which to take the sample selection is not well-defined. Perhaps more importantly, sampling open source projects is methodologically difficult because everything FLOSS research has shown so far points to massively skewed distributions across almost all points of research interest [1] [8]. Selecting at random from these highly skewed datasets will yield samples which will be heavily weighted to single-developer projects, or projects which are still in listings but which are stillborn, dormant, or dead. These are often not the most interesting research subjects.

The large skew also makes reporting distributions of results at best difficult and at worst misleading because averages and medians are not descriptive of the distribution. The difficulty of sampling is demonstrated in the tendency of FLOSS studies to firstly limit their inquiries to projects using one repository (usually Sourceforge), and often to draw on samples created for entirely different purposes (such as top-100 lists as in [6]), neither of which is a satisfactory general technique.

2.2. Data Collection

Once the projects of interest have been located, the actual project data must be collected. There are two techniques that prevail in the FLOSS literature for collecting data: web spidering and obtaining database dumps.

Spidering data is fraught with practical complexities [5]. Because the FLOSS repositories are usually maintained using a database back-end and a web front-end, the data model appears straightforward to reproduce. The central limitation of spidering, however, is that the researcher is continually in a state of discovery. The data model is always open to being changed by whoever is controlling the repository and there is usually no way that the researcher will know of changes in advance. Spidering is a time- and resource-consuming process, and one that is being unnecessarily replicated throughout the world of FLOSS research.

Getting direct access to the database is clearly preferable, but not all repositories make their dumps available. And understandably so: it is not a costless process to make data-dumps available. Dumps can contain personally identifiable and/or financial information (as with the Sourceforge linked donation system) and so must be anonymized or otherwise treated. Repositories are facing an increasing number of requests for database snapshots from academics and are either seeking a scalable way to do releases or declining to release the data¹. It is often unclear

whether database dumps obtained by one research project can be shared with other academics, so rather than possibly breach confidentiality or annoy their subjects by asking for signed releases, it is understandable that academics who do get a database dump do not make those dumps easily available.

Even when a dump is made available, it is necessary to interpret the database schema and identify missing data elements. This is not always as straightforward as one would expect. After all, the databases were designed to be used to build Web pages quickly, not to conduct academic analyses. Furthermore, they have been built over time and face the complexity that any schema faces when stretched and scaled beyond its original intended use: labels are obscured, extra tables are used, there are inconsistencies between old and recently-added data. The interpretation and transformation of this data into information that is interesting to researchers is not a trivial process, and there is no reason to think that researchers will make these transformations in a consistent fashion. It is also possible that some repositories do not themselves store the type of historical information about projects that would be interesting for academic research. For example, while a snapshot of a repository might show the current list of developers each project, it could be missing important historical information about which developers have worked on which projects in the past.

Even pristine and well-labeled data from repositories is not sufficient because different repositories store different data elements. Different forges can have projects with the same names; different developers can have the same name across multiple forges; the same developer can go by multiple names in multiple forges. In addition, forges have different terminology for things like developer roles, project topics, and even programming languages. The differences are compounded by fields which are named the same but which represent different data. This is especially true of calculated fields, such as activity or downloads, for which there is incomplete publicly-available information how these fields are calculated.

2.3. Data Validation

Once projects have been selected and the available data harvested, researchers must be confident that the data adequately represents the activities of a project. For example, projects may use the given repository tools to differing degrees: many projects are listed on Sourceforge, and use the mailing lists and web hosting provided there. But some of these same projects will shun the notoriously quirky *Tracker* bug-tracking system at Sourceforge, preferring to set up their own systems. Other projects host their activities outside Sourceforge but maintain a 'placeholder' registration there. These projects will often have very out-of-date registration information, followed by a link to an external Web site. It is very difficult, without doing detailed manual examination of each project, to know exactly how each project is using its repository tools. It is thus difficult to be confident that the data collected is a reasonable depiction of the project's activities.

Complete accuracy is, of course, not always required because in large scale data analysis some 'dirty' data is acceptably handled through statistical techniques. At a minimum, though, researchers contemplating the accuracy of their data must have some reason to believe that there are no systematic reasons that the data collected in the name of the group would be unrepresentative. Unfortunately, given the idiosyncrasies of FLOSS projects, confidence on this point appears to require project-by-project

¹ It is understood that an NSF funded project on which the Sourceforge project manager is a co-PI is planning to make Sourceforge dumps generally available, but the details of this project are, at the time of writing, not available. See <http://www.nd.edu/~oss/People/people.html>

verification, a time-consuming process for individual researchers and projects, and one which is presumably repeated by every researcher going through this information-gathering exercise.

The upshot of this issue is that each step of the typical FLOSS research process introduces variability into the data. This variability then underlies any quantitative analysis of FLOSS development. Decisions about project selection, collection, and cleaning are compounded throughout the cycle of research. FLOSS researchers have not, so far, investigated the extent to which this variability affects their findings and conclusions. The demands of traditional publication also mean that the decisions are not usually fully and reproducibly reported.

Our critique is not against the existence of differences in research methods or even datasets. There is, rightly, more than one way to conduct research, and indeed it is this richness that is at the heart of knowledge discovery. Rather, our critique is that the research community is currently unable to begin a meta-analysis phase because the current process of FLOSS research is hampered by variability, inconsistency, and redundant, wasted effort in data collection and analysis. It is time to learn from the free and open source approaches we are studying and develop an open, collaborative solution.

3. PROPOSED SOLUTION

3.1. Goals of OSSmole

The above problem description allows us to identify requirements for building a system to support research into FLOSS projects. We call the system we have built OSSmole. The OSSmole system is a central repository of data and analyses about FLOSS projects which have been collected and prepared in a decentralized, collaborative manner. Data repositories have been useful in other fields, forming datasets and interchange formats (cf ARFF) around which research communities focus their efforts. For example, the TREC datasets have supported a community of information retrieval specialists facilitating performance and accuracy comparisons². The GenBank is the NIH database of all publicly-available gene sequences.³ The PROMISE software engineering repository is a collection of data for building predictive models of the software engineering process.⁴ The goal of the OSSmole project is to provide a high-quality, widely-used database of FLOSS project information, and to share standard analyses for replication and extension of this data.

A data and analysis clearinghouse for FLOSS data should be:

Collaborative—The system should leverage the collective effort of FLOSS researchers. It should reduce redundancies in data collection and free a researcher's time to pursue novel analyses. Thus, in a manner akin to the BSD rather than the GPL licensing model, OSSmole expects but does not require that those that use data contribute additional data and the analysis scripts that they obtain or use.

Available—The system should make the data and analysis scripts available without complicated usage agreements, where possible through direct unmonitored download or database queries. This ease the startup requirements for new researchers who wish to implement novel techniques but face high data collection costs.

² <http://trec.nist.gov>

³ <http://www.ncbi.nlm.nih.gov/GenBank>

⁴ <http://promise.site.uottawa.ca/SERepository>

This will also lower the barriers to collegial replication and critique.

Comprehensive and compatible—Given the multiplicity of FLOSS project forges identified above, the system should cover more than just one repository. The system should also be able to pull historical snapshots for purposes of replication or extension of earlier analyses. Compatibility requires that the system should translate across repositories, allowing researchers to conduct both comprehensive and comparative analyses. There is also the potential to develop a data interchange format for FLOSS project collateral. FLOSS project leaders, fearing data and tool lock-in, might find this format useful as they experiment with new tools or and repositories.

Designed for academic research—The data model and access control features should be designed for convenience for academic researchers. This means a logical and systematic data model which is properly documented with well-labeled fields. The source of each data element should be known and transparent. Researchers should be able to trace the source of each data element so that they can make decisions about whether to include a particular record or attribute in their analyses.

Of high quality—Researchers should be confident that the data in the system is of high quality. The origins and collection techniques for individual data elements must be traceable so that errors can be identified and not repeated. Data validation performed routinely by researchers can also be shared (for example, scripts that sanity-check fields or distributions) and analyses can be validated against earlier ones. This is a large advantage over individual research projects which may be working with single, non-validated datasets. It reflects the “many-eyes” approach to quality assurance, familiar from FLOSS development practices.

Support reproducible and comparable analyses—The system should specify a standard application programming interface (API) for inserting and accessing data via programmed scripts. That allows analyses to specify, using the API, exactly the data used. It is also desirable that data extracted from the database for transformation be exported with verbose comments detailing its origin and how to repeat the extraction. The best way to ensure reproducible and comparable analyses is to have as much of the process as possible be script-driven. Ideally, these scripts could be available for analysis by the research community.

A system that meets these requirements, we believe, will promote the discovery of knowledge about FLOSS development by facilitating the next phase of extension through replication, apposite critique, and well-grounded comparisons.

3.2. OSSmole Data Model

The OSSmole data model is designed to support data collection, storage and analysis from multiple open source forges in a way that meets the above requirements. OSSmole is able to take both spidered data and data inserted from a direct database dump. The raw data is timestamped and stored in the database, without overwriting any data previously collected about the same project. Finally, periodic raw and summary reports are generated and made publicly-available on the project web site.

The type of data that is currently collected from the various open source forges includes: the full HTML source of the forge data page for the project, project name, programming language(s), natural language(s), platform(s), open source license type,

operating system(s), intended audience(s), and the main project topic(s). Developer-oriented information includes: number of developers, developer information (name, username, email), and the developer's role on the project. We have also collected issue-tracking data (mainly bugs) such as date opened, status, date closed, priority and so on. Data has been collected from Sourceforge, GNU Savannah, the Apache foundation's Bugzilla and Freshmeat. We are currently creating mappings between fields from each of these repositories and assessing how comparable the fields are. The forge-mapping task is extensive and time-consuming, but the goal is to build a dataset that is more complete and is not specific to only one particular forge.

Because OSSmole is constantly growing and changing as new forges are added, and because data from multiple collectors is both expected and encouraged, it is important that the database also store information about where each data record originally came from (i.e. script name, version, command-line options used, name and contact information of person donating the data, and date of collection and donation). This process ensures accountability for problematic data, yet encourages collaboration between data collectors. The information is stored inside the database to ensure that it does not get decoupled from the data. Donated raw data files are also stored in their original formats, in case of problems with the database imports or unforeseen mapping problems between projects.

Likewise, it is a general rule that data is not overwritten when project details change; rather, one of the goals of the OSSmole project is that a full historical record of the project be kept in the database. This will enable researchers to analyze project and developer changes over time and enable access to data that is difficult or impossible to access once it has slipped from the repositories front ends.

Access to the OSSmole project is two-pronged: both data and scripts are continually made available to the public under an open source license. Anyone can download the OSSmole raw and summary data for use in their own research projects or just to get information about "the state of the industry" in open source development. The raw data is provided as multiple text files; these files are simply tab-delimited data dumps from the OSSmole database. Summary files are compiled periodically, and show basic statistics. Examples of summary statistics that are commonly published would be: the count of projects using a particular open source license type, or the count of new projects in a particular forge by month and year, or the number of projects that are written using each programming language. It is our hope that more sophisticated analyses will be contributed by researchers and that the system will provide dynamic and up-to-date results rather than the static "snapshots" that traditional publication unfortunately leaves us.

The scripts that populate the OSSmole database are also available for download under an open source license. These scripts are given for two reasons: first, so that interested researchers can duplicate and validate our findings, and second, so that anyone can expand on our work, for example by modifying a script to collect data from a new forge. Indeed this process has begun with the recent publication of a working paper comparing and critiquing our spidering and summaries and beginning collaboration [7]. OSSmole expects and encourages contributions of additional forge data. (Each set of donated data is given a unique number so that the different "data sources" can be included or excluded for a given analysis. This allows us to

accept donated data, along with a description of where the data came from. This transparency gives researchers the ability to include or exclude the donation from their analyses.) Researchers interested in donating or using OSSmole data should see the OSSmole project page at <http://ossmole.sf.net> and join the mailing list for information on how to contribute.

4. RESULTS

Because it is a regularly-updated, publicly-available data repository, OSSmole data has been used both for constructing basic summary reports about the state of open source, as well as for more complex social network analyses of open source development teams. For example, summary reports posted as part of the OSSmole project regularly report the number of open source projects, the number of projects per programming language, the number of developers per project, etc. This sort of descriptive data is useful for constructing "state of the industry" reports, or for compiling general statistical information about open source projects. The OSSmole collection methods are transparent and able to be reproduced, so OSSmole can serve as a reliable resource for these metrics. Having a stable and consistently-updated source of this information will also allow metrics to be compared over time. One of the problems with existing analyses of open source project data is that researchers will run a collection and analyze it once, publish the findings, and then never run the analysis again. The OSSmole data model and collection methodology was designed to support historical comparisons of this kind.

OSSmole data was used in a number of large-scale social network analyses of FLOSS project development. Crowston and Howison [3] reports the results of a SNA centralization analysis in which the data suggests that, contrary to the rhetoric of FLOSS practitioner-advocates, there is no reason to assume that FLOSS projects share social structures. Further OSSmole data was used in the preparation of [2] which, in an effort to avoid the ambiguities of relying on ratings or downloads, develops a range of quantitative measures of FLOSS project success including the half-life of bugs. OSSmole makes available the full data and analysis scripts which make these analyses fully reproducible and, we hope, extendable.

Another project using OSSmole data [1] explored whether open source development teams have characteristics typical of a self-organized, complex network. This research investigated whether FLOSS development networks will evolve according to "rich get richer" or "winner take all" models, like other self-organized complex networks do. Are new links (developers) in this network attracted to the largest, oldest, or fittest existing nodes (project teams)? The OSSmole data was used to determine that there are indeed many characteristics of a complex network present in FLOSS software development, but that there may also be a mutual selection process between developers and teams that actually stops FLOSS projects from matching the "winner take all" model seen in many other complex networks.

Recently, another researcher, Dawid Weiss, collected data by spidering Sourceforge [7]. Weiss then compared the data and collection methodology to the OSSmole data collection techniques and results. He chose to focus mostly on the changes between when his results were gathered, and when the first OSSmole results were gathered a few months prior. There are two main differences noted in this technical report. First, he discovered that the Sourceforge management team made changes to the data in between the two gathering processes (specifically,

they relabeled all the target operating systems and recategorized them). Second, there are differences in how data is gathered and cleaned between research projects (specifically, the OSSmole team cleaned out any inaccessible project for which we could gather no information other than a name, but he did not do this cleaning). These two observations about the data collection and analysis effort are precisely why OSSmole desires to be a collaborative, "many eyes" approach.

The most interesting thing about the intersection of the Weiss research with OSSmole is that he found the OSSmole dataset without our assistance, conducted numerous analyses, then contacted our team to share his results. This experience illustrates the convenience and necessity of having a publicly-available dataset of this information. Because OSSmole is designed with collaboration in mind, these sorts of comparative results can be easily integrated into the OSSmole database, and then used in tandem with native OSSmole data or alone. As such, we have now fully integrated the Weiss data into the OSSmole database.

5. LIMITATIONS AND FUTURE WORK

There are, of course, limitations in the OSSmole project and our approach. Firstly, it is limited to data available online as a result of documented project activities. Certainly, these are not the only interactions FLOSS team members have. Thus while textual data like mailing lists, source control system history and comments, forums, and IRC chat logs could be included, OSSmole does not capture unlogged instant messaging or IRC, voice-over-IP or face-to-face interactions of FLOSS developers. Nor do we intend to store interviews or transcripts conducted by researchers which would be restricted by policies governing research on human subjects. We are also following the discussion about the ethical concerns of using data about open source projects closely [4].

There are also dangers in this approach which should be acknowledged. The standardization implied in this kind of repository, while desirable in many ways, runs the risk of reducing the valuable diversity that has characterized academic FLOSS research. We hope to provide a solid and traceable dataset and basic analyses which will support, not inhibit, interpretative and theoretical diversity. This diversity also means that research is not rendered directly comparable simply because analyses are based on OSSmole data or scripts. We are hopeful that OSSmole, by acting as a scaffold, will give researchers more time for such interesting work.

We will not be surprised to find parallel proposals or projects being prepared or implemented by others in the academic research community, although we are not aware of any detailed proposals or existing code at the time of writing.

It is quite likely that a functional hierarchy could develop between cooperating projects, something akin to the relationship between FLOSS authors and distributions, such as Debian or Red Hat and their package management systems (*apt* and *rpm*). For example, such an arrangement would allow groups to specialize in collecting and cleaning particular sources of data and others to concentrate on their compatibility. Certainly the existing communities of academics interested in FLOSS, such as <http://opensource.mit.edu>, are encouraged to be a source of data and support. Similarly, we would like to extend to people who donate data the ability to specify a license for that data.

One of the practical problems with spidering projects, like OSSmole, is keeping abreast of changes to the web site (or data source) being spidered. This is a known challenge with any spidering project, and was one of the main motivators for starting this project in the first place: if one research team can worry about spidering, saving, and aggregating the data, then that frees other teams to do other interesting analyses with the data, or to collect new data.

6. CONCLUSION

Researchers study FLOSS projects in order to better understand collaborative human behavior during the process of building software. Yet it is not clear that current researchers have many common frames of reference when they write and speak about the open source phenomenon. As we study open software development we learn the value of openness and accessibility of code and communications; OSSmole is a step towards applying that to academic research on FLOSS. It is our hope that by providing a repository of traceable and comparable data and analyses on FLOSS projects, OSSmole begins to address these difficulties and supports the development of a productive ecosystem of FLOSS research.

7. ACKNOWLEDGMENTS

Our thanks to the members of the ossmole-discuss mailing list, especially Gregorio Robles, Dawid Weiss, and Niti Jain.

8. REFERENCES

- [1] Conklin, M. Do the rich get richer? The impact of power laws on open source development projects. Open Source Convention. (*OSCON '04*) (Portland, Oregon, USA, July 25-30, 2004). At http://www.elon.edu/facstaff/mconklin/pubs/oscon_revised.pdf.
- [2] Crowston, K., Annabi, H., Howison, J., and Masano, C. Towards a portfolio of FLOSS project success metrics. In *Proceedings of the Open Source Workshop of the International Conference on Software Engineering (ICSE '04)*.
- [3] Crowston, K. and Howison, J. The social structure of free and open source software development. *First Monday* 10, 2 (February, 2005).
- [4] El-Emam, K. Ethics and Open Source. In *Empirical Software Engineering* 6, 4 (Dec. 2001), 291-292.
- [5] Howison, J. and Crowston, K. The perils and pitfalls of mining Sourceforge. In *Proceedings of the Workshop on Mining Software Repositories at the International Conference on Software Engineering (ICSE '04)*.
- [6] Krishnamurthy, S. Cave or community? An empirical examination of 100 mature open source projects. *First Monday* 7, 6 (June, 2004).
- [7] Weiss, D. A large crawl and quantitative analysis of open source projects hosted on sourceforge. Research Report ra-001/05, Institute of Computing Science, Pozna University of Technology, Poland, 2005. At <http://www.cs.put.poznan.pl/dweiss/xml/publications/index.xml>
- [8] Xu, J, Gao, Y., Christley, S. and Madey, G. A topological analysis of the open source software development community. In *Proceedings of 38th Hawaii International Conference on System Sciences (HICSS 05)* (Hawaii, USA, January 4-7, 2005).