Proceedings

1st International Workshop on Mining Software Repositories

MSR 2004

Proceedings

1st International Workshop on Mining Software Repositories

MSR 2004

Edinburgh, Scotland, United Kingdom
25th May 2004

Co-located With
International Conference on
Software Engineering
(ICSE 2004)

Edited by
Ahmed E. Hassan, Richard C. Holt, and Audris Mockus

Contents

International Workshop on Mining Software Repositories

MSR 2004

Infrastructure and Extraction

Integration and Presentation

System Understanding and Change Patterns

Defect Analysis

Process and Community Analysis

Software Reuse

Message From Workshop Chairs

MSR 2004

Welcome to MSR 2004, the 1st international workshop on Mining Software Repositories. MSR 2004 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, 26 papers were chosen for publication. Selected papers were chosen for presentation to spur discussion of key concepts. We allocated one hour in the middle of the day where attendees are encouraged to bring in their laptops and present their work/tools to others. We hope this provides interested parties the opportunity to learn more details about other work in the field.

We are delighted that a selected number of papers will be invited for a special issue of the IEEE Transactions on Software Engineering.

We thank Richard van de Stadt for providing round the clock support for the online submission system. We thank Michael Godfrey and Nenad Medvidovic for their prompt replies to our inquires on various workshop organization details. 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

Audris Mockus
Avaya Labs Research

Program Committee

MSR 2004

Harald Gall, *University of Vienna, Austria*
Les Gasser, *University of Illinois at Urbana Champaign, USA*
Daniel German, *University of Victoria, Canada*
James Herbsleb, *Carnegie Mellon University, USA*
Katsuro Inoue, *Osaka University, Japan*
Philip Johnson, *University of Hawaii, USA*
Dewayne Perry, *University of Texas, USA*
Andreas Zeller, *Saarland University, Germany*

Additional Reviewers

MSR 2004

Omar Alonso, *University of California at Davis, USA*
Mohammed El-Ramly, *University of Leicester, UK*
Mike Godfrey, *University of Waterloo, Canada*
Chris Jensen, *University of California, Irvine, USA*
Timothy C. Lethbridge, *University of Ottawa, Canada*
Ying Liu, *University of Alberta, Canada*
Amir Michail, *University of New South Wales, Australia*
Parastoo Mohagheghi, *Ericsson, Norway*
Gabriel Ripoche, *University of Illinois at Urbana Champaign, USA*
Robert J. Sandusky, *University of Illinois at Urbana Champaign, USA*
Jelber Sayyad Shirabad, *University of Ottawa, Canada*
Kevin Schneider, *University of Saskatchewan, Canada*
Elaine J. Weyuker, *AT&T Research, USA*
Jingwei Wu, *University of Waterloo, Canada*
Zhenchang Xing, *University of Alberta, Canada*

MSR 2004: International Workshop on Mining Software Repositories
msr.uwaterloo.ca

9:00-9:15	**Welcome and Introduction** *Ahmed E. Hassan, Richard C. Holt, and Audris Mockus*
9:15-10:30	**Session 1: Infrastructure and Extraction** • Research Infrastructure for Empirical Science of FOSS Les Gasser, Gabriel Ripoche, and Robert Sandusky (University of Illinois at Urbana Champaign) • Preprocessing CVS Data for Fine-Grained Analysis Thomas Zimmermann (Saarland University) and Peter Weißgerber (Catholic University of Eichstätt-Ingolstadt) • **Discussion Leader:** Daniel German (University Of Victoria)
10:30-11:00	**Coffee Break**
10:30-11:15	**Session 2: Integration and Presentation** • Using CVS historical information to understand how students develop software Ying Liu,Eleni Stroulia, Ken Wong (University of Alberta), and Daniel German (University of Victoria) • **Discussion Leader:** Katsuro Inoue (Osaka University)
11:15-12:00	**Session 3: System Understanding and Change Patterns** • Four Interesting Ways in Which History Can Teach Us About Software Michael Godfrey, Cory Kapser, Xinyi Dong, and Lijie Zou (University of Waterloo) • **Discussion Leader:** Annie Ying (IBM T.J. Watson Research Center)
12:30-1:30	**Lunch**
1:30-2:30	**Demos and Walkaround Presentations**
2:30-3:30	**Session 4: Defect Analysis** • Towards Understanding the Rhetoric of Small Changes Ranjith Purushothaman (Dell Computer Corporation) and Dewayne Perry (University of Texas at Austin) • Bug Driven Bug Finders Chadd Williams and Jeff Hollingsworth (University of Maryland) • **Discussion Leader:** Thomas Ostrand (AT&T Labs - Research)
3:30-4:00	**Coffee Break**

4:00-4:30	**Session 5: Process and Community Analysis** • Applying Social Network Analysis to the Information in CVS Repositories Luis Lopez-Fernandez, Gregorio Robles, and Jesus M. Gonzalez-Barahona (Rey Juan Carlos University) • **Discussion Leader:** Chris Jensen (University of California, Irvine)
4:30-5:00	**Session 6: Software Reuse** • A Case Study on Recommending Reusable Software Components using Collaborative Filtering Frank McCarey, Mel Ó Cinnéide, and Nicholas Kushmerick (University College Dublin) • **Discussion Leader:** Pankaj Garg (Zeesource)
5:00-5:30	**Wrap-up: Common Themes and Future Direction** *Ahmed E. Hassan, Richard C. Holt and Audris Mockus*

Infrastructure and Extraction

Preprocessing CVS Data for Fine-Grained Analysis

Thomas Zimmermann
Saarland University, Saarbrücken, Germany
tz@acm.org

Peter Weißgerber
Cath. Univ. of Eichstätt-Ingolstadt, Germany
peter.weissgerber@ku-eichstaett.de

Abstract

All analyses of version archives have one phase in common: the preprocessing of data. Preprocessing has a direct impact on the quality of the results returned by an analysis. In this paper we discuss four essential preprocessing tasks necessary for a fine-grained analysis of CVS archives: (a) data extraction, (b) transaction recovery, (c) mapping of changes to fine-grained entities, and (d) data cleaning. We formalize the concept of sliding time windows and show how commit mails can relate revisions to transactions. We also present two approaches that map changes to the affected building blocks of a file, e.g. functions or sections.

1. Introduction

One of the first papers that analyzed version archives has the striking title "If Your Version Control System Could Talk..."[1]. In these days, many CVS [4] archives are freely available, e.g. via SourceForge.net. They all provide lots of information on the evolution of a software project: *who* changed *what* and *why*.

Such data enables many new analyses. Besides the obvious analysis of software evolution, it is also valuable input for program analysis (e.g. [2, 10, 19]), as well as for metrics (e.g. [3]). All approaches have one thing in common—they have to *preprocess* data, because direct access via CVS clients is rather slow. Additionally, some important information is not accessible via CVS: Which files have been changed in conjunction, and which methods have been affected by a change. The latter is essential for fine-grained analysis of version archives, e.g. on function-level.

In this paper, we focus on four preprocessing tasks that are performed by most analyses:

- *Data Extraction*—In Section 2 we present a lightweight and fast approach to mirror CVS information in a database.
- *Restoring Transactions*—Many analyses require the information which files have been changed in conjunction. In Section 3 we present two approaches that restore such transactions based on sliding time windows and commit mails.
- *Mapping Changes to Entities*—CVS stores only changes on files. For an analysis of functions, changes have to be examined in more detail. Section 4 presents an extensible approach that determines entities affected by a change on a file.
- *Data Cleaning*—Some transactions require special treatments by an analysis: For example, *large transactions* often result from infrastructure changes. *Merge transactions* simply reproduce changes and thus are often noise. Section 5 discusses such topics.

Preprocessing is a prerequisite for a fast access to CVS data. This data is enriched by additional information (transactions, fine-granular changes). Section 6 gives further references to related work, and Section 7 concludes the paper.

2. Data Extraction

One goal for preprocessing is to enable a fast access to the content of a CVS archive. A common solution extracts all data from the CVS repository and mirrors it in a database.

In general, it depends on the analysis what data needs to be extracted. For instance, if we analyze software evolution we are interested in everything, including deleted files. If the purpose of our analysis is to guide programmers along related changes [20], we need only existing files, because suggesting that the user should change deleted files would be awkward. In this case it suffices to extract only a subset of all files stored in the repository. But in practice, the filtering should be performed within the analysis and not within the extraction.

The extraction calls the CVS *log* command in the root directory of the project to be extracted. This returns information on all files that have ever existed in the repository. We parse this output as illustrated in Figure 1 and store the data in appropriate tables:

- Obviously, all *files* and *directories* are stored.

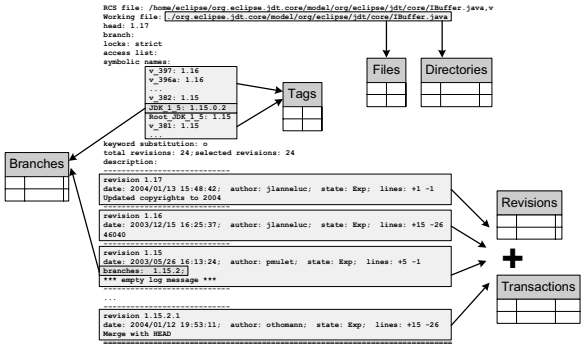
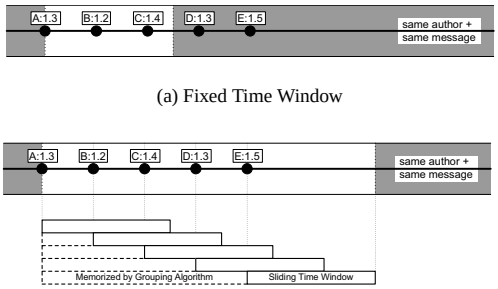


Figure 1. Data Extraction



(a) Fixed Time Window

(b) Sliding Time Window

Figure 2. Fixed vs. Sliding Time Window

- Information about single *revisions* is stored in the table *Revisions*. The author and the log message are stored in the table *Transactions*, because in this step, we treat each revision as one *transaction*—Section 3 groups several revisions/transactions together.
- With CVS the user can set symbolic names for revisions. These symbolic names are called *tags* and are frequently used to mark releases or other events.
- The table *Branches* records the branch points and branch names. This information has to be gathered from two different sections of the CVS *log* output (see Figure 1). Branch names are symbolic names for revision numbers that contain a zero, e.g. *JDK_1_5* for revision number 1.15.0.2. The branch prefix is constructed by removing the zero—in our example it is 1.15.2. The link between the section "symbolic names" and the branch point is established by a hash map using the branch prefix as keys.

Note that all preprocessing steps can also be done *incrementally*—it is only necessary to preprocess the data for new revisions instead of working on the whole repository again. To determine new revisions several approaches exist: Many open-source projects send an email to a mailing list for each commit. This approach is based on the *commitinfo* and *loginfo* files that can be used to track commits on the server-side. A possibility to get recently changed files on the client-side is the CVS *rdiff* operation (with option *-s* for *summary*) or the CVS *status* operation.

3. Restoring Transactions

CVS does not keep track of which files have been changed in conjunction in one commit operation. Often this information is required for an analysis, e.g. for determination of logical coupling [10, 19]. An obvious solution is to consider all changes by the same developer, with the same log message, made at the same time as one *transaction*. The term "same time" is inaccurate in this context, because usually commit operations take several seconds or minutes—especially if many files are involved. In practice, many approaches consider not only checkins at the same time as candidates, but also checkins during a time interval:

Fixed Time Windows restrict the maximal duration of a transaction. The time interval always begins at the *first checkin*. This approach has been used by [15, 10] for the analysis of CVS archives.

Sliding Time Windows restrict the maximal gap between two subsequent checkins of a transaction. The begin of the time interval is shifted to the *most recent checkin*. Thus, this approach can recognize transactions that take longer to complete than the duration of the time window. This approach originates from *ChangeLog* programs like *cvs2cl* [9] or *CVSps* [13].

Figure 2(a) illustrates fixed time windows: After the checkin of A:1.3 both checkins B:1.2 and C:1.4 are part of the same transaction, because they are visible within the time window (drawn in white). Figure 2(b) shows that a sliding time window additionally considers D:1.3 and E:1.5, because the time window "slides" from checkins A:1.3 to finally E:1.5. The transaction is closed after E:1.5 as no further checkins are visible within the time window.

Formally, using a sliding time window of 200 seconds, for all checkins $\delta_1, \ldots, \delta_k$ (sorted by $\mathit{time}(\delta_i)$) that are part of a transaction Δ, the following conditions hold:

$$\forall \delta_i \in \Delta : \mathit{author}(\delta_i) = \mathit{author}(\delta_1)$$
$$\forall \delta_i \in \Delta : \mathit{log\_message}(\delta_i) = \mathit{log\_message}(\delta_1)$$
$$\forall i \in \{2, \ldots, k\} : |\mathit{time}(\delta_i) - \mathit{time}(\delta_{i-1})| \leq \langle 200 \text{ sec} \rangle$$

Additionally, each file can only be part of a single transaction once, because CVS does not allow to commit two revisions of a file at the same time:

$$\forall \delta_a, \delta_b \in \Delta : \ \delta_a \neq \delta_b \Rightarrow \mathit{file}(\delta_a) \neq \mathit{file}(\delta_b)$$

The algorithm for grouping checkins to transactions is straightforward: Simply sort checkins by author, checkin time, and log message. Iterate over checkins in this order: Each time the author or log message differs to the ones of the previous checkin or the time window is exceeded start a new transaction.

Based on our experience, sliding time windows are superior to fixed time windows, because they deal with transactions of any duration. The selection of the length of a time window (fixed or sliding) depends on the analyzed project and the analysis itself. The time window should be chosen based on the assumption on how long it takes to check in the largest file with high network latency. Up to now, most lengths of time windows are arbitrary: They range from two to four minutes.

In our approach we chose 200 seconds which is three minutes plus a buffer of 20 seconds. Without this buffer the end of the time window can clash with the release of a CVS *lock*. In this case the continuation of an interrupted transaction is considered as a new transaction. Using such a time window for the GNU Compiler Collection (GCC), the average duration of a transaction is 6.2 seconds and the maximal duration 1 hour 32 minutes[1].

Time windows are a good approximation for restoring transactions from CVS. A more precise solution is based on *commit mails*—that are mails sent to developer mailing lists after a commit. Such a mail contains the committer, the timestamp, the modified files, and the log message. With this information it is straightforward to relate files to revisions and then to transactions. Commit mails are available for many open-source projects, e.g. GCC.

4. Mapping Changes to Entities

CVS provides only information on files and differences, but not which function has been changed. For an analysis of such fine-grained entities, another preprocessing step is required: Each revision is compared with its predecessor and the changes are mapped to syntactic components of files. If a revision is a merge of multiple predecessors, it should get a special treatment (see Section 5). A revision with no predecessors is compared against an empty file.

Fine-grained changes can be computed using a *diff*-tool and a light-weight analysis that creates the building block of files. This approach is open to everything: source code, documentation, XML files and even diagrams. For a change from revision r_1 to r_2 we compute the entities as follows:

1. Create mappings E_i : *int* $\rightarrow$ *entities* from source code lines to entities using a light-weight analysis (e.g. counting brackets). The mapping for revision r_1 is called E_1 and for r_2 it is E_2.

[1]Transaction "dummy import to prevent merge lossage" (*4081 files*)

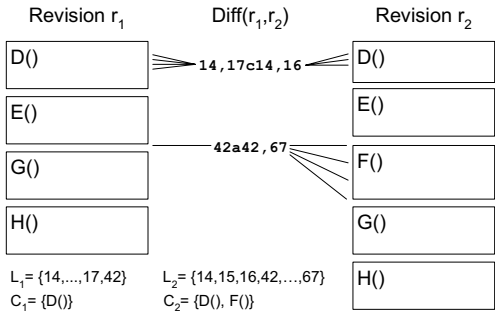


Figure 3. Map Changes to Entities

2. Perform a *diff* between r_1 and r_2. The results are the lines affected by the change: lines L_1 for revision r_1 and L_2 for r_2.

3. The entities (functions, sections) in r_1 affected by the change are C_1 (respectively C_2 for r_2):

$$C_1 = \bigcup_{l \in L_1} E_1(l) \qquad C_2 = \bigcup_{l \in L_2} E_2(l)$$

4. Thus the change from revision r_1 to r_2...
   - actually *changed* entities $C_1 \cap C_2$
   - *added* entities $C_2 \setminus C_1$
   - *removed* entities $C_1 \setminus C_2$

Figure 3 shows an example for the above algorithm. First each revision is decomposed into its building blocks—in our example functions. Then a diff between the two revisions r_1 and r_2 is calculated. The result is used to create the sets $L_1 = \{14, \ldots, 17, 42\}$ and $L_2 = \{14, 15, 16, 42, \ldots, 67\}$. Next, each line is mapped to its enclosing function and the sets $C_1 = \{\mathsf{D()}\}$ and $C_2 = \{\mathsf{D()}, \mathsf{F()}\}$ are created. Now we know that the function D() has been modified ($C_1 \cap C_2$) and F() has been inserted ($C_2 \setminus C_1$).

This approach has two weaknesses: First, its quality depends largely on the precision of the used diff tool, and second, it determines changes only based on lines, rather than on exact source code positions. Thus, in some rare cases this approach recognizes too many changed entities.

A more precise but more expensive approach first determines all entities that occur in both revisions. Then it compares the source codes of each of those entities. In other words, the diff operation is pushed from file-level to entity-level:

1. Determine all entities $\mathcal{E}_1$ of revision r_1 and all entities $\mathcal{E}_2$ of revision r_2.

2. The *added* entities are $\mathcal{E}_2 \setminus \mathcal{E}_1$, and the *removed* entities are $\mathcal{E}_1 \setminus \mathcal{E}_2$.

3. All entities in $\mathcal{E}_1 \cap \mathcal{E}_2$ *may* have been changed. Whether an entity e has been actually changed is decided by performing a diff between the source-code of e in r_1 and its source-code in r_2.

For the example of Figure 3 the above algorithm first determines that the function F() is new, because it appears only in revision r_2. Next, it compares for each function the respective parts and recognizes that D() has been changed.

The ECLIPSE platform [16] provides a powerful and extensible framework for comparing files. Both approaches described above can be realized using this framework:

- *Range Differencer*—The RangeDifferencer[2] class compares two versions based on *tokens*. This approach is based on the traditional *diff* algorithm [14]. The tokens are created using classes implementing the interface ITokenComparator[3], e.g. for lines the class DocLineComparator[4]. The calculated differences are returned in a list.

- *Structure Merge Viewer*—The Differencer[5] class compares two versions of any given *hierarchical structure* and returns a delta tree describing each change in detail. The structure is created with an own implementation of the interface IStructureCreator[6]. The fearless can use existing *internal* classes[7], e.g. the JavaStructureCreator[8].

Furthermore, ECLIPSE provides easy access to JAVA abstract syntax trees and facilitates further analysis of source code. The only drawback is that many of those features cannot be executed from the command line.

5. Data Cleaning

The previous sections described the extraction of data that is needed for fine-grained analysis. However, several issues call for identifying noise and appropriate cleaning (i.e. a special treatment). *Large transactions* which often result from infrastructure changes and *merge transactions* which simply reproduce changes are such noise.

Large Transactions

Large transactions are very frequent in real-life. Here are two examples from OPENSSL:

[2] `compare.rangedifferencer.RangeDifferencer`
[3] `compare.contentmergeviewer.ITokenComparator`
[4] `compare.internal.DocLineComparator`
[5] `compare.structuremergeviewer.Differencer`
[6] `compare.structuremergeviewer.IStructureCreator`
[7] Read [17] before you decide to use internal classes.
[8] `jdt.internal.ui.compare.JavaStructureCreator`

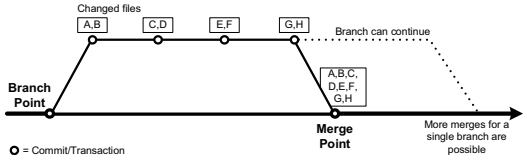


Figure 4. Merges Considered Harmful

- "Change #include filenames from <foo.h> *[sigh]* to <openssl.h>." *(552 files)*

- "Change functions to ANSI C." *(491 files)*

As the log messages indicate, the files contained in these transactions have been changed because of infrastructure changes and not because of logical relations. We refer to such transactions as noise, as it is likely that we will get incorrect results if we use them for any analysis.

A solution is to filter out transactions of size greater N in the analysis. The upper bound N depends on the examined software project.

Merge Transactions

Another more sophisticated kind of noise are merges of branches. CVS simply reproduces all changes made to one branch to the other—in one large transaction. One real-life example taken from GCC is the following:

"mainline merge as of 2003-05-04" *(5874 files)*

Figure 4 shows a smaller example: On the branch four transaction have been committed: $\{A, B\}$, $\{C, D\}$, $\{E, F\}$, and $\{G, H\}$. These files are now again changed at the merge point within a transaction that contains all changes made on the branch: $\{A, B, C, D, E, F, G, H\}$.

Merge transactions are noise for two reasons: First, they contain unrelated changes (e.g. B and C), and second they rank changes on branches higher (because they are duplicated, e.g. A and B). Taking such transactions into account has a significant influence on the results. Thus transactions that resulted from merges have to be identified. Depending on the analysis they should be ignored or at least get some special treatment.

Unfortunately, CVS does not keep track of which revisions resulted from a merge. Michael Fischer et al. proposed a heuristic to detect these revisions [8]. Their approach is restricted to merges to the main branch, but it is straightforward to apply it to other branches. Additionally, they work only on revisions instead of analyzing complete transactions. Analyzing transactions simplifies the detection of merges, because if a merge is detected for a single file, the whole transaction is probably a merge. Nonetheless, automatic merge detection is difficult to realize, because of the large number of existing merge policies. For

example, as Figure 4 indicates the development can continue on both branches after a merge, creating additional complexity for all heuristics.

6. Related Work

Data extraction from CVS is very well covered and many tools are available for free: Daniel German and Audris Mockus created *SoftChange*[9]—a tool that extracts and summarizes information from CVS and bug tracking databases [11]. Dirk Draheim and Lukasz Pekacki developed *Bloof*[10] which extracts CVS log data into a database and visualizes the software evolution using metrics [6].

Michael Fischer et al. demonstrated how to populate a *release history database* linking data from CVS and BUGZILLA [8]. In [7] they also combined their approach with features. Another project that considers additional data sources is *Hipikat* by Davor Čubranić and Gail Murphy [5]. They link information from CVS, BUGZILLA and developer mailing lists using text similarity.

To our knowledge, transaction recovery has been used by many approaches but has nowhere been covered in detail: Harald Gall, Daniel German, and Audris Mockus used fixed time windows in the past [10, 11, 15], and we used sliding time windows in our previous work [19, 20]. Commit mails have not been used in recent work to restore transactions.

Up to now, only a few approaches have considered fine-grained changes: Harald Gall et al. [10] and James Bieman et al. [3] both analyzed relations between classes. In our previous work we applied the approach presented in Section 4 and mined for relations [19] and association rules [20] between functions, sections and other fine-grained building blocks.

Michael Fischer et al. also proposed an algorithm for detecting merges of revisions in their release history database paper [8]. Lijie Zou and Michael Godfrey showed how to use origin analysis to detect merging and splitting of functions in [21]. Nonetheless, data cleaning is often neglected and there is still much room for improvement.

7. Conclusion

CVS archives contain lots of information—which is usually accessible via clients. This data provides a basis for analyses that mine additional knowledge. But CVS has some weaknesses: it is slow and loses information on transactions, fine-grained changes and merges. Thus, a preprocessing step is required.

This paper is a first attempt to collect and formalize preprocessing tasks that are used by analyses of version archives. We hope that it facilitates upcoming research in this area and provides a fruitful base for further discussions.

Acknowledgments. This project is funded by the Deutsche Forschungsgemeinschaft, grant Ze 509/1-1. Stephan Diehl, Richard Kuntschke, Andreas Zeller and the anonymous MSR reviewers gave helpful comments on earlier revisions of this paper.

[9] http://sourcechange.sourceforge.net

[10] http://bloof.sourceforge.net

References

[1] T. Ball, J.-M. Kim, A. A. Porter, and H. P. Siy. If your version control system could talk. . . . In *ICSE Workshop on Process Modelling and Empirical Studies of Software Engineering*, 1997.

[2] J. Bevan and J. Whitehead. Identification of software instabilities. In WCRE 2003 [18], pages 134–143.

[3] J. M. Bieman, A. A. Andrews, and H. J. Yang. Understanding change-proneness in OO software through visualization. In *Proc. 11th International Workshop on Program Comprehension*, pages 44–53, Portland, Oregon, May 2003.

[4] P. Cederqvist. *Version Management with CVS*, Dec. 2003. www.cvshome.org/docs/manual/.

[5] 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.

[6] D. Draheim and L. Pekacki. Process-centric analytical processing of version control data. In IWPSE 2003 [12].

[7] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In WCRE 2003 [18].

[8] 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.

[9] K. Fogel and M. O'Neill. *cvs2cl.pl: CVS-log-message-to-ChangeLog conversion script*, Sept. 2002. http://www.red-bean.com/cvs2cl/.

[10] H. Gall, M. Jazayeri, and J. Krajewski. CVS release history data for detecting logical couplings. In IWPSE 2003 [12], pages 13–23.

[11] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of ICSE '03 Workshop on Open Source Software Engineering*, Portland, Oregon, USA, May 2003.

[12] *Proc. International Workshop on Principles of Software Evolution (IWPSE 2003)*, Helsinki, Finland, Sept. 2003. IEEE Press.

[13] D. Mansfield. *CVSps – Patchsets for CVS*, Feb. 2004. http://www.cobite.com/cvsps/.

[14] W. Miller and E. W. Myers. A file comparison program. *Software—Practice and Experience*, 15(11):1025–1040, Nov. 1985.

[15] 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.

[16] Object Technology International. *Eclipse Platform Technical Overview*, Feb. 2003. Available at www.eclipse.org.

[17] J. des Rivières. *How to use the Eclipse API*, May 2001. http://eclipse.org/articles/Article-API%20use/eclipse-api-usage-rules.html.

[18] *Proc. 10th Working Conference on Reverse Engineering (WCRE 2003)*, Victoria, British Columbia, Canada, Nov. 2003. IEEE.

[19] T. Zimmermann, S. Diehl, and A. Zeller. How history justifies system architecture (or not). In IWPSE 2003 [12], pages 73–83.

[20] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proc. 26th International Conference on Software Engineering (ICSE)*, Edinburgh, Scotland, May 2004.

[21] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In WCRE 2003 [18].

The perils and pitfalls of mining SourceForge

James Howison and Kevin Crowston
Syracuse University School of Information Studies
4-206 SciTech, Syracuse, New York, USA
{jhowison, crowston}@syr.edu http://floss.syr.edu

Abstract

SourceForge provides abundant accessible data from Open Source Software development projects, making it an attractive data source for software engineering research. However it is not without theoretical peril and practical pitfalls. In this paper, we outline practical lessons gained from our spidering, parsing and analysis of SourceForge data.

SourceForge can be practically difficult: projects are defunct, data from earlier systems has been dumped in and crucial data is hosted outside SourceForge, dirtying the retrieved data. These practical issues play directly into analysis: decisions made in screening projects can reduce the range of variables, skewing data and biasing correlations.

SourceForge is theoretically perilous: because it provides easily accessible data items for each project, tempting researchers to fit their theories to these limited data. Worse, few are plausible dependent variables. Studies are thus likely to test the same hypotheses even if they start from different theoretical bases. To avoid these problems, analyses of SourceForge projects should go beyond project level variables and carefully consider which variables are used for screening projects and which for testing hypotheses.

1 Introduction

We are interested in identifying factors that predict the performance of Free, Libre and Open Source Software (FLOSS) teams. As part of this inquiry, we chose to analyze data from the SourceForge website, the largest repository of FLOSS project data and, as such, an excellent source of data on FLOSS team practices [9, 1].

While Data mining is the process both of data collection and data analysis this paper focuses only on the challenges faced in our first steps of data collection. Our data collection process involved spidering numerous Web pages, parsing the downloaded HTML files and producing summary data for analysis. Our research encountered a number of practical pitfalls that this paper outlines. Yet the difficulties are not merely practical—in our investigation of SourceForge data and in other papers dealing with SourceForge data we have encountered a number of theoretical caveats that we present here.

1.1 Research background

Our interest in FLOSS stems from a broader interest in distributed teams. The focus of our research is on team practices: coordination, development of collective mind and individual and organizational learning. Therefore we intend to examine a number of projects in detail with both qualitative content analysis and ethnographic methodology. However given the sheer number of projects and the volumes of data available, a prerequisite to our research is to identify appropriate projects for detailed study. We are seeking both successful and unsuccessful projects using the model of FLOSS project success we explored in [3].

We collected data from the project demographics, developer mailing lists and the SourceForge Tracker system—which is largely bug tracking data. With this data we conducted social network analysis to identify variance in communication structure (Results reported in [5]), process analysis of bug fixing (Reported in [10]) and an analysis of the speed with which projects fix bugs (Preliminary analysis reported in [4]).

2 Avoiding Pitfalls in Data collection

After receiving no response to our requests for direct access to the SourceForge database, we concluded that the best available method of data collection would be to spider and "screen-scrape" the data. We initially spidered the SourceForge project pages in April 2002 and used this data to identify 140 projects that had greater than seven listed developers and more than 100 bugs in the system. These criteria were theoretically chosen to match our interest in distributed teams and the bug-fixing process. We spidered the mailing lists and bug-tracking pages in April 2003, accessing data on over 62,110 bug reports.

There were three stages in our data collection: Spidering, Parsing and Summarizing [1]. Each presented its own practical difficulties and necessary choices. We outline these, and our solutions, below. We utilized Perl scripts for the data collection—some comments are specific to Perl, most are not. We conclude this section with testing strategies that we recommend for mining software repositories.

2.1 Avoiding Pitfalls in Spidering

Our spidering scripts utilized the `WWW::Mechanize` module available from `CPAN` [2]. The unfortunate necessity of spidering large datasets can place large strains on the servers. It is therefore important to be well behaved both during the development of your scripts and in their use.

- Code a `-n` (`--do-nothing` or 'dry-run') option to test your scripts.
- Consider running a local test server that mirrors the structure of your target site.
- Store the full HTML download rather than parsing 'live'. This ensures that any changes to the parser (expect many!) will not require a 're-spidering' of your target site.
- It is tempting to use forked processes to speed up spidering. But beware of forking too many processes and especially of 'lost children' who, due to a bug, might 'bang away' at the server for days—long after the parent process is killed.
- Code a 'wait loop' into the spidering code to reduce the density of your page requests. At the time we spidered we were never banned from the SourceForge servers, although we have recently heard that others have been. It is believed that this is a new defence introduced by SourceForge, the parameters of which are unknown. The spidering process can take a long time, extending over a number of days. It is therefore crucial to be sure to collect all relevant data at the time of collection.
- Whenever feasible prepare your analysis scripts and test them on data spidered from one or two projects, ensuring that the data being collected is sufficient. Repeating the spidering stage can be very time consuming.
- Be sure to store the time at which the page was downloaded. This is especially important for time-dependent analysis that has to account for censored data (such as Event History analysis) but equally it is required to anchor the effective date of your analysis for comparative and longitudinal analyses.
- It is useful to store the number of pages of each type downloaded, which gives a count of the expected number of Bugs that should be found after parsing. This count can be used as a test for the accuracy of your parsing scripts.

Spidering is clearly an area in which cooperation between research groups could present great benefits. It is also vital to ensure that the SourceForge site is operating properly at the time you spider—this can be checked through the Site Status page [3].

2.2 Avoiding Pitfalls in Parsing

Large websites are generated from HTML templates and databases, giving them a fairly consistent structure suitable for scripted parsing to extract the required data for analysis. Yet the level of consistency is not high enough to ensure that unexpected problems will not be faced.

- Simplify your parsing process as much as possible by reducing excess or non-standard HTML on the pages. Test the results of utilizing the `HTML::Tidy` module or W3Cs 'tidy' application which does a good job of standardizing the HTML and removing 'cruft'. However be sure to check that this has not altered your target data in any way and that its effects are consistent across your downloaded dataset.
- While regular expressions are vital to this type of parsing, we found it far simpler to utilize them in combination with HTML parsing utilities such as `HTML::Parser::Simple` and `HTML::TableExtractor`.

Many of the inconsistencies encountered were contained only in a limited number of projects or even only within a few bugs or mailing lists, yet they can significantly undermine your confidence in the cleanliness of your data. We found these to be important points to be aware of in the SourceForge data:

- Line breaks in fields are especially tough to observe in regular debugging output. Consider converting linebreaks to visible characters to avoid confusion.
- Unexpected characters in fields, such as non-ascii characters or HTML entities. These often show-up as errors in external modules being utilized making the situation difficult to debug

[1] Our analysis scripts are available on request from the first author

[2] the Comprehensive Perl Archive Network—a 'class-library' for Perl.

[3] `http://sourceforge.net/docman/display_doc.php?docid=2352&group_id=1#1076697351`

- Another very frustrating bug was caused by usernames that look like html (Thanks, `<DeXtEr>` ! (gaim/482924)). Our Perl regex to parse the fields of username and Real Name was `/(.*)\s*\((.*)\)/`.
- The layout of the information on status changes in SourceForge was inconsistent in the table at the bottom of the bug. Three separate methods had to be used to find the correct `close_date`. It appears that SourceForge has now changed this layout.

Many projects are inconsistent in their use of the SourceForge system. Be especially aware of projects that have moved old data into the SourceForge system (e.g, tcl). The 'official' fields may contain misleading data (e.g., a Start Date reflecting the day of re-entry). While the free-text fields may contain the data from the old system in parsable form, researchers need to decide whether to write a special case parser for this data or to drop the project from the analysis. Also be aware that the SourceForge Tracker stores interaction information for each Item as 'follow-up messages' in free-text fields that are of arbitrary length and have inconsistent endings as well as containing unexpected characters. See `dynapi` patch 207106 for a tricky example. Our intention was to use XML for data storage between scripts. Beware though: `XML::Simple` cannot read all that it can write! We successfully used `Storable` (Perl data-structure serialization module) to store and pass the data between modules.

2.3 Avoiding Pitfalls in Summarizing

Summarization requirements will vary according to the intended analyses. We pursued Social Network Analysis (SNA), which required interaction matrices, and event history analysis, which required data on lifetimes, bug status and assignment.

One problem in summarizing is missing data. For example, SourceForge allows users to post anonymously, giving such posts the username of "nobody". Since we couldn't predict the effect that different treatments of the "nobody" data would have on our analysis, we created a summarizer that produced each of four treatments for "nobody data" 1. Baseline case: No treatment, "nobody" appears as an individual. 2. Deletion case: All nobody data deleted. 3. Each "nobody" as separate individual. 4. One "nobody" per Item or thread i.e. "nobody340078" as separate individual. We were then able to compare the effect that these different treatments had on the outcome of our analysis (we found surprisingly little difference between the last two strategies) [5].

An opposite problem is that a number of the fields of interest turn out to be multi-valued. For example, a project can be given a development status of `planning`, `alpha` and `beta` simultaneously. To retain these multiple values would make analysis very complex. Researcher ought to make principled decisions about how to handle such cases, rather than letting them be made for convenience in parsing or summarizing.

A final problem is that different analysis tools will require different data output formats. We tested over 5 different Social Network analysis packages before settling on the `sna` module from `r-project` for its high degree of scriptability, vital for large data sets. We also used the `NetMiner` application for its presentable graphics capabilities. Each program required output in subtly different formats. We found that our summarization methods were being used in a number of different scripts, making summarization an excellent candidate for modularization.

2.4 Testing Strategies

On reflection, our testing strategy should have been considerably more systematic. We would recommend these techniques to those pursuing large data-collection projects involving spidering and parsing:

- Random selection of test pages (at least three from each project) that should be checked by hand to create known good output.
- When making changes to the parser or summarizer, preserve the output of earlier runs to check for unexpected regressions. One strategy would be to `diff` old and new results, noting the items whose values have changed and check that against the intended and anticipated changes.
- Remember to note in your comments the bug reports for which special cases of code are written (or alternations in regular expressions). It is surprising how quickly these are forgotten in the flow of bug-fixing.
- Further, it would be useful to create test cases for each quirk identified, both to ensure the correctness of a proposed solution and to prevent regressions.
- Test cases could be shared with others seeking to parse similar data from the repository of interest (e.g., we could have a location to share test cases for SourceForge, CVS, Bugzilla, Subversion, mailing lists etc.)

3 Interpretation and Analysis

There are several important issues to consider when undertaking analysis and interpretation of SourceForge data. Those seeking to utilize this data must carefully consider their choice of screening variables and keep these separate from their analysis variables.

3.1 Challenges in cleaning dirty data

Despite the template and database nature of the SourceForge website there is a significant amount of 'dirty' data and it is hard to be sure of the extent of these problems without time-consuming and costly manual checking.

As described above, there is a large amount of anonymous data in the SourceForge system that cannot be attributed to any individual participant. For some analyses this will not have an impact but it could be crucial for others. Also described above, there is data that has been 'dumped' into the system, yielding valid yet totally inaccurate data.

Furthermore SourceForge has become the 'repository of record' for the FLOSS community, yet for important projects it is not the 'repository of use'. For example `vim`, an important programmers editor, is listed at SourceForge but has only 3 developers and 0% activity and has not released any files—all clearly wrong. The page is simply a placeholder that points to the `vim` 'repository of use'. It is likely that there are many entries like this and identifying them is difficult, at the very least it requires the use of a data source outside of SourceForge.

3.2 Skewed and truncated data

Firstly the projects in SourceForge are of highly different shapes, sizes and structures, which leads to much of the data being highly skewed. For example our screening conditions (> 7 developers and > 100 bugs) reduced the projects of interest from the 52,000 hosted by SourceForge at the time of spidering to only 140 projects. This skew extends to project activity and is reported in the findings of [7]. It appears that there are a very large number of one-person projects entered into SourceForge that never progress beyond announcement [9].

The problems above, and their possible solutions, may yield truncated results which complicate variance analyses, such as regressions. When it is necessary to choose screening variables to reduce the dataset to the projects of theoretical interest, the analysis must acknowledge that the variance in those screening variables has been significantly reduced and attempt to compensate for this reduction (or better still avoid the further use of the use of that variable at all).

Even avoiding the use of screening variables may not be sufficient, because when a dataset is reduced by screening on one variable there may be significant truncation of correlated variables. Even worse, it is difficult to know this in advance without collecting all the data to look for these correlations.

These difficulties are further compounded by the difficulty in predicting the direction of the impact of the truncation. For example truncating a variable that is highly varied may result in a variable that appears to be less varied, increasing correlations. But truncating a variable that exhibits low variance can cause the variable to increase its relative variance, leading to reduced correlations (Figure 1). Equally a decision to truncate only sections of a variable (either the top and bottom, or the mid-range) can have quite unpredictable effects (in Figure 2 substantially increasing the correlation).

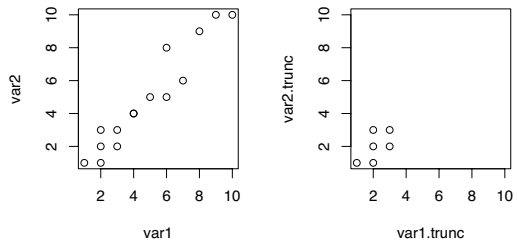


Figure 1. Truncating the upper range of var1 reduces the correlation from 0.91 to 0.52

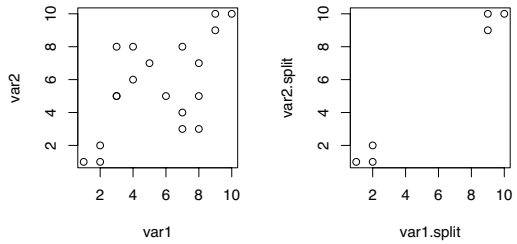


Figure 2. Splitting the distribution increases the correlation from 0.57 to 0.99

4 Peril in Research Design

SourceForge is a highly available dataset, but it provides only a limited number of easily available variables, that is, variables that are pre-calculated and available from each project's homepage or in full lists. Examples include: Number of developers, Project status, Activity, Downloads, Page Views and Numbers of Tracker items. The restricted amount of data poses problems in research design.

One problem is that there are only a limited number of ways of mapping theoretical constructs on to these available variables. As a result, studies using SourceForge data may end up with similar regression equations while purporting to study quite divergent concepts. For example, Crowston and Scozzi present an analysis of OSS communities as virtual organizations by applying Katzy and Crowston's competency rallying theory to the case of OSS development

projects [8, 6]. The theory explains project success in terms of the availability of competencies, the ability of developers to recognize opportunities, the ability of project to marshall resources and the ability to manage short-term cooperation, but end up regressing downloads, development status and activity against development status, number of administrators, popularity of the programming language, number of developers, lifespan, activity and factors for audience and topic (omitting as necessary the dependent variables as independent variables). Chengalur-Smith and Sidorova start with a population ecology perspective on projects, but propose to regress project survival against number of releases, number of development statuses, audiences and topics, age and number of developers [2].

A second problem is that these available variables may have low validity as measures in any particular theory. For example, many studies have chosen to use downloads as their dependent variable, arguing that it is a plausible proxy for "use". This is problematic for two reasons. Firstly, in the Information Systems literature "use" is already largely a proxy for "impact" [11], so downloads becomes a proxy for a proxy. Secondly, downloads is not even a good proxy for use, being both inaccurate and systematically biased. Much fundamental FLOSS software is distributed through distributions (e.g., RedHat CDs or Debian's `apt-get` system and FreeBSD's `ports`) and therefore not often downloaded from SourceForge (how often has a user downloaded the `vim` program or the `Xfree86` distribution directly from SourceForge?). Conversely userland packages facing frequent changes in their environment (e.g., `Gaim`, an instant messenger package), or new packages not yet included in a distribution, would have higher downloads. We deal with similar difficulties with alternative dependent variable measures and develop, hopefully, a more useful approach to FLOSS project success in [4, 3].

5 Conclusion

SourceForge remains an excellent source of data for those interested in studying the processes of FLOSS teams and distributed teams in general—one of many such repositories. Screen-scraping remains an unfortunate necessity faced by researchers seeking to mine online repositories. We have presented our experiences in mining SourceForge, and made available our code. We have also sought to highlight the general lessons for mining software repositories.

Regardless of data collection method those wishing to use sourceforge data face significant challenges in cleaning, screening and interpreting the data, we have outlined those we have identified and the solutions we employ: researchers should be tuned to the impact of their screening and attempt to minimize the impact of that screening on their analyses.

Finally, as a discipline, we must be conscious of the limitations of the 'ready-made' data-points available through repositories such as SourceForge. Researchers must take care in operationalizing their theoretical constructs and should be prepared to go well beyond the "low hanging fruit".

Once these challenges have been faced in gathering data, SourceForge can produce useful research results. In [5] social network analysis showed that FLOSS projects vary widely in their communications centralization and our correlations demonstrated that larger projects decentralize into a 'shallot' shaped structure. In [4] our event history analysis of bug fixing produced an intriguing measure of team performance which is sharply differentiated from other measures of project success. In [10] we mapped clear coordination practices in the bug fixing process. We are continuing to explore this intriguing dataset.

References

[1] A. Capiluppi, P. Lago, and M. Morisio. Evidences in the evolution of os projects through changelog analyses. In *Proceedings of the 3nd Workshop on Open Source Software Engineering, Int. Conf. Software Engineering*, 2003.

[2] S. Chengalur-Smith and A. Sidorova. Survival of open-source projects: A population ecology perspective. In *Proc. of 24th International Conference on Information Systems (ICIS '03*, Seattle, WA., 2003.

[3] K. Crowston, H. Annabi, and J. Howison. Defining open source software project success. In *Proc. of International Conference on Information Systems (ICIS)*, 2003.

[4] K. Crowston, H. Annabi, J. Howison, and C. Masano. Towards a portfolio of FLOSS project sucess measures. In *ICSE Open Source Workshop*, 2004.

[5] K. Crowston and J. Howison. The social structure of open source software development teams. In *OASIS 2003 Workshop (IFIP 8.2 WG)*, 2003.

[6] K. Crowston and B. Scozzi. Open source software projects as virtual organizations: Competency rallying for software development. *IEE Proceedings on Software*, 149(1):3–17, 2002.

[7] R. A. Ghosh, G. Robles, and R. Glott. Free/libre and open source software: Survey and study floss. Technical report, International Institute of Infonomics,, University of Maastricht: Netherlands, 2002.

[8] B. R. Katzy and K. Crowston. A process theory of competency rallying in engineering projects. In *Proc. of CeTIM*, Munich: Germany, 2000.

[9] S. Krishnamurthy. Cave or community?: An empirical examination of 100 mature open source projects. *First Monday*, 7(6), June 2002.

[10] B. Scozzi and K. Crowston. Coordination practices for bug fixing within FLOSS development teams. In *First International Workshop on Computer Supported Activity Coordination (CSAC 2004)*, Porto (Portugal), 2004.

[11] P. B. Seddon. A respecification and extension of the delone and mclean model of is success. *Information Systems Research*, 8(3):240–253, 1997.

Research Infrastructure for Empirical Science of F/OSS

Les Gasser Gabriel Ripoche Robert J. Sandusky
Graduate School of Library and Information Science
University of Illinois at Urbana-Champaign
{gasser,gripoche,sandusky}@uiuc.edu

Abstract

F/OSS research faces a new and unusual situation: the traditional difficulties of gathering enough empirical data have been replaced by issues of dealing with enormous amounts of freely available data from many disparate sources (forums, code, bug reports, etc.) At present no means exist for assembling these data under common access points and frameworks for comparative, longitudinal, and collaborative research. Gathering and maintaining large F/OSS data collections reliably and making them usable present several research challenges. For example, current projects usually rely on "web scraping" or on direct access to raw data from groups that generate it, and both of these methods require unique effort for each new corpus, or even for updating existing corpora. In this paper we identify several common needs and critical factors in F/OSS empirical research, and suggest orientations and recommendations for the design of a shared research infrastructure.

1. Introduction

A significant group of software researchers is beginning to investigate large software projects empirically, using freely available data from F/OSS projects. A body of recent work, along with targeted assessments of researchers in the field, point out the pressing need for community-wide data collections and research infrastructure to expand the depth and breadth of empirical F/OSS research, and several rough proposals have been made [3].

This paper attempts to justify and clarify the need for community-wide, sharable research infrastructure and collections of data. We review the general case for empirical research on software repositories, articulate some specific current barriers to this empirical research approach, and sketch several community-wide options with the potential to address some of the most critical barriers. First, we review the range of research and research questions that could benefit from a research infrastructure and data collections. Second, we expose critical requirements of such a project. We then suggest a set of components that address these requirements, and put forth several specific recommendations.

2. Objects of Study and Research Questions

As an organizing framework, we identify four main *objects of study*—that is, things whose characteristics researchers are trying to describe and explain—in F/OSS-based empirical software research: *software artifacts*, *software processes*, *development communities*, and *participants' knowledge*. In Table 1 we provide a rough map of some representative characteristics that have been investigated for each of these objects of study, and show some critical factors that researchers have begun linking to these characteristics as explanations. It is important to point out that these objects of study are by no means independent from one another. They should be considered as interdependent elements of F/OSS (e.g., knowledge and processes affect artifacts, communities affect processes, etc.) Also, each of the outcomes shown in Table 1 may play a role as a critical factor in the other categories.

3. Current Research Approaches

We have identified two major approaches in empirical research on the objects and factors in Table 1:

- Large-scale cross-analyses of project and artifact characteristics, such as code size and code change evolution, development group size, composition and organization, or development processes [4, 5, 7].
- Smaller-scale case studies of specific practices and processes, for concept/hypothesis development and exposing mechanism details [1, 10].

These two orientations are separated less by fundamental differences in objectives than by technical limitations in

Objects	Success Measures	Critical Driving Factors
Artifacts	Quality, reliability, usability, durability, fit	Size, complexity, software architecture (structure, substrates, infrastructure)
Processes	Time, cost, complexity, manageability, predictability	Size, distribution, collaboration, knowledge/information management, artifact structure
Communities	Ease of creation, sustainability, trust, social capital	Size, economic setting, organizational architecture, behaviors, incentive structures
Knowledge	Creation, use, need, management	Tools, conventions, norms, social structures, technical content

Table 1. Characteristics of empirical F/OSS studies.

existing tools and methods. For example, qualitative analyses are hard to implement on a large scale, and quantitative methods have to rely on uniform, easily processable data. We believe these distinctions are becoming increasingly blurred as researchers develop and use more sophisticated analysis and modeling tools [9], leading to finer gradations in empirical data needs.

4. Essential Characteristics

Empirical studies of software artifacts, processes, communities and knowledge are constrained by several key requirements. They should:

1. *Reflect reality* from actual experience rather than assumed, artificially constructed phenomena.
2. Give *adequate coverage* of naturally-occurring phenomena.
3. Examine *representative levels of variance* in key dimensions and phenomena.
4. Demonstrate *adequate statistical significance*.
5. Provide results that are *comparable across projects*.
6. Provide results that can be *repeated, tested, evaluated, and extended* by others.

Taken together, these six requirements for software research drive several requirements on the infrastructure and data for that research. For example:

- To satisfy the needs for reality and coverage (1,2), data should be *empirical and natural*, from real projects.
- For coverage of phenomena, demonstration of variance, and statistical significance (2,3,4), data should be *available in collections of sufficient size*.
- To allow for comparability across projects, and to allow community-wide testing, evaluation, and extension of findings (5,6), data and findings should be *sharable, in common frameworks and representations*.

5. Available Empirical Data

F/OSS researchers have access to very large quantities and varieties of data, as most of the activity of F/OSS groups is carried on through persistent electronic media whose contents are open and freely available. The variety of data is manifested in several ways.

First, data vary in *content*, with types such as communications (threaded discussions, chats, digests), documentation (user and developer documentation, HOWTOs, FAQs, tutorials), and development data (source code, bug reports, design documents).

Second, data originates from different *media* sources, such as communication systems, version control systems, issue tracking systems, and content management systems.

Third, data can be found from various *locations*, such as community websites, software repositories and indexes, and individual project sites.

Most F/OSS project data is available as *byproducts* of development, maintenance, and system-use activities in F/OSS communities. Very little data is directly available in forms specifically intended for research use. This byproduct origin has several implications for the needs expressed above.

6. Issues with Empirical Data

Many steps often have to be performed to identify, gather, and prepare data before it can be used for research. Data identification and preparation are important aspects of the research process and help guarantee that the six essential characteristics described above are met. The following steps are common barriers that most empirical F/OSS researcher will have to address:

Discovery and Selection

Because so much data is available, and because such a diversity exists in data formats and media, finding and selecting pertinent, usable data to study can be difficult. This is a general Resource Description/Discovery (RDD) and information retrieval issue, appearing here in the context of

scientific data. Appropriate information organization and metadata principles should ideally be employed in the original sources, but this is rare in F/OSS (and other software) data repositories, in part because of the byproduct nature of F/OSS research data.

Access and Gathering

By access we mean the actually obtaining useful data once it has been discovered and selected. Access difficulties include managing administrative access to data, actually procuring data (e.g., overcoming bandwidth constraints) and dealing with difficulties transforming data in a useful format (such as a repository snapshot or via web scraping).

Cleaning and Normalization

Because of the diversity of research questions, styles, methods, and tools, and the diversity of data sources and media available, researchers face several types of difficulty with raw data: original data formats may not match research needs; data of different types, from different sources or projects, may not be integrable in its original forms; and data formats or media may not match those required by qualitative or quantitative data analysis tools. In these cases, research data has to be normalized before it can be used. Data normalization activities may include data format changes, integration of representation schemas, transformations of basic measurement units, and even pre-computation and derivation of higher-order data values from base data. Normalization issues appear at the level of individual data items and at the level data collections.

Linked Aggregation

Normalized data is critical for cross-source comparison and mining over data "joins". However, some F/OSS-based research projects are exploring structural links and inferential relationships between data of very different characters, such as linking social network patterns to code structure patterns, or linking bug report relationships to forms of social order [10]. Linked data aggregation demands invention of new representational concepts specific to the kinds of data links desired for projects, and transformations of base data into forms compatible with those links.

Evolution

Real projects continually evolve, both in content and in format: web sites are redesigned, tools are modified, etc. Research projects may have to track, adapt to, and reflect these changes. This can cause problems at many of the previous levels, as access rights can be modified, formats can change and links can be created or removed. In addition, trajectories of evolution themselves are actually an important object of study for some empirical software researchers. The central issue for this paper is how to adhere to the essential characteristics given above (such as the needs for testable, repeatable, and comparable results) while reacting to and/or managing this evolution.

7. Addressing These Issues

The main objective of a research infrastructure is to address community-wide resource issues in community-specific way [13]. For F/OSS research, the objective is to improve the collective productivity of software research by lowering the access cost and effort for data that will address the critical questions of software development research. In this section we offer some possible approaches to such an infrastructure, by first briefly describing each "component", and then considering its benefits and drawbacks.

Representation Standards

One of the broadest approaches to common infrastructure is the use of representation standards [13]. Such standards would move some issues of cross-source data normalization forward in the process that produces F/OSS projects' information. For example, standard internal formats for objects such as bug reports could eliminate many internal differences between Bugzilla, Scarab, Gnats, etc., fostering simpler cross-analysis of data from these various repositories. Such representation standards would also facilitate exchange of data and/or processing tools within the F/OSS research community. For example, as part of our investigation of F/OSS bug reporting/resolution processes [9], we developed a general XML schematization of bug reports, derived from (but more general than) the Bugzilla internal database schema, and designed as a normalization target and translation medium for multiple types of bug reports from different systems [8]. Issues include the difficulty of developing, promulgating, maintaining, and enforcing such standards.

Metadata

The use of metadata permits researchers to identify relevant characteristics of specific data collections. Metadata can serve numerous roles in the organization and access of scientific data and documents, including roles in location, identification, security/access control, preservation, and collocation [12]. Standardization of metadata and addition of metadata to F/OSS information repositories, especially at the point of creation, would let the research community identify much more easily the data used in each study, understand and compare data formats, and would also simplify the selection process, by making visible critical selection information. Fortunately, some metadata creation can be automated; unfortunately, representation standards are also an issue for metadata.

Tools

Tools could potentially be developed to address each of the issues reviewed in the previous section. Some such tools

already partially exist in a generic form or are developed as needed by research groups. Tools such as web-scrapers that gather data, entity extractors that mine for specific entities like people and dates, or cross-references that link multiple information sources of a single project are commonly developed from scratch in each research effort. These tools are part of the basic toolbox of almost every empirical F/OSS researcher and could easily be provided as such. In fact, several nascent efforts are already underway to produce such tools (e.g. [6]).

Another contribution of a research infrastructure could be to place research data access and manipulation tools upstream, directly within software development tools used by the F/OSS community (e.g., CVS, Subversion, Bugzilla), instead of requiring sometimes-tedious and potentially risky post processing. For example, in most cases, F/OSS tools rely on databases for data storage and manipulation. These databases contain valuable information that is often lost during the translation to a web-visible front-end. (Usually the front-ends rely on web interfaces that display information in a user-friendly fashion but drop important structure in the process). Access to the underlying database can be much more valuable (and in many cases easier) than the current techniques of web-scraping that must recreate such missing relations post-hoc, and may not be successful.

Centralized Data Repositories (CDRs)

Gathering specific snapshots of raw data and making them available to the research community from a controlled "cleanroom" location could provide benchmark data for comparative analyses and measurement of progress—a type of infrastructure that has proven invaluable in other disciplines. It would ensure that data parameters stay constant across studies, and through evolutionary stages of projects. Moreover, it might be easier in many cases to get a snapshot from such a repository than to go through all the steps of collecting the data directly from an F/OSS community. The CDR approach can have advantages of control, organization, and data persistence. However, this approach also raises the issues of *data selection* and *maintenance*. As with any managed information collection, CDRs would need *selection policies* to detail which materials from projects, tools and communities would be chosen for inclusion, and why [2]. The F/OSS community is already too large to attempt building practical evolving archives of *all* F/OSS projects (if such a notion were even meaningful). Selection necessarily induces bias, but careful selection would foster research on a shared body of data, possibly leading to more reliable findings. Second, *preservation policies* need development as F/OSS data is evolving quickly and collections will have to be maintained.

Federated Access

Federating access is another approach to facilitating information sharing without making many redundant copies of original data, while maintaining local control over data access and organization. A central federation repository collects only metadata, and uses it to provide common-framework access to a variety of underlying sources. Federation has the advantages of distributed sharing, such as trading off lightweight central representations and sophisticated search infrastructure, against local data maintenance, information preservation, and access control.

Processed Research Collections

Putting all the previous components together would lead to a set of normalized, processed and integrated collections of F/OSS data made available to the research community through either federated or centralized mechanisms.

Integrated Data-to-Literature Environments

Finally, an advanced contemporary approach would be an attempt to connect both data sources and research literature in a seamless and interlocking web, so that research findings can be traced back to sources, and so that basic source data can be linked directly to inferences made from it. Such arrangements provide powerful intrinsic means of discovering connections among research themes and ideas, as they are linked through both citation, through common or related uses of underlying data, and through associations among concepts. Similar efforts are underway in many other sciences (e.g. [11, 13]). Networks of literature and data created in this way, with automated support, can reduce cognitive complexity, establish collocation of concepts and findings, and establish/maintain social organization within and across F/OSS projects.

8. Recommendations

In accord with the rationales outlined above and the strong sense of the F/OSS community [3], we recommend that F/OSS researchers begin collective efforts to create sharable infrastructure for collaborative empirical research. This infrastructure should be assembled incrementally, with activity in many of the areas defined below:

Refine Knowledge

This paper has provided a sketch of some ideas toward robust and useful research infrastructure. The ideas and motivations here need more development, and collaborative efforts are encouraged.

Exploit Experience

Many standards for sharable scientific data exist for other communities, as do many repositories of data conforming to those standards. We should do further research on what

other communities have done to organize research data. For example, many collections of social science data are maintained around the world[1]. We should use the experiences of these projects as a basis for the F/OSS research infrastructure. The success of these archives in the social science community is also a partial answer to questions of "why bother?"

Instrument Existing Tools

We should work with existing F/OSS community development tool projects to design plugins for instrumenting widely used F/OSS tools (such as Bugzilla, CVS/Subversion, etc.) to make the content of those tools available via APIs in standardized formats, administratively controllable by original tool/data owners. Such an effort could also benefit the community of F/OSS developers itself; this sort of instrumentation could help interfacing multiple tools, projects, and communities, and might increase willingness to participate.

Develop Data Standards

Standards for metadata and representation will help glue together data and tools such as finding aids and normalization tools. In collaboration with F/OSS tool developers, we should work toward standardizing formats and content of repositories of many kinds.

Create Federation Middleware

Federated approaches to data archives will have much lower initial costs and will foster community building while maintaining local control over base data and sharing. Foundations for such middleware exists (e.g. in Digital Library frameworks such as Fedora).

Develop Consensus on Data Selection Policies

We need much more consensus on what kinds of data provide the most utility for the widest variety of empirical F/OSS research projects. Developing this consensus will also help to congeal the community of empirical software researchers.

Create Prototypes

As a proof of concept, we should mock up a complete F/OSS research infrastructure model embodying as many of the desired characteristics as feasible. Such a partial implementation might use, for example, a complete cross section of sharable information from a single project, including chat, news, CVS, bug reporting, and so on. We have already instigated some local efforts in a few of these areas, such as generalized bug report schemas, semi-automated extraction of social processes, preliminary data taxonomies, automated analysis tools, and others have also begun efforts in these directions [4, 6, 8, 9].

[1]See for example `http://www.iue.it/LIB/EResources/E-data/online_archive.shtml` for a list of such collections.

In the end, efforts in these directions will pay off in the form of deeper collaborations in the empirical software research community, wider awareness of important research issues and means of addressing them, and ultimately in more systematic, grounded knowledge and theory-driven practice in software development.

References

[1] M. S. Elliott and W. Scacchi. Free software development: Cooperation and conflict in a virtual organizational culture. In S. Koch, editor, *Free/Open Source Software Development*. Idea Publishing, 2004.

[2] G. E. Evans. *Developing library and information center collections*. Libraries Unlimited, Englewood, CO, 4th edition, 2000.

[3] L. Gasser and W. Scacchi. Continuous design of free/open source software: Workshop report and research agenda, October 2003. `http://www.isrl.uiuc.edu/~gasser/papers/CD-OSS-prelim-report.pdf`.

[4] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on OSS Engineering*, Portland, OR, May 2003.

[5] S. Koch and G. Schneider. Results from software engineering research into open source development projects using public data. Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft, H.R. Hansen und W.H. Janko (Hrsg.), Nr. 22, Wirtschaftsuniversität Wien, 2000.

[6] Libre Software Engineering tool repository. `http://barba.dat.escet.urjc.es/index.php?menu=Tools`.

[7] 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.

[8] G. Ripoche and L. Gasser. Possible bugzilla modification to create a Web-API for direct XML serialization of bug reports. SQA Project Memo UIUC-2003-20, 2003.

[9] G. Ripoche and L. Gasser. Scalable automatic extraction of process models for understanding F/OSS bug repair. In *Proceedings of the International Conference on Software & Systems Engineering and their Applications (ICSSEA'03)*, Paris, France, December 2003.

[10] R. J. Sandusky, L. Gasser, and G. Ripoche. Bug report networks: Varieties, strategies, and impacts in an OSS development community. In *Proceedings of the ICSE/MSR Workshop*, Edinburgh, Scotland, UK, 25 May 2004.

[11] L. Shoman, E. Grossman, K. Powell, C. Jamison, and B. Schatz. The Worm Community System, release 2.0 (WCSr2). *Methods in Cell Biology*, 48:607–625, 1995.

[12] T. R. Smith. The meta-information environment of digital libraries. *D-Lib Magazine*, July/August 1996.

[13] S. L. Star and K. Ruhleder. Steps toward an ecology of infrastructure: Design and access for large information spaces. *Information Systems Research*, 7(1):111–134, 1996.

Mining CVS repositories, the softChange experience

Daniel M. German
Software Engineering Group
Department of Computer Science
University of Victoria
dmgerman@uvic.ca

Abstract

CVS logs are a rich source of software trails (information left behind by the contributors to the development process, usually in the forms of logs). This paper describes how softChange extracts these trails, and enhances them. This paper also addresses some challenges that CVS fact extraction poses to researchers.

1. Introduction

We have defined *software trails* as information left behind by the contributors to the development process, such as mailing lists, Web sites, version control logs, software releases, documentation, and the source code [5]. Software trails maintain a history of the development that can be used to recover the evolution of the project, to help management understand how it evolves and how its contributors work and interact, and to assist its contributors in their daily tasks.

In particular, software configuration management software, and more specifically version control software, keeps the complete history of any file in the project, including who modified what, when, and the delta of the modification. CVS, the Concurrent Versions System, is arguably the most widely used version control management system available in the market and has become a de-facto standard in the development of open source projects. softChange is a tool for the extraction, enhancement and visualization of software trails, primarily from CVS. The architecture of softChange is depicted in figure 1. The *trails extractor* is responsible for retrieving the raw software trails from the different sources. A SQL relational database management system is the core of softChange. A *fact enhancer* analyses the database in order to generate new facts. Finally, the *visualizer* is responsible for showing the trails to the user. softChange has been successfully used to recover the history of the software project Evolution (a mail client for Unix similar to Microsoft Outlook). The results are reported in [5]. softChange was used to extract Evolution's software trails, enhance them, and then query and visualize them. softChange helped us to understand how the project evolved, and how its developers collaborated. Another research project in which softChange was used is described in [4]. In this case we were interested in understanding the way that the software developers of the GNOME project (a large, open source project) collaborated. The analysis of these software trails allowed the discovery of interesting facts about the history of the project: its growth, the interaction between its contributors, the frequency and size of the contributions, and the important milestones in its development.

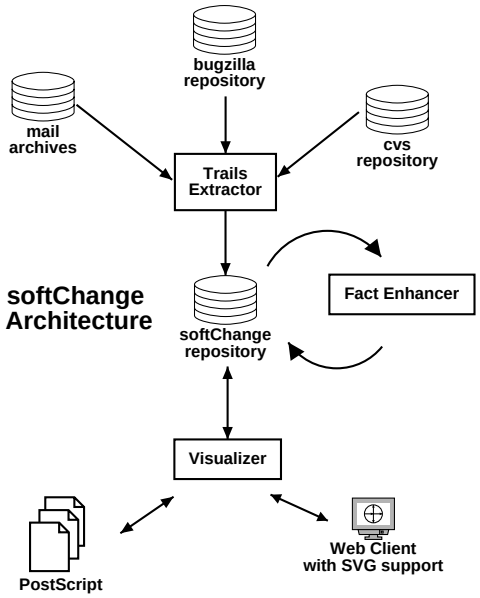


Figure 1. The architecture of softChange

This paper describes how softChange extracts software trails from CVS, and the methods used to create new

facts. Section 2 describes related work, section 3 explains softChange's fact extraction in detail, with examples of how it was used to extract the CVS software trails from four major projects. Section 4 describes current challenges in trail extraction from CVS. We finish with our conclusions and future work.

2. Related Work

The two most commonly used hypertext frontends to CVS are Bonsai [7] and lrx [6]. They operate by retrieving the revision information of each file, which is then stored in a relational database. Xia is a plugin for Eclipse for the visualization of CVS repositories[10] based on the Shrimp visualization tool [9]. Xia does not extract the CVS trails, instead it relies on the Eclipse's API to CVS, which makes it extremely slow in large projects. Liu and Stroulia have developed JReflex, a plug-in for Eclipse for instructors of software engineering courses [8]. It is designed to compare the differences in development styles in different teams, who does what, who works on what part of the project, etc. JReflex is intended to be a management oriented tool for browsing the CVS historical data. It uses the history log in CVS and the output of CVS log and stores the information in a relational database. Fisher and Gall have described a CVS fact extractor in [1], where they described the main challenges of creating a database of CVS historical data and then use it to visualize the interrelationships between files in a project [2]. In [3] CVS logs are used to expose relationships between classes and files that might not be found by other methods, such as call graphs.

3. Mining a CVS repository

3.1. Retrieving file revisions

Projects mining CVS historical data have relied on processing the output of its commands (e.g. `cvs log`), or the log files in the repository (`CVSROOT/history`). Unfortunately, the format of the output of CVS commands and its log files is not fully documented. In order to understand these formats, the first phase of the development of softChange used a "clean room" method. Our goal was to recover all the revisions to all the files in the repository. We followed the following procedure:

1. We selected one project as a test case (Evolution), and detailed the requirements for the extractor.
2. We were divided into two independent teams.
3. Each team reversed engineered the CVS formats and proceeded to create the extractor.
4. The extractors were run on the Evolution CVS repository, and their outputs compared.
5. When there were differences in the outputs both teams discussed the problem and determined which team's extractor was faulty (in some case, both were). Teams exchanged information about the formats but there was never exchange of code between teams.
6. We repeated this process until the extractors generated the same output.
7. The code from one team was dropped, and the other became the core of softChange.
8. We then proceeded to create a set of test-cases for future regression testing.

We have used softChange to extract the file revisions from several projects. Table 1 shows the main statistics of four selected projects. A snapshot of their CVS repositories was made on Feb 17, 2004[1]. Mozilla [2] corresponds to the cross-platform Web browser, Evolution is a email client for Unix similar to Outlook, PostgreSQL is a SQL database management system; GNU gcc is the multi platform, multi language compiler.

Table 1. CVS statistics from selected projects

Project	Authors.	Files	Revisions
Mozilla	672	81,520	709,234
Evolution	245	5,402	92,688
PostgreSQL	24	3,789	74,541
GNU gcc	214	24,463	60,311

3.2. Rebuilding modification requests

CVS mining projects usually work at the file revision level. Files, however, are not usually modified alone. A developer will modify all the files necessary to complete a given task, and then commit them together (using the `cvs commit` command). Knowing which files are modified at the same time is important because it means that these files are somehow related (the change in one file is related to the change in the other file).

[1] You can find a copy of the cvs log command for each of the reviewed projects in http://view.cs.uvic.ca/softChange/mining2004/

[2] The Mozilla CVS repository keeps track of the email address of the developer in its cvs id. A typical Mozilla cvs id has the form userid%domainname. An inspection of the different cvs ids suggests that the same developer has used different cvs ids, as her corresponding email address changes. For example these are three different cvs ids that seem to correspond to the same person: `alecf`, `alecf%flett.org`, `alecf%netscape.com`. There are 505 unique cvs ids when the domain name suffix has been stripped.

Unfortunately CVS does not keep track of which files are committed at the same time. By analyzing the files' revisions softChange tries to recover, for each cvs commit, the files that its invocation modified. We denote a modification request (MR) as the set of files committed simultaneously by a developer in a "cvs commit" command.

To our knowledge, softChange is the only tool that attempts to recover modification requests. It uses a heuristic that is based on a sliding window algorithm. This algorithm takes 2 parameters as input: the maximum length of time that an MR can last δ_{max}, and the maximum distance in time between two file revisions τ_{max}. This algorithm is depicted in figure 2. Briefly, a file revision is included in a given MR if a) all the file revisions in the MR and the candidate file revision were created by the same author and have the same log message (a comment added by the developer during the commit); b) the candidate file revision is at most τ_{max}seconds apart from at least one file revision in the MR; and c) the addition of the candidate file revision to the MR keeps the MR at most δ_{max}seconds long.

```
// front(List) removes the front of the list
// top(List) and last(List)
//    query the corresponding elements of the list
// Initialize set of all MRs to empty
MRS = ∅
for each A in Authors do
    List = Revisions by A ordered by date
    do
        MR.list = {front(List)}
        MR.sTime = time(MR.list_1)
        while first(List).time − MR.sTime ≤ δ_max ∧
                first(List).time −
                    last(MR.list).time ≤ τ_max ∧
                first(List).log = last(MR.list).log ∧
                first(List).file ∉ MR.list do
                queue(MR.list, front(List))
        od
        MRS = MRS ⋃ {MR}
    until List ≠ ∅
od
```

Figure 2. Algorithm to recover MRs

Most MRs take few seconds to complete. But some tend to be rather long. There are several factors that affect the length of a MR. First, the size and number of files that compose the MR; second, the bandwidth available between the developer's computer and the CVS server (a slow link will slow down the time required to do the commit); and third, the load of the CVS server. In our experiments we have found that $\tau_{max} = 45s$ and $\delta_{max} = 600s$ are good values for these parameters (these values were used to extract the MRs discussed in this paper). Smaller values for these parameters tend to split MRs, and larger numbers tend to combine two MRs into one).

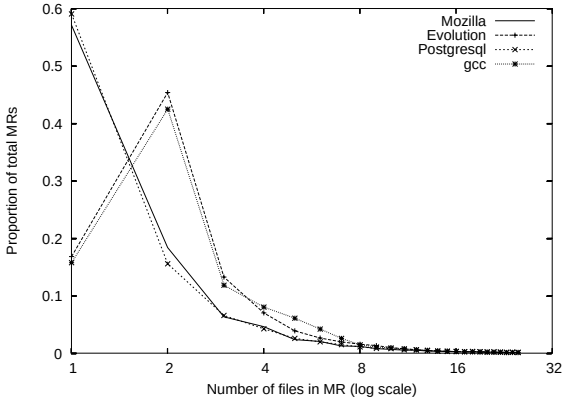


Figure 3. Number of files in an MRs

Most MRs contain very few files. Figure 3 shows the distribution of the number of files in a MR (normalized to values from 0 to 1). The plot only shows MRs with 25 or less files), but there are larger MRs (for example, in Evolution we detected an MR which included 650 files, and in Mozilla one that included 5838 files). Note that the four projects have only 2, almost identical curves. This effect was interesting enough to further explore. We discovered that the use of *ChangeLog* files (files that document the changes made to the software) accounted for this sharp difference. Evolution and GNU gcc use *ChangeLogs*, and almost every MR that includes two or more files includes a change to a *ChangeLog* file. Mozilla and PostgreSQL do not use them. When *ChangeLogs* are not taken into account all the curves look remarkably similar. Further research is needed to verify if this is a coincidence or, indeed, this is a normal pattern in software development. Figure 4 shows the distribution of MRs during 2003 for the chosen projects.

3.3. Other software trails

softChange is able to retrieve and use other trails:

- *ChangeLog* files. If the project uses *ChangeLogs*, for every MR softChange extracts the delta of the corresponding *ChangeLog* file and associates it with it. *ChangeLogs* were originally defined by the Free Software Foundation, and they are commonly found in open source projects, and their objective is to explain how earlier versions of software are different from the current version. Figure 5 shows an excerpt of a *ChangeLog*. The format of a *ChangeLog* delta is very simple: the first line contains the date and author, followed by a sequence of changes (all indented).

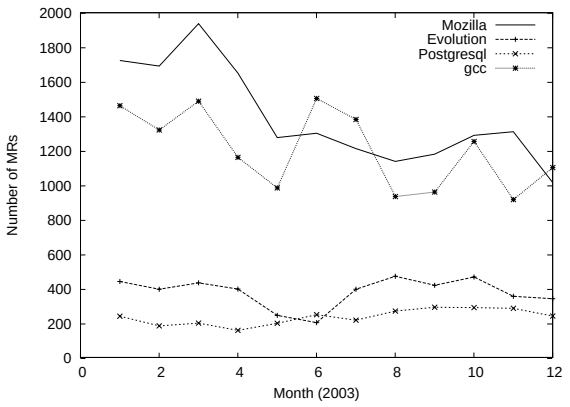


Figure 4. MRs per month, 2003

ChangeLogs are usually created by hand, although there are some utilities that help the developer in their creation. We plan to compare the information found in MRs with the one recorded in *ChangeLogs* in order to verify, both, the MR extraction algorithms and the quality of the *ChangeLogs*. Some projects use the *ChangeLog* delta as the corresponding CVS log message for a given MR.

- Bugzilla: It is customary for developers to record the Bugzilla bug number in the corresponding CVS log message of the MR that fixes it. Because this is a free-form, textual field, there is no standard on how this information should be recorded. Figure 6 shows several cvs log comments that correspond to bug fixes. Based on our observations, the following regular expression matches the most commonly forms in which a bug number is reported ($\backslash s$ corresponds to any white space character):

```
(\#[0-9][0-9]+|bugs?\s+\#?[0-9][0-9]+)(,\s+\#[0-9][0-9]+)*
```

  Unfortunately this is an error prone approach and the bug numbers identified need to be correlated to the Bugzilla database, in order to find out if the time that the MR was committed is consistent with a change in the bug report (softChange does not currently support this verification).

- Mailing lists. Mailing lists are an important source of information about the evolution of the project. We currently correlate MRs to mail messages by using the author and the date attributes of both the MR and the message. One of the problems we have encounter mining email messages is that a person tends to have multiple email address, which might not be the same as the ones recorded in the *ChangeLogs*.

```
2003-01-27  Ettore Perazzoli  <ettore@ximian.com>

        * tools/evolution-addressbook-export.c: #include bonobo-activation
        instead of oaf.
        (main): Initialize using gnome_program_init().
        (save_cards): Use g_main_loop_quit() instead of gtk_exit().

        * tools/evolution-addressbook-import.c: Update include list for
        GNOME 2.
        (main): Initialize using gnome_program_init().
        (unref_executable): Use g_main_loop_quit() instead of gtk_exit().
        (add_cb): Likewise.
```

Figure 5. Excerpt from a *ChangeLog*

```
...
        * mail-display.c (mail_display_render): Set default text color
        as black in body when doing printing preview. Fixs bug #48290.
...
Bugzilla bug #218: define HAVE_STRERROR only if it is not yet defined.
Thanks to David Nebinger (dnebinger@synertech.highmark.com) for reporting
the problem and suggesting the fix.
```

Figure 6. Excerpts from cvs log comments referring to bug reports.

4. The challenges of mining CVS repositories

One of the consequences of a growing number of projects extracting different software trails is the explosion of terms to describe them. For example, [1] refers to a CVS revision as a *cvsitem*, and its log as *description*. As the field of mining software repositories matures, we expect that the nomenclature becomes more consistent.

CVS commands are able to access a CVS repository in two different forms: across the network (when the CVS server is located on a different computer), or in the local file system (when the CVS repository is located on the same computer as the working copy of the repository). Mining the repository might result in numerous requests and a large amount of resulting data. For example, softChange can regenerate every revision of a file. In a project such as Mozilla, this will require requesting more than 0.7 million different files (one for each filename, revision pair). If the repository is located in a different computer, this process will most certainly stressed it, and it will consume a large chunk of its bandwidth. This problem will be aggravated if several researchers start using Mozilla as a test case. This problem can be avoided by having a local snapshot of the project's CVS repository. Having a local copy of the repository will guarantee that the resources of the software project are not over-used.

The creation of a common set of test cases is also needed. Different research groups analyzing different software trails have chosen different applications for their analysis. This makes it difficult to compare approaches. We propose the selection of a small set of application that could be used for this purpose. The applications should satisfy the following requirements:

- These applications should be a mixture of old and new,

applications, large, medium sized and maybe small ones. Some should include a GUI, while some should not have any. Some might be dead projects.

- Ideally, the original logs and historical data should be made available to the researchers. For instance, the researchers should have a copy of the CVS repository, a dump of the Bugzilla data, a copy of the raw mailing lists archives, etc. This is important because it avoids potential problems created by extracting the data from views of it (such as scrapping bugzilla data from its Web front-end) and it also avoids the extra load on the project servers due to the requests made by the research project.

- It is necessary to agree on the period of observation of the project. Most likely, the chosen projects are alive, and keep changing. Hence it is necessary to specify the start and end date for the observation of a given project.

- Projects with open source licenses are desirable. It is important that the project being analyzed does not put any restrictions on the researcher (like not being able to publish benchmarks of the application). An open source license guarantees no discrimination against using the software. It also provides access to the source code, and equally important, to its software trails. It is undeniable that close-source applications are worth exploring, but they cannot be used as test-cases because they might not be available to any researcher that wants to look at them.

Some projects have become typical test cases in the literature. Mozilla, for example, is one of them. But one has to understand the characteristics of a project before using it as a test case, in order to interpret its data correctly. The Mozilla project started using CVS when the source code of Netscape became Mozilla, and therefore, not all its history is recorded. Another peculiar feature of Mozilla is that several developers have more than one CVS id (we have found developers with two or three cvs ids). Nonetheless, it is a very valuable test case, as it provides the researcher with a large, mature and widely used project that keeps evolving and it is maintained by a large number of individuals.

One important issue that has not been clearly addressed yet is the ethical one. Would the developers of an open source project consider their software trails open too? What are the implications of publishing aggregated data about a project? For example, would it be ethical to claim (in a research paper for example) that code from certain developer tends to have more defects than any other developer's code in the same project? Should projects and their developers be anonymized? The answers to these questions could be the subject of an entire paper.

5. Conclusions and Future Work

CVS is widely used in software projects, some of which are several years old. The information available in its logs can be very valuable for its developers, their management and researchers as it provides a fine-grained view of how the software project is evolving. Unfortunately the amount of data can be overwhelming. Work is needed in several directions: models to describe this data, and query and visualization tools to inspect it. softChange is still under development, but we welcome people interested on using it.

Acknowledgments

This research was supported by NSERC Canada, and the Advanced Systems Institute of British Columbia. The author would like to thank A. Mockus (original co-author of `softChange`) and the anonymous reviewers of this paper.

References

[1] 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.

[2] M. Fisher and H. Gall. MDS-Views: Visualizing problem report data of large scale software using multidimensional scaling. In *Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA)*, September 2003.

[3] H. Gall, M. Jazayeri, and J. Krajewski. CVS Release History Data for Detecting Logical Couplings. In *Proc. of the International Workshop on Principles of Software Evolution (IWPSE)*, pages 12–23. IEEE Press, November 2003.

[4] D. M. German. Decentralized open source global software development, the GNOME experience. *Journal of Software Process: Improvement and Practice*, accepted for publication.

[5] D. M. German. Using software trails to rebuild the evolution of software. *Journal of Software Maintenance and Evolution: Research and Practice*, to appear, 2004.

[6] A. G. Gleditsch and P. K. Gjermshus. lrx Cross-Referencing Linux. http://lxr.sourceforge.net/, Visited Feb. 2004.

[7] T. Hernandez. The Bonsai Project. http://www.mozilla.org/projects/bonsai, Visited Feb. 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] M.-A. D. Storey, C. Best, and J. Michaud. SHriMP Views: An Interactive and Customizable Environment for Software Exploration. In *Proc. of International Workshop on Program Comprehension*, May 2001.

[10] X. Wu. Visualization of version control information. Master's thesis, University of Victoria, 2003.

Text is Software Too

Alexander Dekhtyar
Dept. Computer Science
University of Kentucky
dekhtyar@cs.uky.edu

Jane Huffman Hayes
Dept. Computer Science
University of Kentucky
hayes@cs.uky.edu

Tim Menzies
Dept. Computer Science,
Portland State University,
tim@menzies.us

Abstract

Software compiles and therefore is characterized by a parseable grammar. Natural language text rarely conforms to prescriptive grammars and therefore is much harder to parse. Mining parseable structures is easier than mining less structured entities. Therefore, most work on mining repositories focuses on software, not natural language text. Here, we report experiments with mining natural language text (requirements documents) suggesting that: (a) mining natural language is not too difficult, so (b) software repositories should routinely be augmented with all the natural language text used to develop that software.

1 Introduction

"I have seen the future of software engineering, and it is.....Text?"

Much of the work done in the past has focused on the mining of software repositories that contain structured, easily parseable artifacts. Even when non-structured artifacts existed (or portions of structured artifacts that were non-structured), researchers ignored them. These items tended to be "exclusions from consideration" in research papers.

We argue that these non-structured artifacts are rich in semantic information that cannot be extracted from the nice-to-parse syntactic structures such as source code. Much useful information can be obtained by treating text as software, or at least, as part of the software repository, and by developing techniques for its efficient mining.

To date, we have found that information retrieval (IR) methods can be used to support the processing of textual software artifacts. Specifically, these methods can be used to facilitate the tracing of software artifacts to each other (such as tracing design elements to requirements). We have found that we can generate candidate links in an automated fashion faster than humans; we can retrieve more true links than humans; and we can allow the analyst to participate in the process in a limited way and realize vast results improvements [10, 11].

In this paper, we discuss:

- The kinds of text seen in software;
- Problems with using non-textual methods;
- The importance of early life cycle artifacts;
- The mining of software repositories with an emphasis on natural language text; and
- Results from work that we have performed thus far on mining of textual artifacts.

2 Text in Software Engineering

Textual artifacts associated with software can roughly be partitioned into two large categories:

1. Text produced during the initial development and then maintained, such as requirements, design specifications, user manuals and comments in the code;
2. Text produced after the software is fielded, such as problem reports, reviews, messages posted to on-line software user group forums, modification requests, etc.

Both categories of artifacts can help us analyze software itself, although different approaches may be employed. In this paper, we discuss how lifecycle development documents can be used to mine traceability information for Independent Validation & Verification (IV&V) analysts and how artifacts (e.g., textual interface requirements) can be used to study and predict software faults.

3 If not text..

One way to assess our proposal would be to assess what can be learned from alternative representations. In the software verification world, reasoning about two represenations are common: formal models and static code measures.

A formal model has two parts: a *system model* and a *properties model*. The system model describes how the program can change the values of variables while the properties model describes global invariants that must be maintained when the system executes. Often, a temporal logic[1] is used

[1]Temporal logic is classical logic augmented with some temporal operators such as $\Box X$ (always X is true); $\Diamond X$ (eventually X is true); $\bigcirc X$ (X is true at the next time point); $X \bigcup Y$ (X is true until Y is true).

to express the properties model. Modern model checkers such as SPIN [15] search the systems model for a method of proving the negation of the properties model. The cost of formal modeling includes the *writing cost*, the *running cost*, and the *rewriting costs*. The writing cost has two components. Firstly, there is a short supply of analysts skilled in creating temporal logic models. Secondly, even when analysts with the right skills are available, the writing process is time-consuming.

Another significant cost of formal modeling is the *running cost* of model checking. A rigorous analysis of formal properties implies a full-scale search through the systems model. This space can be too large to explore, even on today's fast machines. Much of the research into formal modeling focuses on how to reduce this running cost of model checking. Various techniques have been explored but none are panaceas. For example, optimisations based on clustering (e.g. [3]) generally fail for tightly connected models. Consequently, in the general case, classic formal methods do not reduce the effort of testing a system. However, for the kernel of mission-critical or safety-critical systems, the large cost of formal methods is often justified.

At the other end of the spectrum from model-rich formal modeling are defect measures based on syntactic static code measures such as the Halstead [7] or Mccabe [18] metrics. Such static code measures are a weak *primary* method of finding errors. Such metrics are collected on a module-by-module basis. Hence, they know neither: (a)how often that module will be called, nor (b)the severity of the problem resulting from the module failing, nor (c)the connections *from* this module *to* other modules.

However, static code measures are adequate *secondary* detectors that can audit the results of primary methods. Elsewhere, we have shown that such detectors are stable across multiple projects and can be selected such that they have a very low probability of false alarms [19].

Nevertheless, the current situation is as follows. Complex comprehension methods such as formal modeling can be too complex for many applications. Simpler methods such as static code measures may only be suitable for *augmenting* other methods. In neither case do we possess methods that are both very insightful and widely applicable. We are hence motivated to work on other methods.

This understanding has been demonstrated by several other researchers including Di Lucca, Di Penta and Gradara [5], who have examined the problem of classifying, via a variety of different algorithms, textual maintenance requests into eight categories. Their best result, using support vector machines (SVM) [16], was 84% accuracy. Lee and Bryant [17] examined the problem of formalizing natural language requirements specifications unsing Natural Language Processing (NLP) techniques. Thus, we observe that in recent years, information retrieval and text mining methods are starting to be applied to adress Software Engineering problems.

4 Possibility

So, what can we do if we add text to software repositories? Here, we discuss two different problems that can be addressed in such a manner: fault analysis and requirements tracing.

4.1 Fault Analysis

Barry Boehm's seminal work in software engineering economics convinced us that faults found early in the lifecycle are less expensive and time consuming to correct [2]. We have worked as practitioners and researchers in the area of verification and validation for over twenty years, and we have seen this confirmed many times. In fact, we are convinced that faults found early in the lifecycle can serve as predictors of faults that will be found later in the lifecycle. We further believe that textual analysis can assist. For example, an unsatisfied high level requirement (one that does not have design elements to satisfy it) may lead to a missing capability in the as-built software product.

Evidence of such fault links (the relationship of one fault to another) was presented by Hayes and Offutt [9, 12, 13]. Specifically, high-level interface requirements (textual) were examined using a technique called input validation analysis. Faults in the interface requirements were identified as well as potential faults (ambiguities, for example). Test cases were generated on the basis of these early life cycle faults. The test cases were then executed on the as-built software and 13% of these revealed later life cycle faults in the delivered product. Hayes postulated that these early life cycle faults were late life cycle predictors for two reasons: developers tend to make the same kinds of mistakes, regardless of the life cycle phase; and faults do not get repaired early in the life cycle and are detected later (latent defects) [9].

Knowing that faults caught early in the life cycle are easier and less costly to repair AND can assist us in predicting and discovering later life cycle faults is a call to action. We should fully explore techniques that allow us to analyze early lifecycle artifacts for such faults. We should not be dissuaded from our duty by the existence of textual narrative in these early artifacts.

4.2 Requirements Tracing

Requirements tracing, a bane of Independent Verification & Validation (IV& V) analysts, is a prolonged, tedious, but *incredibly important* task of making sure that all initial software requirements have been adequately reflected in the design specifications for the software, and, eventually, in the code. Traditional approaches to requirements tracing ivolve repeatedly going through hardcopies of requirements documents, building and manually maintaining spreadsheets, or, at best, using requirements management tools that allow manual assignment of keywords to requirements and use simple keyword matching algorithms to find candiate links. Such procedures reflect the nature of the task: requirements documents are written in natural, if somewhat

more bureaucratically formal, language. So far, human cognitive powers are unmatched in detecting correspondence between two (or more) text fragments: requirements and design elements, for example.

The only reason why, up to this day, such procedures are still employed is the relatively small size of the documents under consideration for the requirements tracing task. Even then, large projects have thousands of requirements and, potentially, tens of thousands of design elements: approaching the limits of what IV& V analysts are prepared to suffer through without extra help.

At the same time, in the core of the requirements tracing task, lies a problem well-known to computer scientists and, in fact, well-studied by them: *given a document collection, and a document (query) find all such documents in the collection that are similar (relevant) to it.* This problem, addressed by decades of intensive research in Information Retrieval (IR), is becoming ubiquitous, at the very least for those of us who use Internet on a daily basis. And our ability to search for, *and find*, information in the pits of the World Wide Web only attests to the success of Information Retrieval in dealing with this problem.

Thus, we have reached the conclusion that by taking the low level requirements (design elements) to be the document collection, and by treating high level requirements as queries, we can use the vast array of IR algorithms (see [1] for the starting point) to produce lists of candidate links for the requirements traceability matrix. Our preliminary experiments, reported in [10, 11] showed that:

- automated means of generating candidate links work much faster than humans (even when humans are assisted by existing requirements management software);
- automated means of generating candidate links retrieve more *true links* than the human analyst/requirements management software combination;
- automated means of generating cadidate links tend to report *more false positives* than human analyst/requirements management software combination;
- analyst participation in the process, as the validator of the candidate links, is still important.

Following this work, we have implemented additional IR algorithms, and incorporated *user feedback analysis* into the system, making the requirements tracing process interactive again and giving human analysts the last word in determination of the links. At the same time, feedback analysis has shown the ability to improve significantly both the *recall* (percentage of true links found) and *precision* (the measure of the signal-to-noise ratio in the list of candidate links), especially when combined with techniques for filtering outputs produced by our IR methods [11]. This lead to creation of RETRO (REquirements Tracing On-target), a standalone, IR-based requirements tracing tool for IV& V analysts [11].

In Table 1, we briefly summarize RETRO's achievements to date. The two main metrics of success of an IR task that are applicable to the requirements tracing problem itself are *recall*, the percentage of all true links retrieved, and *precision*, the percentage of true links in the answer set. Recall measures coverage, while precision measures signal-to-noise ratio. The top two rows in Table 1 show the precision and recall obtained from a commercial requirements management tool, SuperTracePlus (STP) [8, 20], and from a senior analyst working with the output of the SuperTracePlus on a simple test set consisting of 19 high-level and 49 low-level requirements. The remaining rows show how different methods that we have implemented in RETRO fare on the same test. TF-IDF is a standard [1] IR method that computes similarity between documents as the cosine of the angle between their vector representations. This method has been enhanced with user feedback [1], various filtering techniques [11], and a simple thesaurus [11]. LSI stands for "Latent Semantic Indexing", a dimension reduction technique that proved to work very well on small document collections [4].

Method	Precision	Recall
STP	38.80%	63.41%
Analyst+STP	46.15%	43.9%
TF-IDF	11.3%	57.3%
TF-IDF+ Feedback	18.6%	**76.2%**
TF-IDF+ Feedback+Filter	**60.9%**	59.5%
TF-IDF+ Thesaurus	12.2%	**64.2%**
TF-IDF+ Thesaurus + Feedback	18.1%%	**83.3%**
TF-IDF+ Thesaurus + Feedback + Filter	**39.5**%	**80.9%**
	73.8%	**73.8%**
LSI (10 dim, 0.32 coverage)	5%	**90.4%**
LSI (10 dim, 0.32 coverage)+ Thesaurus	5%	**92.85%**
LSI (40 dim, 0.92 coverage)	5%	**80.95%**
LSI (40 dim, 0.92 coverage)+ Thesaurus	5%	**85.71%**

Table 1. Using information retrieval to trace requirements.

In each row we report the best, in our opinion, combination of recall and precision that was obtained during the experiments (for TF-IDF+Thesaurus+Feedback+Filter we report two different results achieved). As can seen from the table, most of the methods tested within RETRO consistently outperform humans and STP in recall. At the same time, these methods trail in precision, a feat that can be corrected by the use of filtering techniques at the price of some decrease in recall. In general, our findings today show that there is significant potential in the use of well-established IR methods for analyzing textual artifacts.

5 But What's the Price?

All things considered, *textual artifacts* are less well-understood than *code*. It stands to reason that the analysis of such artifacts must be conducted with more complicated algorithms, than the analysis, even mining, of code alone. At the same time, we should also be prepared for the taste of failure: not all methods of text analysis will work in our settings. Our experience with the use of IR algorithms for requirements tracing lead us to discover the following specific features of mining software-related text:

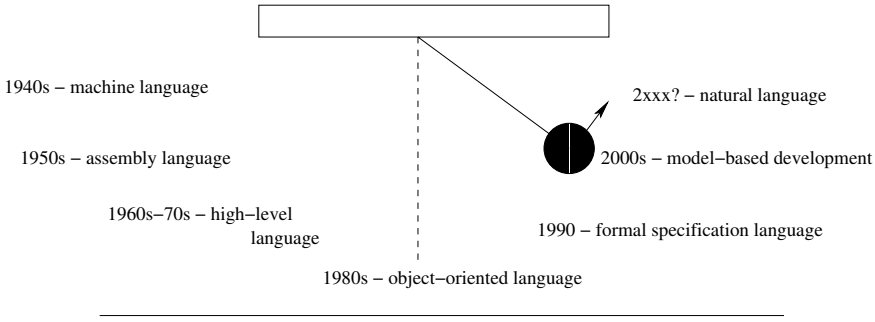


Figure 1. The Artifact Pendulum Swings from Structure to Less Structure.

- domain size: traditional IR algorithms represent documents as vectors of keyword weights. Such methodology works very well when the document collection is large enough to approximate the real use of different terms in English. Thus, the methods of determining the importance of a keyword for a document that work extremely well when there are billions of documents in the collection, at times, have strange effects when the number of documents is in tens or hundreds.
- document size: traditional IR algorithms assume that the individual documents contain significant text. Most of traditional test collections for IR algorithms [14] use documents that have on average more than one hundred words. At the same time, it is not unusual for a requirement to consist of one or two simple sentences. That is to say, the fewer the words in the document, the fewer keywords detected.
- incomplete vocabulary: requirements documents are, very often, written in a very specific lingo. Combined with the relatively small number of requirements, it makes the vocabularly of the entire collection, both high- and low- level requirements, incomplete, and sometimes, different from the traditional English usage vocabularly (in terms of frequency of use of words). Thus, some words, that are treated as almost stopwords[2] elsewhere, may suddenly give the appearance of very important keywords, only because they are used in only one or two requirements.
- recall vs. precision: typically, both recall and precision are equally important. However, their roles are drastically different in requirements tracing tasks. Recall appears to us as more important as, at the end of the day *all* matching requirement pairs must be found. Precision plays a role of a "filter": it determines how many false positives will be examined by the human analyst. Very low precision, makes automated candidate link generation useless, however, while our goal is always 100% recall, precision of even 40%–50% is excellent (analyst has to examine about one false positive per true link) as it drastically reduces human effort as compared to the manual process.

[2]Stopwords are words that are not considered to be keywords: articles, prepositions, pronouns, modal verbs, and some common verbs and nouns (such as "be", "get", "thing", "stuff"). In addition, special collections of documents may have extra stopwords, e.g., "software" is a stopword in a collection of Software Engineering papers.

All of this leads to the observation that while we should attempt to take as much advantage of already designed information retrieval, text mining and/or natural language processing methods, we should also be ready to: (a) accept unapplicability of some of them to specific problems, and (b) not only *adopt* but ***adapt*** them to the needs and features of these problems.

6 Conclusion

One could argue that the software engineering artifact pendulum has been swinging from more formal, structured and parseable means of describing software to more text-based ever since the inception of the discipline. As can be seen in Figure 1, the beginning of it all was the pleasant-to-parse machine language (top left). Wise pioneers of our field realized that the price was too high for the poor human programmers, and came up with assembly language... and the pendulum came into motion. High-level procedural languages that came next attempted to record the algorithm rather than its direct execution by the computer. Assembly command abbreviations were replaced with keywords, control structures and identifiers of (practically) arbitrary length. Then, in the 1980s, we decided that an even higher level of abstraction was needed: the ability for developers to think of things in terms of objects.

All this high level thinking, however, had not been leading to drastically better software. In fact, requirements were just as poorly specified by software engineers using UML and use cases and other "texty" artifacts that came along with object oriented techniques.

The next step (the pendulum starts going up) lead to formal specification languages. The potential of these languages cannot be denied. The ability to parse such artifacts and even use specification provers to ensure that

source code implements a formal specification is powerful indeed. Formal methods promise automatic verification and automatic generation of demonstrably correct code. However, the experience with such tools to date is not positive. Ph.D.-level mathematical skills may be required to specify the knowledge required for such tools [21]. Commercial practitioners may lack either the required training or the required time needed for such specification.

So the pendulum continues to rise. Other researchers have argued for lightweight modeling languages with formal semantics (e.g. [6]). Here, we propose something different. After decades of research, we have evidence that *mere* text can be more useful than previously believed. Our recommendation is that when repositories are built, we should always include all available text artifacts.

Text mining from software engineering text is a high risk, high return adventure. The translation steps from high level artifacts, such as concept documents and high level requirements statements, to low level implementation, such as source code, inject a tremendous amount of variance into the final artifacts. At the same time, it is precisely this variance that hurts software development process, especially the validation and verification part of it. Thus, we maintain that *achieving a better understanding of how text turns into code* will lead to improved software.

Acknowledgements

This research was conducted at West Virginia University, Portland State University, and the University of Kentucky under NASA contracts NCC2-0979, NCC5-685, and NAG5-11732. The work was partially sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, Addison-Wesley, 1999.

[2] B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[3] P. Clark and T. Ng. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[4] S. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the Society for Information Science*, 41(6):391–407, 1990.

[5] G. Di Lucca, M. Di Penta, and S. Gradara. An approach to classify software maintenance requests. In *Proc., International Conference on Software Maintenance (ICSM)*, 2002.

[6] S. Easterbrook, R. R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. Experiences using lightweight formal methods for requirements modeling. *IEEE Transactions on Software Engineering*, pages 4–14, 1998.

[7] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[8] J. Hayes. Risk reduction through requirements tracing. In *The Conference Proceedings of Software Quality Week*, 1990.

[9] J. H. Hayes. Input validation testing: A system level, early lifecycle technique. In *ICSE '97 Doctoral Consortium, published in the Proceedings of the Seventeenth International Conference on Software Engineering Doctoral Consortium*, May 1997.

[10] J. H. Hayes, A. Dekhtyar, and J. Osbourne. Improving requirements tracing via information retrieval. In *International Conference on Requirements Engineering, Monterey, California*, pages 151–161, 2003.

[11] J. H. Hayes, A. Dekhtyar, S. Sundaram, and S. Howard. Helping analysts trace requirements: An objective look. In *International Conference on Requirements Engineering (RE'2004)*, 2004.

[12] J. H. Hayes and J. Offutt. Input validation testing: A requirements-driven, system level, early lifecycle technique. In *Proceedings of the 11th International Conference on Software Engineering and its Applications*, October 1998.

[13] J. H. Hayes and J. Offutt. Increased software reliability through input validation analysis and testing. In *Proceedings of The Tenth IEEE International Symposium on Software Reliability Engineering*, pages 199–209, 1999.

[14] W. Hersh and P. Over. The trec-9 interactive track report. In *Proc. Text Retrieval Conference (TREC-9)*, pages 41–50, 2000.

[15] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[16] T. Joachims. Text categorization with suport vector machines: Learning with many relevant features. In *Proc. ECML*, pages 137–142, 1998.

[17] B. Lee and B. Bryant. Contextual knowledge representation for requirements documents in natural language. In *Proceedings of FLAIRS, the 15th International Florida Artificial Intelligence Research Symposium*, 2002.

[18] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[19] T. Menzies, J. S. D. Stefano, C. Cunanan, and R. M. Chapman. Mining repositories to assist in project planning and resource allocation. In *International Workshop on Mining Software Repositories (submitted)*, 2004. Available from `http://menzies.us/pdf/04msrdefects.pdf`.

[20] T. Mundie and F. Hallsworth. Requirements analysis using supertrace pc. In *Proc. American Society of Mechanical Engineers (ASME) for Computers in Engineering Symposium at the Energy and Environmental Expo*, 1995.

[21] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, 1990.

Integration and Presentation

GlueTheos: Automating the Retrieval and Analysis of Data from Publicly Available Software Repositories

Gregorio Robles
Universidad Rey Juan Carlos
grex@gsyc.escet.urjc.es

Jesus M. González-Barahona
Universidad Rey Juan Carlos
jgb@gsyc.escet.urjc.es

Rishab A. Ghosh
MERIT - Univ. Maastricht
rishab@merit.unimaas.nl

Abstract

For efficient, large scale data mining of publicly available information about libre (free, open source) software projects, automating the retrieval and analysis processes is a must. A system implementing such automation must have into account the many kinds of repositories with interesting information (each with its own structure and access methods), and the many kinds of analysis which can be applied to the retrieved data. In addition, such a system should be capable of interfacing and reusing as much existing software for both retrieving and analyzing data as possible.

As a proof of concept of how that system could be, we started sometime ago to implement the GlueTheos system, featuring a modular,flexible architecture which has been already used in several of our studies of libre software projects. In this paper we show its structure, how it can be used, and how it can be extended.

Keywords: *Mining source code repositories, proposals for exchange formats, meta-models, and infrastructure tools, integration of mined data with other project data*

1 Introduction

Libre software projects[1] range from very small ones (with just one developer commited to his own toy) to large-scale global projects whith thousands of collaborating developers [9]. Specially, most of the larger projects follow a way of organization that has been called the 'bazaar'-style development [14], open to everybody willing to participate. Thus, all elements taking part in the software development process are as much open as possible, in the sense that the generated information is publicly available so that it is easier for 'newcomers' to become integrated in the project. Fortunately, this strategy offers to researchers the chance to access large amounts of data about the development process, the participants and, of course, the output product: the software.

Previous studies have taken advantage of this situation, and several research groups have focused their attention on the libre software phenomenon in the last years. For instance, [6] offers a software evolution analysis of the Linux kernel versions -without doubt the most known libre software project- following the classical software evolution point of view [11]. Others have paid attention to economic parameters [10] and have investigated how well classical software cost prediction models (as among others COCOMO [1]) can be applied. In [13] it is shown how libre software projects are composed usually of 10 to 15 core developers who lead the software process, a group of around one order of magnitude larger that participate in minor development tasks (bug fixes, etc.) and a final group around another order of magnitude that helps by other means (bug reports, etc.). In any case, the availability of data has proven to be very positive for research in the libre software environment.

But the amount of data and information available for inspection is that big that these analysis are often regarded as being too superficial. An example where this is common case are source code repositories. In such systems, not only the last state of the code is available for download but also all previous states. The amount of information that is ready for being extracted and analyzed is enormous and two factors become key points: automation and data mining.

When analyzing the data available in publicly accessible repositories, the automation of the data retrieval and the quantitative analysis is of great importance[15][3]. In the case of libre (free, open source) software projects, repositories are managed with very similar software (if not the

[1] In an attempt to avoid any confusion regarding the meaning of free in free software, throughout this article, the term libre software is used instead. It was chosen because of its meaning pointing towards liberation, and not of merely being costless. The term Open Source is refused for its ignorance about the philosophical foundations of what free software meant in the first place. "Libre software" is a term which is more and more usual in some communities, among them many European and Latin American countries.

same)[2], and similar access protocols, so automation allows for the access to most of the available projects. Some methodologies have already being described which make strong use of some kind of automated tools to perform this tasks [3, 4, 7, 17, 10, 15] but they usually make use of ad-hoc tools, without proposing a general architecture flexible enough to make several kinds of distinct analysis on different kinds of software repositories.

That is precisely what we have addressed with GlueTheos: to design a system with an architecture which allows in a way as general and flexible as possible the data retrieval and analysis of public software development data repositories. Currently it can access CVS repositories and archives of source packages (both in deb and rpm formats), but others (such as bug tracking systems and mailing lists) are being included soon in the system. It is designed in a highly modularized way, so that adding new retrieval methods (from CVS or other data repositories) and analysis procedures is simple.

2 The GlueTheos system

The structure of the GlueTheos system is simple. Around a core of coordination scripts, there are input modules which download raw data (currently source code) from the repositories where it resides, modules which analyze such code from several points of view (counting lines of code or identifying authorship information), modules which store the information obtained in the previous phase (as a set of XML files or in an SQL database, for instance) and modules which produce the final reports (tables with data, graphs, etc.)

In the rest of this section, all those modules will be discussed in more detail:

- Core scripts. These scripts are the usual interface for users. With the help of a configuration file, they decide which repository is to be used, and which downloading, analyzing, storage and reporting modules will be run for it, according to the characteristics of the repository, the kind of intermediate storage desired and the final reports wanted. The configuration file can also be used to determine, for instance, that periodic snapshots from a CVS are to be retrieved, to study the evolution of a project. Or to perform only some stages of the whole process (like, for instance, downloading, analyzing and storing results, skipping the reporting phase which could be done later).

- Downloading modules. For each kind of repository, a downloading module is available. Currently, there are three: one for accessing CVS repositories, another for storages of RPM source packages (and in particular, those found in Red Hat Linux distributions), and yet another for Debian repositories (with source packages in the deb format). Those modules are capable of downloading the source code, unpackaging it if necessary (for instance, in the case of source packages), and having it ready for the next stage (which usually is the analysis of the code).

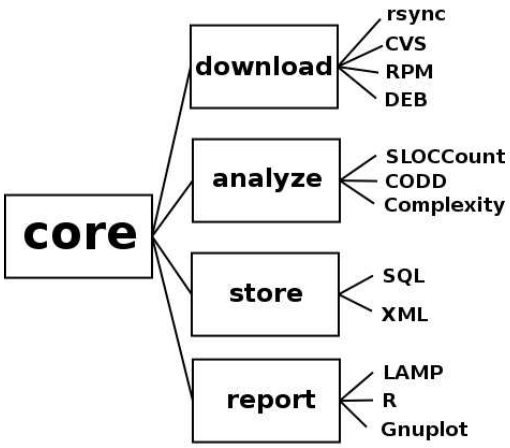


Figure 1. The Gluetheos modules

- Analyzing modules. For this stage, mostly external tools are used, like SLOCCount [17, 16], CODD [2], tools for metrics estimation (which include algorithms to calculate Halstead's[8] and McCabe's[12] complexity measures), raw count of file sizes (using for instance the wc utility), and others. Therefore, these modules are mainly drivers for those tools. Usually, they run the specific external program they drive, and produce results in a given data directory, in the output format used by that program.

- Storage modules. For making it simple the generation of reports, the information has to be in an easy to query storage. In addition, exchange formats have to be defined when information is to be moved or disseminated for study by other groups or at other locations. Currently, for most analyzing modules we have two storage modules, one generating XML files and other using SQL commands to feed a database. The first one is mainly intended for data interchange, while the other is better used for querying in the final stage. Now, we are moving to an architecture where there are only SQL modules for each analyzing module, and an

[2]The 12 biggest projects in size in Debian 3.0 use a software repository, all of them CVS besides Linux which uses BitKeeper, a proprietary solution

SQL to XML translator (also dependent on the analyzing module used, but much more simple).

- Reporting modules. These modules are the producers of the outputs of the system. They usually query the database, and massage the obtained information to produce tables, statistical analysis or graphs. For doing their work, in many cases those modules use also external tools, such as Ploticus or Gnuplot for generating graphs, or R for statistical analysis. There are also reporting modules which generate information in a format suitable for being browsed via web, with the help of some PHP code (LAMP = Linux + Apache + MySQL + PHP).

Currently, GlueTheos is written in Python, using Python standard libraries to access SQL databases, to generate XML files or to interface with other tools.

3 Some examples of use

The GlueTheos system has been used in the two following cases (which may serve as examples illustrating its capabilities):

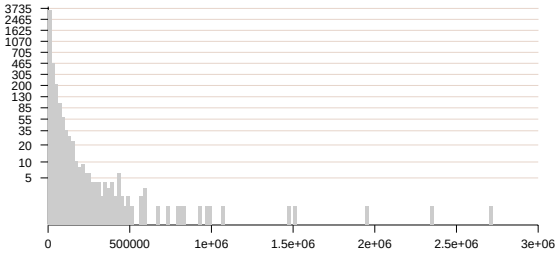


Figure 2. Histogram with the SLOC distribution for packages in Debian 2.0

- The study of the packages in the Debian GNU/Linux[7] and Red Hat Linux[17], distributions (Figure 1 and Figure 2). For this study, GlueTheos downloaded several Debian and Red Hat distributions, and analyzed the resulting source code by using SLOCCount to count its lines in several ways, like SLOC (source lines of code) per package or SLOC per programming language.
- The construction of the website `http://libresoft.dat.escet.urjc.es`. This site includes information and results about several libre software projects, from several points of view gained with the aforementioned analysis and measurement tools GlueTheos makes use of. Most of the information available publicly there has been built with the help of GlueTheos. One of the goals of this website is to offer the libre (free, open source) community the possibility to obtain feedback from our research.

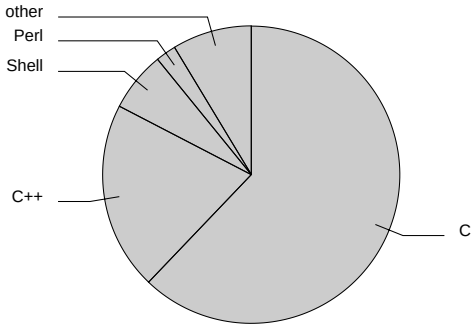


Figure 3. . Pie graph with the SLOC count for main languages in Red Hat 8.0 distribution

GlueTheos also provides an excellent opportunity for economists to measure the demonetized, previously invisible productivity of open source software projects, and also to analyze the organization and production methods of software at a level of detail probably unmatched by any other field of economic activity. This is because almost every single act of production, direct or indirect, is documented and recorded somewhere in the open source development process, much of which is captured and quantified by GlueTheos, which already pays attention to economic measures (such as the Gini[5] coefficient of concentration).

Although this sort of measurement may not, initially, be in monetary terms, it does represent human time and effort spent on productive activity, and can be "remonetized" at least for the purposes of measurement. One use for this may be to improve models for cost estimation of software development, by correlating time spent as reported by individual developers in surveys, with their productivity as determined through the examination of source code and related metadata (such as CVS).

4 Conclusions and further work

The GlueTheos system is an attempt to build a set of tools capable of automating most of the tasks related to the

analysis of publicly available information about libre software projects. Currently, it can access CVS repositories and archives of some GNU/Linux distributions. By using external tools it can make several different analysis on the fetched data, and produce several kinds of reports (from tables with organized data to graphs or information suitable for being offered in a website. GlueTheos pretends to fill the gap that exists for in-depth, fully-automated analysis.

Our group is working currently in stabilizing the system, making it more versatile (including more downloading, analyzing and reporting modules), and exploring data formats for the exchange of information about libre software projects. We are planning also to put a big effort in the reporting modules, so that information from different sources can be integrated and correlated giving a wider picture than the one that a unique tool may offer. Special attention is being given in showing the huge amount of data in a way that it is comprehensible avoiding the problem of information overload that is common in these scenarios.

Future plans also include to set up an interactive website where libre software developers can request their projects to be analyzed. Developers would have only to fill out a form where the location of the publicly available data sources should be specified and the system will automatically retrieve and analyze them, putting up a web-sites with the results and finally notifying the developers that they can see results there.

All the GlueTheos system, and the external tools it uses, are libre (free, open source) software.

References

[1] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[2] Codd website. `http://codd.berlios.de/`.

[3] D. Germán and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, Portland, Oregon, 2003.

[4] R. A. Ghosh. Clustering and dependencies in free/open source software development: Methodology and preliminary analysis. In *Open Source Workshop*, Toulouse, France, June 2002.

[5] C. Gini. *On the Measure of Concentration with Espacial Reference to Income and Wealth*. Cowles Commission, 1936.

[6] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. Oct. 2000.

[7] J. M. González-Barahona, M. A. Ortuño Pérez, P. de las Heras Quirós, J. Centeno González, and V. Matellán Olivera. Counting potatoes: The size of Debian 2.2. *Upgrade Magazine*, II(6):60–66, Dec. 2001. `http://people.debian.org/~jgb/debian-counting/counting-potatoes/`.

[8] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, USA, 1977.

[9] K. Healy and A. Schussman. The ecology of open-source software development. Technical report, University of Arizona, USA, Jan. 2003. `http://opensource.mit.edu/papers/healyschussman.pdf`.

[10] S. Koch and G. Schneider. Results from software engineering research into open source development projects using public data. *Diskussionspapiere zum Tätigkeitsfeld Informationsverarbeitung und Informationswirtschaft*, (22), 2000. `http://wwwai.wu-wien.ac.at/~koch/forschung/sw-eng/wp22.pdf`.

[11] M. Lehman, J. Ramil, P. Wernick, and D. Perry. Metrics and laws of software evolution - the nineties view. 1997.

[12] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 1976.

[13] A. Mockus, R. Fielding, and J. Herbsleb. A case study of open source software development: The Apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 263–272, Limerick, Ireland, 2000.

[14] E. S. Raymond. The cathedral and the bazar. *First Monday*, 1997. `http://www.firstmonday.dk/issues/issue3\_3/raymond/`.

[15] G. Robles-Martinez, J. M. Gonzalez-Barahona, J. Centeno-Gonzalez, V. Matellan-Olivera, and L. Rodero-Merino. Studying the evolution of libre software projects using publicly available data. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, pages 111–115, Portland, Oregon, 2003.

[16] Sloccount. `http://www.dwheeler.com/sloccount/`.

[17] D. A. Wheeler. More than a gigabuck: Estimating gnu/linux's size, June 2001. `http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html`.

Using CVS Historical Information to Understand How Students Develop Software

Ying Liu, Eleni Stroulia, Kenny Wong
University of Alberta
Edmonton, Alberta, Canada
{yingl, stroulia, kenw}@cs.ualberta.ca

Daniel German
University of Victoria
Victoria, BC, Canada
dmg@cs.uvic.ca

Abstract

Software engineering courses are expected to teach students a wide range of knowledge and skills, e.g. software-development methodologies, tools, work habits, collaboration skills, a good sense of scheduling, etc. In this paper, we present a method to track the progress of students developing a term project, using the historical information stored in their CVS repository. This information is analyzed and presented to the instructor in a variety of forms. The goal of this analysis is, first, to understand how students interact, and second, to find out if there is any correlation between their grades and the nature of their collaboration. Understanding these factors will enable instructors to detect potential problems early in the course of the students' projects, so they can concentrate their help on those teams who need it the most.

1. Introduction

During an undergraduate software-engineering education, students are expected to become knowledgeable in advanced software-development methods, acquire good time-management habits, and learn effective collaboration skills. In this paper, we discuss a set of analyses that support the monitoring of student teams and their progress, based on information collected from their CVS [5] project repositories. These analyses infer a multi-perspective trail of the project development, and a set of corresponding visualizations presents various statistics, diagrams, and reports on this trail. Based on the information produced, instructors can track the evolution of a team's work against other teams or compare the performance of members within a team. Furthermore, instructors can inspect the revisions to an individual file or the modification requests of the project. (A modification request or MR is a set of revisions that is considered atomic and results from a CVS commit command [1]). We believe that if this information is suitably presented and highlighted, it could also be useful to students to self-evaluate their own progress.

This paper begins with a description of the directly collected and derived data examined by our method. Section 3 discusses our case study and highlights some initial experimental results. Section 4 outlines the related work. Section 5 concludes the paper and discusses some future work.

2. Collected data

Useful data is captured in the CVS logs. Examining the information implicit within them about development processes and software trails is crucial for monitoring and managing a software project.

2.1 Directly collected data

A substantial amount of information can be extracted by examining the data directly collected by the CVS repository. Trends in this data can be inferred and presented through diagrams or reports, leading to meaningful insights regarding the development of the team projects.

2.1.1 The *team* level

With the same project requirements, comparisons across teams are very useful for instructors to monitor the performance of the whole class, and quickly notice unusual trends and events that might signify problems. For each team, we record the following parameters: total number of files, total number of Java files, total number of MRs, total number of revisions, average CVS operation distribution by type [2] and date, average work days, average work days on Java files, the proportion of MR size, and so on. Diagrams based on these parameters are introduced in the next section.

2.1.2 The *individual-developer* level

Within a particular team, we need to look into each member's contribution, and suggest adjustments if necessary. We also ask each student to complete a questionnaire to describe their own contributions and what they perceive as the contributions of their team members. For each member, the following parameters are gathered: number of total CVS operations (of each type), number of modified files, number of modified Java files, number of added files, number of added Java files, and number of Java files last modified, total number of added and deleted lines of code (LOC), total number of work days, total number of work days on Java files, the first checkout date, the first file add date, the first file modification date, the last file modification date, the last operation date, self and peer assessment questionnaire data, and scores achieved on the project stages. Using this data, instructors and team members can become aware of

the work habits of individual students, their workload, and any problems that need to be addressed.

2.1.3 The *file* level

The above parameters enable the comparative analysis of individual students' work. To discover potential problems with the project design and the task division, more data about the project files themselves is relevant. For example, two potential problems may be files that have a high occurrence of colliding changes, or files that end up being modified by multiple members. For each file, the following parameters are gathered: final size, log size, number of revisions, number of individuals who modified it, total numbers of operations of each type, and added and deleted LOC of each member.

2.2 Derived data

In addition to the information captured in the CVS history, there still exist underlying relationships between the team members' work habits, their roles, their main tasks, the project-design structure, and project schedule which are implicit in this data. Examples of further analyses that could shed some light on the above relationships are the following: the proportion of each student's idle days to project duration, the proportion of a student's leading idle days (between the start of the project and the student's first CVS operation) to the entire project duration, the proportion of the tailing idle days (between the student's last CVS operation and the end of the project) to the entire project duration, the proportion of Java files to the whole project, and the proportion of various types of CVS operations on Java files to all CVS operations.

3. The case study

To evaluate the usefulness of the information discussed above as a means of monitoring project progress, we conducted a case study, in the context of a third-year software engineering course, in which students work on a project in teams of four. This work was originally done in the context of our JRefleX project [3,4] (http://www.cs.ualberta.ca/~stroulia/JRefleX). Here, we further analyze the collected data to include information collected at the MR level [1] and compare the inferences of our analyses against the students' own understanding of their project work.

In the context of this course, students coordinate their software changes using CVS. The project is common across all teams, with three delivery dates spanning a two-month development period. Although various deliverables are required, including unit test cases, UML diagrams, and a user manual, we initially focus our analysis on changes to the source code over time. Detailed information can be seen in [2].

This section introduces selected charts and statistics generated in our case study on five student teams (labeled A to E). Diagrams can be presented at various levels: by course, by team, by individual, or by file. Such diagrams intuitively show trends, enabling the users to gain a high-level impression of team and individual performance. One aim is to notice anomalies, such as delayed or unbalanced workloads. Statistical reports list the data gathered in our database for more detailed inspection.

3.1 MRs and CVS operations for all teams

Figure 1 shows the number of MRs over time. Figure 2 compares the average number of CVS operations by type across teams. A comparison of team operation numbers across time can also be generated, although not shown here for the sake of space.

The aim of these diagrams is to compare the various work habits of the student groups. How fast do they start? How long is their actual development process? How many idle days do they have? How many files do they work on at a time? What proportion of files are Java? What is the distribution of their CVS operations? Considering these diagrams, we observe that group D has the most CVS operations (such as file additions and modifications), has more regular workload habits than the other teams, and has a medium number of MRs. Groups B, C, and E have sharp peaks around each delivery deadline, preceded by long idle periods. Group B has the most MRs at the second deadline. Group C has the smallest number of MRs.

3.2 CVS operations and file-related information for individuals

For a specific team, three kinds of diagrams and two kinds of reports can be generated. One diagram (see Figure 3) compares across team members the number of CVS operations over time. A second diagram compares across team members the numbers of CVS operations of each type (see Figure 4 for group D). A third diagram displays, for all the files by a given team, the proportion of added and deleted lines of code by the team members (see Figure 5). The daily report lists all the history logs in the CVS repository by date chronologically, while the individual report shows all CVS operations by each member chronologically.

The purpose of these diagrams and reports is to compare workloads and work habits of individuals within a team. For example, we want to discover who contributes what in which role, such as whether there are students who write stubs to be filled in by others. Are certain members focused on testing and debugging? Or, are all members assigned a full variety of development tasks?

3.3 CVS operations and modifications at the file and revision level

Two file-level diagrams (one shown in Figure 5) and one file-level report are intended to display the information related to all the files being worked on by a team. Users can notice the high collision files and files modified by multiple members quickly in the diagrams. The file-level report lists the detailed history of CVS operations on a

file in time order. Many questions are relevant at this level. What are the number and types of files that a team has? How many files are there? What is the proportion of heavily modified files? How many times is a file modified or touched? Who is the author of a file? Is the author the last person to modify a given file? How does the number of files evolve over time? What is the distribution of CVS operations on each file? Which files belong to the core of the software? Which files should be in an independent module?

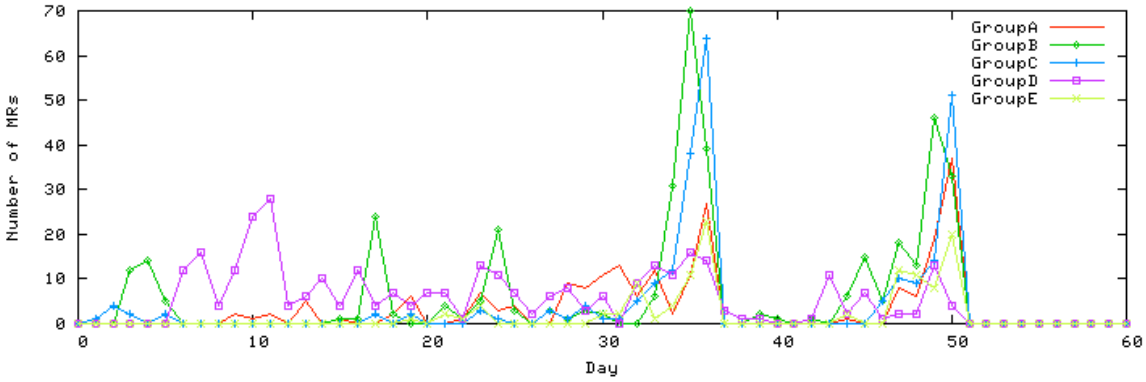


Figure 1: Modification requests of the teams

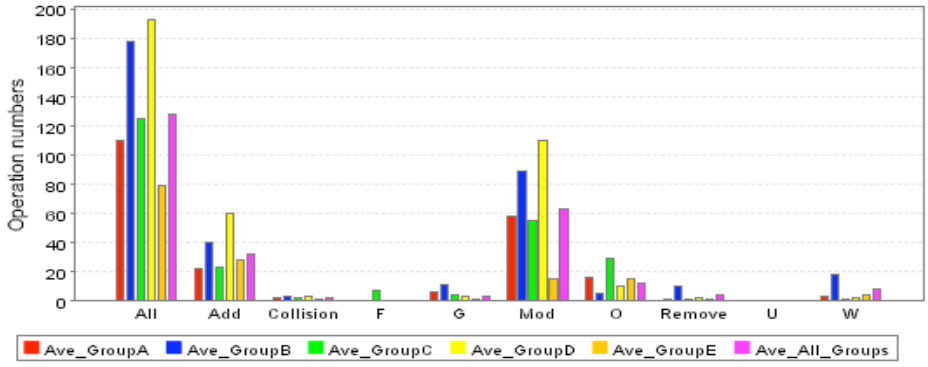


Figure 2: Numbers of operations of all types for the teams

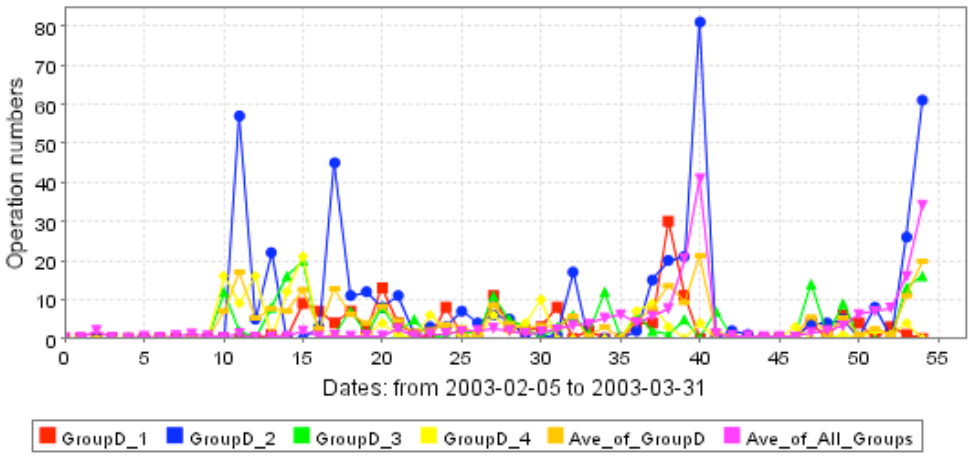


Figure 3: Total numbers of operations of Group D members over time

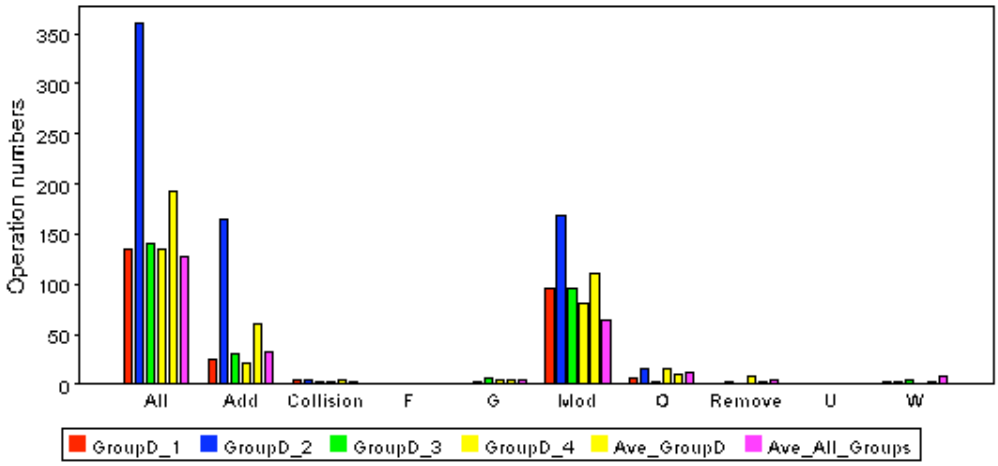


Figure 4: Numbers of operations of all types of Group D members

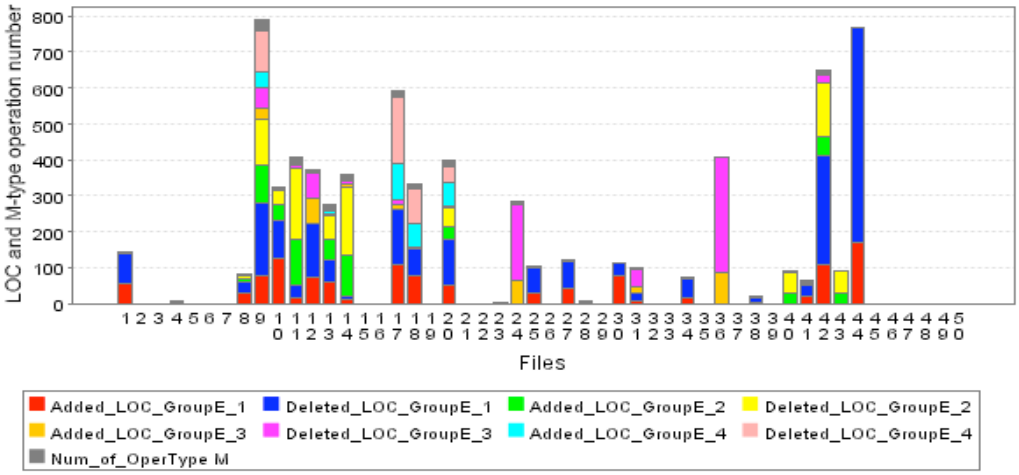


Figure 5: The files of Group E and their revisions

From such diagrams (some omitted here for brevity), we notice that the four members in Groups D and E exhibit different work habits. Group D members started earlier and have a larger proportion of working days. In contrast, all the members in Group E only start work just before the deadlines, with long silent periods according to their CVS repository. Furthermore, Group E had very few revisions per file, suggesting that they may have done most of their work at the very last moment. Student 2 dominated the CVS activity in Group D, with the CVS operations distributed to almost every day of the whole project except the initial planning time. He created many Java files for later completion, and he completed some of them without involving any other members. Although Group D has better habits, their project has an abnormally high number of Java files, with only very few of them being modified. After inspecting the file report, two reasons were found: all the members moved their individual assignment work into the common project directory, and Student 2 dumped another 37 Java files of his own into a directory named "demo/newLayout" on Feb. 22, and never touched them any more.

For group E, although Student 2 started much later, he still ended up adding and changing the most Java files, and took over some Java files from his teammates, being the last to check in most files. Student 3 is an early bird with the least total CVS operations, and the smallest number of Java files related to him, but his work is independent. Student 4 always works hard just before the deadlines, with most of his operations in adding new, complete files. However, his proportion of Java files is very small. These symptoms of poor habits can be correlated to a poor software design as well.

3.4 The students' view

At each deadline, we required each student to submit a self and peer-evaluation for his teammates. Looking at

their responses, we found substantial evidence in support of some of our conjectures in this study.

All four members of Group D had good feelings about their progress. Here are some examples of student comments: “because of my group members work ethics in being determined to start early, work regularly, and keeping each other updated on one another's progress”, “communication was open and constant via ICQ and email”, “each member was more than willing, if not enthusiastic, to contribute and participate”, “I was very impressed with other members’ willingness to help other members with problems in their ‘assigned’ areas”. In addition, we got some explicit validation for Student 2’s outstanding performance: Student 1 assessed him as “Student 2 did a lot of work with the coding (especially the interface design)”, and Student 4 gave such an evaluation: “very impressed with the effort that Student 2 and Student 1 put into the GUI”.

Group E also had their own collaboration feedback: most members felt “ok” at beginning although their team had some member changes. However, in their project part 2 report, a lot of problems appeared: “some confusion as to who was doing what. Some parts were done out of order so we couldn't do our part until all this other stuff was built”, “some miscommunication of what the plan was”, “concentrated largely on the front-end and the back-end was poorly formed and probably will have to be redone for the next part of the project”. Student 1 complained about the uneven workloads, and in part 3, he said that some teammates “reverted to the old ways of the computer geek” implying a substantial last minute effort. Finally, Student 4 expressed his feeling that “it is better to underestimate yourself than to overestimate”. Changes in the team members was one of the reasons for Student 2’s late start; in the end, he had a lot of Java file operations because he “played a key role in trying to integrate the front-end and backend as well as integrating other classes”.

From the above students’ comments, we can infer that the analyses we discussed here are both valid and potentially effective in noticing problems in a timely way. With the associated simple and easy-to-read visualizations, we are confident that we can help instructors and students to understand their development process better and detect symptoms earlier.

4. Related work

Various CVS repository analysis tools currently exist, such as CVSAnalY [6], CVSMonitor [7], CVSPlot [8], and CVSStat [9]. They collect and present statistical information from CVS files and logs. Most of these tools are mainly suitable for open source projects that span a long period of time. They focus on component size, file revisions, work effort measured in LOC, and all of them can present trends over time with a selectable time granularity. Our research delves more into the types of CVS operations, a wide variety of collected data parameters, and the use of KDD techniques. Ultimately, the goal is to provide a rapid-feedback, process-mentoring tool for novice developers in a small-team environment. The various diagrams we produce are more aimed to assist users in locating poor work habits or work imbalances. Besides having instructors monitoring the teams, the teams themselves can see and reflect on their own progress.

5. Conclusions

In this paper, we described our purpose for analyzing CVS repository histories, outlined the various data parameters recorded in our database, and introduced selected charts and reports for use by instructors and students. Some of the diagrams have been embedded into the Eclipse environment as a plugin. Indeed, an entire environment for process and design mentoring, called JReflex [4] is being developed. Future work consists of polishing this environment, implementing more case studies, and accumulating more data for KDD analysis.

References

[1] D. M. German, *Using software trails to rebuild the evolution of software*, Proceedings of the International Workshop on Evolution of Large-scale Industrial Software Applications (ELISA), September 2003.

[2] J. Han, M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, August 2000.

[3] Y. Liu, E. Stroulia, *Reverse engineering the process of small novice software teams*, 10th Working Conference on Reverse Engineering, November 13–16, 2003, pp. 102–112, IEEE Press.

[4] K. Wong, W. Blanchet, Y. Liu, C. Schofield, E. Stroulia, Z. Xing, *JRefleX: towards supporting small student software teams*, OOPSLA Workshop on Eclipse Technology eXchange 2003, pp. 50–54.

[5] CVS, http://www.cvshome.org

[6] CVSAnalY, http://barba.dat.escet.urjc.es/index.php?menu=Tools&Tools=CVSAnalY

[7] CVSMonitor, http://ali.as/devel/cvsmonitor/

[8] CVSPlot, http://sourceforge.net/projects/cvsplot

[9] CVSStat, http://www.gnu.org/directory/all/cvsstat.html

Database Techniques for the Analysis and Exploration of Software Repositories

Omar Alonso[1], Premkumar T. Devanbu, Michael Gertz
Department of Computer Science
University of California at Davis
oralonso@ucdavis.edu, devanbu@cs.ucdavis.edu, gertz@cs.ucdavis.edu

Abstract

In a typical software engineering project, there is a large and diverse body of documents that a development team produces, including requirement documents, specifications, designs, code, and bug reports. Documents typically have different formats and are managed in several repositories. The heterogeneity among document formats and the diversity of repositories make it often not feasible to query and explore the repositories in an integrated and transparent fashion during the different phases of the software development process.

In this paper, we present a framework for the analysis and exploration of software repositories. Our approach applies database techniques to integrate and manage different documents produced by a team. Tools that exploit the database functionality then allow for the processing of complex queries against a document collection to extract trends and analyze correlations, which provide important insights into the software development and maintenance process.

We present a prototype implementation using the Apache Web-server project as a case study.

1. Introduction

In a typical software engineering project, there is a large and diverse body of assets, usually documents that the development team produces over time. Those assets include requirement documents, specifications, design charts, code, bug reports, user manuals, etc. Different people create and update those heterogeneous types of documents that typically reside in different repositories. At different phases in a software development process, it would be useful to pose queries against these repositories. For example, one might ask about the status of a certain component, whether specified requirements have been met, or the amount of time that it took to fix a bug. The diversity of the data models and repositories from which these answers must be drawn complicates the task of processing such queries. This naturally augments the learning curve for new members of the team. It also makes it difficult to derive metrics and get insight into the software process.

In this paper, we present a framework for the analysis and exploration of software repositories. Our approach provides database techniques for the integration, processing, and management of different types of documents produced by the team over time.

Data integration enables us to manipulate different data sources within a single conceptual framework. An off-the-shelf query language and associated optimization and evaluation tools let us process the data to answer complex queries, even analytical questions to extract trends and correlations. The manageability of data is inherent to the use of a full-fledged modern database system.

Our contribution is a framework for incremental analysis and exploration of different data sources. Each data source is unique in certain ways so different techniques have to be applied to extract meaningful information. Finally, the combination of different data sources and extracted information within the framework allows the mining of the repositories.

We present a case study using the development mailing list of the Apache Web server project [1]. This is the main communication vehicle between the members of the team [2], [3]. We implemented a prototype that shows our ideas in practice using a commercial database product.

Current approaches on mining software repositories consist on ad-hoc scripts tailored to a particular data source. Those scripts manipulate the data source in the file system and produce metrics [5], [7], [12]. More recently, some research is starting to consider either database or information retrieval techniques to aid the manipulation of repositories [6], [8]. Finally, the social aspect is also an important component of the mining

[1] The author is also affiliated with Oracle Corp.

process like the identification of expert knowledge in a team [11].

2. Integration

In any particular software project the content arrives from different data sources in different formats. These sources can include project documentation using word processing template formats, web pages with miscellaneous information in different websites, email messages about project communication, reports from bug databases, etc. The content, except source code, is mainly free text with some rudimentary level of structure, if any. The idea is then to apply some structuring process to those data sources to produce more standardized data that would be easier to load and manipulate in a database. Once the content is in the database we can use all the existing techniques available to work on the sources like SQL to run queries, statistics packages to run analytics and so on.

A very arduous and time-consuming task is to extract information from a data source, modify it, and finally load it into the database. This step involves knowing the structure of the data source, the information that we want to extract, and the metadata that is possible to derive in the same process. For a web page, this implies parsing HTML, extracting the text of the document and metadata such as anchor text and other tags. In an email message, the main source of data is obviously the message but we also consider date, subject, and author to be important metadata.

Ultimately the goal is to transform the data extracted from a source into a more standard format that later will be loaded into the database using an existing tool. The final step is to generate all this information for a particular loading tool. Once this step is completed, the data is already in the database and it is possible to start the analysis phase.

In our case study, the integration is fairly straightforward and involves just one data source: email messages.

3. Analysis and Exploration

Our methodology emphasizes on databases techniques for managing the contents of all data sources. There are several reasons why we want to use databases. They provide the basic infrastructure for large existing information systems in a wide range of application domains. Today's databases are more than just tables and SQL statements. They can manage multimedia content, and they provide functionality like back up, recovery, replication, partitioning, etc. All features that are useful for managing large-scale data collections. From a data management point of view, there are a number of benefits that we would like to take advantage of:

1. Uniform way to access data storage and other language access methods (JDBC, ODBC, SQL) are standards for managing data.
2. Access to advanced data management features (full text indexing, XML etc.): there is support for new technologies like XML and specialized indexes for managing text.
3. Analytics: most databases systems have built-in APIs for performing advanced statistical analysis that can be part of OLAP and data mining applications.

We would like to process the data sources (in this particular case an email data source), extract useful information and then load the content into an appropriate schema in the database. Instead of writing scripts to extract some data, we use standard languages like SQL or XPath to issue queries against the database. In the following, we discuss some types of queries that can pose, taking advantage of the capabilities we have just described.

In the software engineering domain there is a wide range of questions that we want to ask. We can partition the set in two ways: technical and social.

In the technical aspect we would like to know if a component has been tested, if the requirements for a feature are met, or if a bug was fixed to name a few.

In the social area, as we all know, software development involves people. Apart from productivity metrics like numbers of line/year per developer, it is also interesting to know who is the main owner of a piece of the system, who is more active in different phases, etc.

For both cases, one can trace people to assets (documents) once all this information is in the database. If we want to know who was the more productive developer we can issue a query for the number of code checked in for a particular person. Similar queries are possible to get the number of bugs open/fixed, etc.

At first this looks like information that should always be available at a glance but unfortunately, due to the heterogeneity of the data sources, it is not always the case. Without a proper database, basic facts are almost impossible to retrieve and correlate in an automated way.

So far, a query language can help when one is interested in information that we know it is possible to retrieve. For example, one might be interested in the number of developers in a project or the severity of new bugs after a release date. Those queries are typically reports on the status of the development process. The

argument for using a database here is that is easy to run any type of report.

Databases can also help in the discovery process. The first step is to add information retrieval to already existing data retrieval capabilities. Data retrieval, as presented earlier, returns objects with a clear defined condition. Information retrieval involves returning objects about a query and, because of its probabilistic nature, may contain minor errors or be ambiguous [9].

Full-text search over email messages is useful when we are looking if a particular topic has been covered. It is also possible to restrict the search for a particular time frame. In our email example, it is interesting to retrieve documents about a topic per year.

At this point we presented the functionality where a user can run a query against the database. It is also important to see the other side where the database can arrange content semi-automatically using statistical techniques.

There are different statistical techniques to discover data patterns. In our framework, we use clustering to discover grouping of documents [10]. In our case, we want to use clustering as an unsupervised classification of email messages that can give us an idea of what the communication of the development team is all about.

We believe that the analysis and exploration involves a number of techniques like query languages, information retrieval, and data mining, among others. Databases provide almost all those techniques in some sort of low-level interface. Making them work together to mine useful information is the challenge. In the following section, we present a prototype implementation of the ideas presented so far.

4. Prototype Implementation

This section describes Minero[2], a prototype for mining information from software repositories. There are two parts of the implementation: the backend and the user application. The backend consists of the schema creation, content and metadata extraction from email messages, and post-processing. The application is a web interface that has a number of features available like text searching, cluster browsing, and reporting, among others.

Minero runs on top of an Oracle 9.2 database with Solaris as the operating system. The database runs in one machine while the middle-tier resides in a separate one. The web application is a mix of PSP (PL/SQL Server Pages), PL/SQL, and JSP (Java Server Pages) code.

[2] www.db.cs.ucdavis.edu/minero

4.1. Backend

As mentioned earlier, there is a significant amount of work that needs to be done to extract information from a particular data source, making it available in a consistent format, and finally loading it into a database.

For the case of the Apache development mailing list, all email archives are available from the web [1]. Each archive consists of a large number of email messages. The first step is to run an email extractor script that parses the archive and creates a file per email plus all other metadata that is useful for later processing like author, date, etc. The extractor also generates a special file that a database utility will use to load all the content into the Oracle database. As part of the loading process there is the schema creation.

Now that the content is inside the database, we need to perform some post-processing tasks. The tasks range from simple clean up scripts to advanced information retrieval and mining operations using the Oracle Text API [4]. Figure 1 summarizes the loading and post-processing steps.

As a simple first step, a few SQL scripts group email messages by date and year. Now we can issue queries that can give us information about the traffic on the development list (stored in the table `dev_ml`) in a particular time frame.

```
select count(tk) from dev_ml
where mdate = '1995';
```

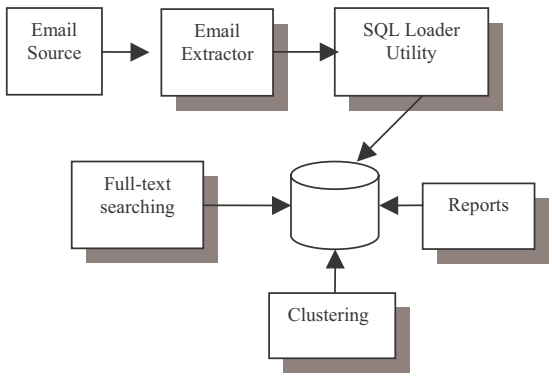


Figure 1. Extraction, loading, and post-processing of Email messages.

Moving into more advanced features, we want to make all the content searchable. For this, we create a couple of full text indices that will allow users to search

for emails about particular topics. Oracle provides an index type for full text searching that is available for querying using an extension of SQL. Because of the structure of an email message, we decided to create one text index on the subject only and a second one on the actual content. For example, to create a text index on a table we issue the following command:

```
create index devml_idx on dev_ml(text)
indextype is context
parameters ('filter null_filter');
```

A full-text search query for the term Java then has the following form:

```
select title, author, date
from dev_ml
where contains(text,'Java')>0;
```

The final step is to apply text mining techniques to the collection for identification of patterns. One way of doing that is to run a clustering algorithm that produces a flat partition of the content. The clusters will give us a "snapshot" of the content, and we can use them to refine searches, browse content, and produce a more accurate classification. The clusters can describe the areas of main interest in the mailing list. Minero uses a *k*-Mean clustering package that is also available with the database. Usually the running time of this cluster algorithm is $O(n*k*l)$ where n is the number of documents, k is the number of clusters, and l the number of iterations. For this prototype, we set $k = 200$ and $l = 6$. The output consists of tables that contain the description, metrics, and other data about the clusters and documents that belong to them.

4.2. Web application

We have built a front-end web application for users to explore and discover information from the mailing list. The front-end consists of a user interface where the user can search for email messages, browse clusters, run reports, identify main contributors to the list, and browse the entire collection by different attributes.

The user interface is based on a two-view model where the left side shows structure, and the right side, content of the collection of email messages. Minero presents the search results in two structures: a list of hits sorted by relevance score or a list of cluster descriptions that the user can open to see the hits. The right frame shows content (email messages) and message operations. The operations are: message with highlighted query terms, main themes, and gist (document summary).

Figure 2 shows a particular cluster about "processes, threads, MPM" and all the related email messages (documents). Again, users can view the email messages on the right frame and perform operations on them.

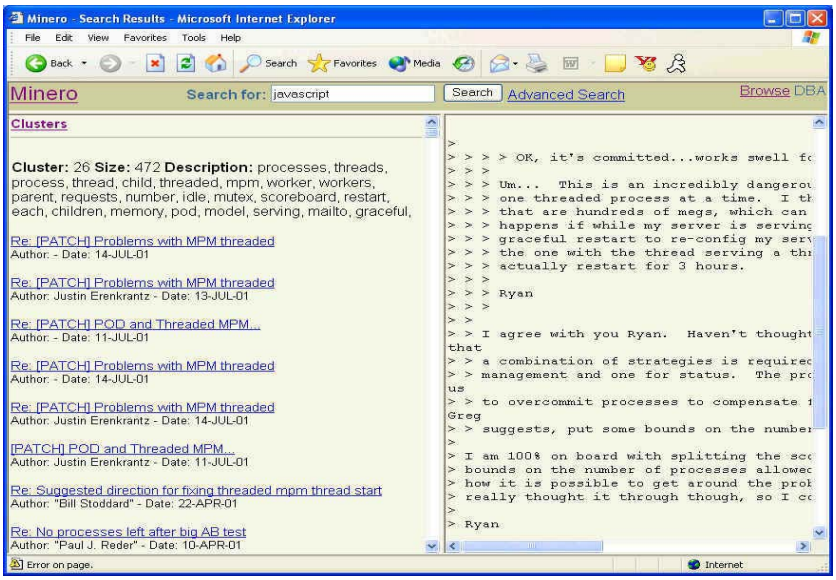


Figure 2. Using clusters to examine the Email collection

5. Results

The current database contains about 72,000 email messages. The content is searchable and can be restricted by date. The collection is partitioned into clusters that are available for browsing and discovery. There are a number of reports available for mailing list participation and overall message traffic per year.

Although our implementation is restricted to email messages only, we report similar findings to [12] in terms of the size and participation of the development community.

Going a little bit further and using time as a variable, we can report participation in the list per year using the following query:

```
select author, count(tk) cnt from dev_ml
where mdate = '1998'
group by author
order by cnt desc;
```

The past and present results are consistent with the public information about current and past team members [1]. We can also report that the main contributors have been doing so consistently over the life of the project.

Other results that we consider interesting from the software engineering perspective are as follows:

- We can identify two defined trends in the collection clustering. One is obviously technology. Examples are clusters about C/Unix, security, memory management, and threads. The second one is about process where the clusters are about releases, beta versions, patches, and the famous voting scheme for new features.
- As in any project, the early stages contain lots of process activities. Between the first and second year, there is a lot of traffic about the voting process, which later became stable.
- People change work places but continue participating on the list. This shows the commitment to certain open source projects.

6. Conclusions and Future Work

We presented a framework for the analysis and exploration of software repositories that relies on database techniques. We presented a prototype implementation of the ideas discussed so far. The project uses commercial database technology and it can be applied to other open source projects. We illustrated that a database is very convenient for this kind of projects. It allows us to define a schema where we can later perform queries or run more sophisticated mining techniques.

We were able to answer some software process questions like participation on the development list and also discover the technologies behind Apache using clustering.

We plan to continue working on Minero from the system perspective and on the overall methodology. We also plan to integrate the current schema with the source code repository and later the bug database so we can have a unified and integrated view of the project and the documents accompanying the project development.

7. References

[1] Apache Web server project http://httpd.apache.org

[2] R. Fielding and G. Kaiser, "The Apache HTTP Server Project". *IEEE Internet Computing*, 1(4), July/August 1997.

[3] R. Fielding "Shared Leadership in the Apache Project". *Comm. of the ACM.* Vol. 42, No. 4, April 1999.

[4] Oracle Text 9.2 Reference Guide (2003).

[5] D. German "Using software trails to rebuild the evolution of software", *Int. Workshop on Evolution of Large-scale Industrial Software Applications*. The Netherlands, September 2003.

[6] T. Zimmermann *et al*. "Mining Versions Histories to Guide Software Changes", *Proceedings of ICSE*, Scotland, UK, May 2004.

[7] A. Mockus, R. Fielding, and J. Herbsleb "Two Case Studies of Open Source Software Development: Apache and Mozill". *ACM TOSEM*, Vol. 11, No. 3 July 2002.

[8] S. Kawaguchi *et al* "Automatic Categorization Algorithm for Evolvable Software Archive", *Proceedings of IWPSE,* September 2003.

[9] R. Baeza-Yates and B. Ribeiro-Neto *Modern Information Retrieval*, Addison-Wesley (1999).

[10] A. Jain, M. Murty, and P. Flynn "Data Clustering: A Review". *ACM Computing Surveys*, Vol 31, No. 3, September 1999.

[11] A. Mockus and J. Herbsleb "Expertise Browser: A Quantitative Approach to Identifying Expertise", *Proceedings of ICSE,* Orlando FL. 2002.

[12] A. Mockus, R. Fielding, and Herbsleb "A Case Study of Open Source Software Development: The Apache Server", *Proceedings of ICSE* Limerick, Ireland, IEEE, 2000.

Empirical Project Monitor: A Tool for Mining Multiple Project Data

Masao Ohira[†], Reishi Yokomori[‡], Makoto Sakai[††],
Ken-ichi Matsumoto[†], Katsuro Inoue[‡], Koji Torii[†]
[†] Nara Institute of Science and Technology
ohira@empirical.jp, {matumoto, torii}@is.aist-nara.ac.jp
[‡] Graduate School of Information Science and Technology, Osaka University
{yokomori, inoue}@ist.osaka-u.ac.jp
[††] SRA Key Technology Laboratory, Inc.
sakai@sra.co.jp

Abstract

Project management for effective software process improvement must be achieved based on quantitative data. However, because data collection for measurement requires high costs and collaboration with developers, it is difficult to collect coherent, quantitative data continuously and to utilize the data for practicing software process improvement. In this paper, we describe Empirical Project Monitor (EPM) which automatically collects and measures data from three kinds of repositories in widely used software development support systems such as configuration management systems, mailing list managers and issue tracking systems. Providing integrated measurement results graphically, EPM helps developers/managers keep projects under control in real time.

1 Introduction

In software development in recent years, improvement of software process is increasingly gaining attention. Its practice in software organizations consists of repeatedly measuring the development activities, finding potential problems in the processes, assessing improvement plans, and providing feedback into the processes. Project management for effective software process improvement must be achieved based on quantitative data.

Many software measurement methods have been proposed to better understand, monitor, control, and predict software processes and products [4]. For instance, the Goal-Question-Metric (GQM) paradigm [2] provides a sophisticated measurement technique. GQM guides to set up measurement goals, create questions based on the goals, and determine measurement models and procedures based on the questions. The measurement based on GQM is a logical and reasonable method.

However, in its practice, members who participate in measurement activities need to strive for the measurement processes on every last detail. Data collection for measurement in general requires high costs and collaboration with developers. It is difficult to collect coherent, quantitative data continuously and moreover to utilize the collected data for practicing software process improvement. Few studies have proposed measurement tools for dealing with a number of project data especially in terms of a large-scale software organization.

As a measurement-based approach to the above issues, we have been studying empirical software engineering [1, 3] which evaluates various technologies and tools based on quantitative data obtained through actual use. Our goal is to develop an environment composed of a variety of tools for supporting measurement based software process improvement, which we call Empirical software Engineering Environment (ESEE).

In this paper, we introduce Empirical Project Monitor (EPM) as a partial implementation of ESEE, which automatically collects and measures quantitative data from three kinds of repositories in widely used software development support systems such as configuration management systems, mailing list managers and issue tracking systems. Collecting such the data in software development automatically and providing integrated measurement results graphically, EPM helps developers/managers keep their projects under control in real time.

2 Empirical Project Monitor (EPM)

We have developed Empirical Project Monitor (EPM) [9] which automatically collects and analyzes data from multi-

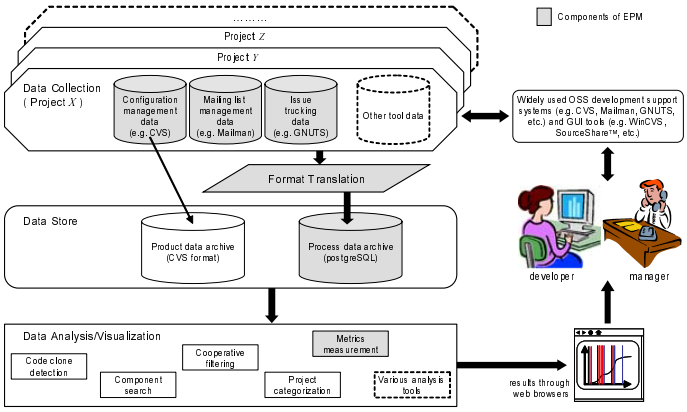


Figure 1. The architecture of EPM in the ESEE framework

ple software repositories. Figure 1 shows the architecture of EPM in the ESEE framework. The ESEE framework is designed for supporting measurement based process improvement in software organizations by providing various pluggable tools. EPM consists of four components according to the ESEE framework: data collection, format translation, data store, and data analysis/visualization. This section describes an overview of EPM and the basic data flow through EPM.

Automatic data collection: EPM automatically collects multiple project data from three kinds of repositories in widely used software development support systems. For instance, EPM collects versioning histories from configuration management systems (e.g. CVS[1]), mail archives from mailing list managers (e.g. Mailman[2], Majordomo[3], fml[4]), and issue tracking records from (bug) issue tracking systems (e.g. GNATS[5], Bugzilla[6]). Because these data are accumulated through everyday development activities using common GUI tools (e.g. SourceShare™[7], WinCVS[8]), developers/managers do not need additional work for data collection. Also, it dose not take high costs to introduce EPM into projects/organizations because the systems as the sources of data collection are open source freeware.

Format translation and data store: EPM converts the collected data into the XML format called the standardized empirical software engineering data, so that EPM can deal with not only the above three kinds of software repositories but also various kinds of repositories according to purposes for measurement. Data from other systems are available by small adjustments of parameters. The data converted into the XML format is stored in the PostgreSQL[9] database.

Analysis and visualization: EPM analyzes the data stored in the PostgreSQL database. For instance, in order to analyze data related to CVS, EPM extracts the process data about events such as checkin/checkout, transitions of source code size, version histories of components, and so forth. Then, EPM visualizes various measurement results such as the growth of lines of code and the relationship between checkin and checkout. EPM also provides summaries of each repository such as information of CVS logs. All the measurement results are available through using common web browsers (e.g. see Figure 2), so that users are easy to share the results.

In this way, EPM supports users to obtain quantitative data at low cost in real time and provides them with various measurement results for understanding the current development status. This would help users keep their projects under control.

3 Visualizations of measurement results

Data mining techniques for software repositories have been proposed to understand reasons of software changes [7], to identify how communication delay among developers in physically distributed environments have effects on software development [8], to detect potential software changes and incomplete changes [11], and so forth. In contrast to these tools, the features of EPM are to visualize

[1]CVS, http://www.cvshome.org/
[2]Mailman, http://www.list.org/
[3]Majordomo, http://www.greatcircle.com/majordomo/
[4]fml, http://www.fml.org/index.html.en
[5]GNATS, http://www.gnu.org/software/gnats/
[6]Bugzilla, http://www.bugzilla.org/
[7]SourceShare™, http://www.zeesource.net/
[8]WinCVS, http://wincvs.org/
[9]PostgreSQL, http://www.postgresql.org/

Figure 2. Measurement results through web browsers

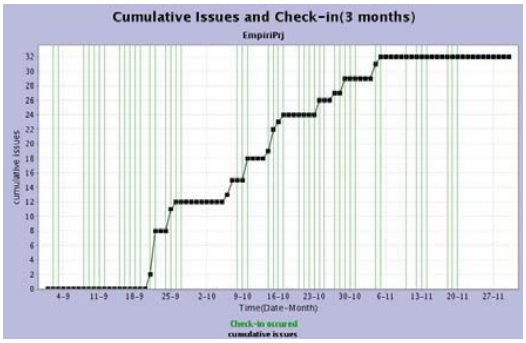


Figure 3. Relationship between issues and checkins

combinations of measurement results from three kinds of software repositories and to be able to deal with data from multiple projects simultaneously.

3.1 Combinations of measurement results

In addition to providing visualizations of measurement results from each software repository, EPM also visualizes combinations of measurement results from three kinds of repositories. The followings show two examples of them.

Bug issues and checkins: Figure 3 represents the relationship between the transition of the cumulative total of issues (the line graph) and the time of checkins (the grayed vertical lines on the X-axis) in our EASE project [6]. The number of issues and checkins are measured from the repository in GNATS and CVS respectively. A checkin often occurs after bug issues are reported because developers try to modify or resolve the issues. The graph helps users (developers/managers) remember the situation where issues by every file versions were raised. To the contrary, the file itself which is checked in CVS may include some bugs if the graph indicates that there are issues after checkins.

Bug issues and e-mails among developers: Figure 4 illustrates the communication history among developers in the EASE project. The black line graph is the transition of the cumulative total of e-mails exchanged through using Mailman. The vertical shorter/longer dashed lines represents when bug issues were raised/resolved. The light-gray vertical lines mean when the checked-in files by developers were uploaded to CVS. From the graph, users can confirm the state of the communication among developers and identify the file versions which might have problems. Because discussions on issues become active usually when issues are reported to an issue tracking system, the communication itself among developers might have problems if many issues are reported but developers did not discuss on the issues. Communication problems among developers bring the decrease of software productivity and reliability [8].

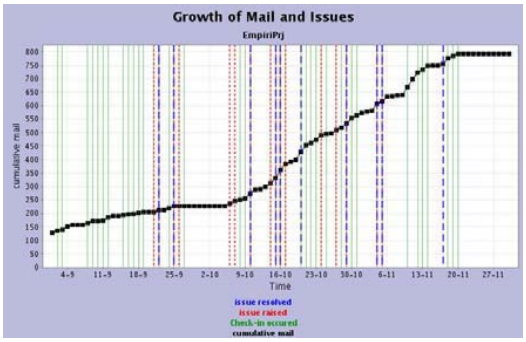


Figure 4. History of bug issues and e-mails among developers

The integrated measurement results based on data from configuration management systems, mailing list managers, and issue tracking systems help developers understand current and past events in development activities.

3.2 Visualizations of multiple project data

EPM has the capability to visualize multiple project data. Comparing current projects with past ones would be helpful for managers to estimate the progress of projects and to detect the unusual status in projects.

Comparison of measurement results among multiple projects: EPM makes measurement results comparable with multiple projects. Figure 5 represents the relationship of the growth of lines of code between two project (the upper line: SPARS [10], the lower line: EASE). The both projects have been proceeding under the collaborative research with authors' universities and some software companies. Some researchers and developers have been participating in the both projects. Actually although the both have different purposes and aspects, suppose here that they have been developing software systems respectively under similar conditions. The project managers can confirm some common characteristics and roughly estimate the progress of the later project (EASE) from the graph. For instance, SPARS has the two phases in which it have evolved rapidly for releasing major versions. EASE has just released the first major version. The managers are easy to guess the near future of the progress of EASE: the development of EPM will stop for a while to test the EPM, to reconsider the design, and so forth.

Figure 5. Comparison of two projects

Distribution maps of multiple projects: Using measurement results from three kinds of repositories in multiple projects, EPM can generate distribution maps. Figure 6 is a distribution map using 100 Open Source Software Development (OSSD) projects data collected from SourceForge.net[10,11], which represents the relationship between lines of code (the X-axis) and number of checkins (the Y-axis). Suppose here that these projects are managed by one software organization. The graph can be used for helping managers identify "unusual" projects which indicate extreme high or low values.

[10]SourceForge.net, http://sourceforge.net/

[11]We selected the 100 projects in Figure 6 randomly from the list of most active projects in SourceFroge.net.

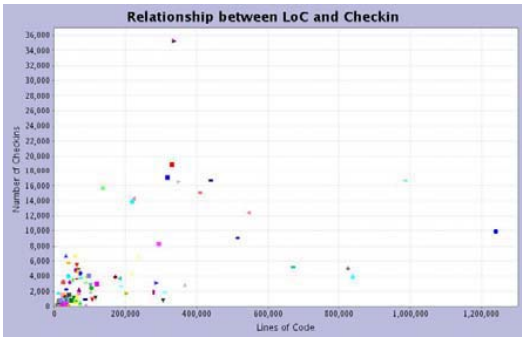


Figure 6. Distribution map of 100 OSSD projects

3.3 Customizations of measurement parameters

EPM currently provides users with the five types of graphs including Figure 3-5 and two types of summary information from CVS and mailing list data. Users have the choice of visualizing single project data or multiple project data according to purposes of analysis. EPM also provides an interface to customize queries for the PostgreSQL database. Using the database schema for EPM which is open to the public, users are able to input SQL sequences and to create bar graphs, line graphs, and distribution maps such as Figure 6. Because we would like to support various projects and organizations which have own problems respectively, we decided to provide the minimum types of graphs and summary information rather than to provide a lot of them in advance. After feedback from software organizations using EPM, we will add other types of graphs in the near future. Currently EPM can be viewed as a tool for exploratory data analysis [5].

4 Discussion

In this section, we report a case study of applying EPM to our project itself, in order to observe the actual usage of the pre-defined 5 types of graphs mentioned above. We have interviewed four developers on the advantages and the disadvantages of using EPM. The development environment of this project is summarized in Table 1.

One of the advantages is that the graphs make developers easy to understand the status of the project by identifying distinctive parts indicated in the graphs. For instance, the part of the flat line in the LoC graph reminded them why the development seemed to be stopped. In fact, all developers were on a business trip at the time. This could help them

Table 1. EPM development project

Target project	EPM development project
Programming language	Ruby, Java
Number of developers	4
Repositories	CVS, Mailman, GNATS
Development period	3 months
Preparation period	1 week

increase the accountability for their managers. Other one is that the graphs generated in real time motivated developers to fix bugs, since they could be aware that there were still unresolved issues.

In contrast to these advantages, some problems related to the usage of EPM have been found. One is that visualizations are too complicated to understand the status of the project in some cases. For instance, developers could not distinguish which file versions corresponded to which vertical lines in Figure 3, since one developer checked in CVS for backup of his files every day and therefore a number of checkins occurred. In this case, developers might need to use two CVS (e.g. one is for software release and another is for backup).

The above results are still the initial evaluations for EPM. EPM will be introduced in some software companies in the near future. We intend to evaluate the usefulness of EPM with respect to (1) the effects on software development and process improvement by providing measurement results from multiple software repositories, and (2) the benefit of giving the capability to manage multiple projects.

5 Conclusion and Future Work

The goal of this research is to construct an environment for supporting measurement based software development according to the ESEE framework. In this paper, we introduced Empirical Project Monitor (EPM) as a partial implementation of ESEE, which helps developers/managers keep projects under control by providing various visualizations of measurement results related to project activities. Nowadays, we can gather and analyze massive data on software development in a large scale using rapidly growing hardware capabilities. By analyzing such the huge data collected from thousands of software development projects, we would like to provide useful knowledge and benefit not only to individual developers/managers but also to organizations.

Empirical study on software development is an active area in the field of Empirical Software Engineering (ESE). But the approaches of ESE have not been sufficiently applied to software development in software industry although companies hold many problems. The data related to software development from the industrial world has seldom been provided with university's research. We are collaborating with some software development companies as the EASE project. Therefore, it would be a strong trigger for going beyond the obstacle of the technical progress in software engineering.

Acknowledgment

This work is supported by the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology. We thank Satoru Iwamura, Eiji Ono and Taira Shinkai for supporting the development of Empirical Project Monitor.

References

[1] A. Aurum, R. Jeffery, C. Wohlin, and M. Handzic. *Managing Software Engineering Knowledge*. Springer, Germany, 2003.

[2] V. Basili. *Goal Question Metric Paradigm, in Encyclopedia of Software Engineering (J. Marciniak ed.)*, pages 528–532. John Weily and Sons, 1994.

[3] V. Basili. The experimental software engineering group: A perspective. ICSE'00 award presentation, June 2000. Limerick, Ireland.

[4] L. Briand, C. Differding, and D. Rombach. Practical guidelines for measurement-based process improvement. Technical Report ISERN-96-05, Department of Computer Science, University of Kaiserslautern, Germany, 1996.

[5] S. Card, J. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan-Kaufmann Publishers, San Meteo, CA, 1999.

[6] EASE. The EASE (Empirical Approach to Software Engineering) project, http://www.empirical.jp/intex-e.html.

[7] D. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, pages 63–67, Portland, Oregon, 2003.

[8] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: Distance and speed. In *Proceedings of the 23rd international conference on Software engineering (ICSE'01)*, pages 81–90, Toronto, Canada, 2001.

[9] M. Ohira, R. Yokomori, M. Sakai, K. Matsumoto, K. Inoue, and K. Torii. Empirical project monitor: Automatic data collection and analysis toward software process improvement. In *Proceedings of 1st Workshop on Dependable Software System*, pages 141–150, Tokyo, Japan, 2004.

[10] SPARS. The SPARS (Software Product Archiving and Retrieving System) project, http://iip-lab.ics.es.osaka-u.ac.jp/SPARS/index.html.en.

[11] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering (ICSE'04)*, Edinburgh, Scotland, UK, 2004 (to appear).

System Understanding and Change Patterns

Mining Version Control Systems for FACs (Frequently Applied Changes)

Filip Van Rysselberghe and Serge Demeyer
Lab On Re-Engineering
University Of Antwerp
Middelheimlaan 1
filip.vanrysselberghe@ua.ac.be

Abstract

Today, programmers are forced to maintain a software system based on their gut feeling and experience. This paper makes an attempt to turn the software maintenance craft into a more disciplined activity, by mining for frequently applied changes in a version control system. Next to some initial results, we show how this technique allows to recover and study successful maintenance strategies, adopted for the redesign of long–lived systems.

1. Introduction

As stated by Lehman's 1st law of software evolution, a system has to undergo continuous change in order to remain satisfactory for its stakeholders [8]. Adding new features, correcting faults and accommodating for new changes, are generally considered the prime reasons for these changes [12].

Unfortunately, the changes applied during maintenance are seldom documented. Even for refactorings –currently the best known approach towards a systematic catalogue of maintenance tasks–, there is no indication which refactoring is most suitable for a certain situation. This observation can easily be illustrated by looking through Fowler [4], which is considered standard work on the subject and counting the number of conditional sentences. Knowledge on which changes are most appropriate for an occurring problem or situation is therefore private to experienced software maintainers.

By making this knowledge general, software maintenance can be improved and lose its status of ad hoc discipline. In order to meet this goal, we propose a technique analogue to the idea of frequently asked questions or FAQs. These FAQs are summaries of frequent questions and corresponding answers to reduce the continual posting of the same basic question. Analogue, we propose to identify frequently applied changes (FACs) since these changes record general solutions to frequent and recurring problems.

To detect such frequently applied changes, a technique based on clone detection is used. Due to their central position in modern development processes and their ability to record a project's entire change history, versioning systems contain a wealth of change information. Therefore the data for the detection process is provided by a versioning system.

In the remainder of this paper we will introduce the technique, evaluate it and position it in a broader context. The first section is reserved for introducing the technique (section 2). Afterwards, the results of an initial case study to evaluate the technique are discussed in section 3. Section 4 on the other hand, explores how the resulting change sets can be used to study software maintenance. Our future directions are discussed in section 5. The last section (6) discusses related work.

2. The Technique

With our technique we want to focus on how a system changed during its maintenance. Therefore we are interested in systems which are able to tell which changes were made. Typically, version or change management systems offer this functionality.

Such versioning systems like CVS, ClearCase and SourceSafe, can be considered as a large source code repository containing all the versions of a program. However internally, most versioning systems store the entire source of one version only. CVS for example, stores the last version entirely since that is the most likely version to be checked out for additional editing. Other versions are re–constructed by means of delta's, relative to the one complete version. For CVS, these delta's record which lines have to be added, deleted or changed in order to get a previous version. Since these delta's record which changes were made, we can ex-

tract change information from a versioning system.

In practice, extracting change information from a versioning system is not difficult as we found out by our initial study. For this study we targeted the CVS versioning system since it is used for many successful open–source projects, providing a lot of changes to study. Using proper CVS commands, change information can be extracted and afterwards processed. We combined the "cvs log" and "cvs diff" commands to extract change data like the difference in code before and after the change, the date and time of the change, the file involved etc. Since this basic change data is stored in about any versioning system, other versioning systems than CVS can be used as well.

Being able to extract change information from a versioning system is only part of the technique. Processing the changes in order to locate frequently applied changes is therefore the second step in our technique. However, first, we define a frequently applied change as a change to the code which occurs multiple times in the evolution of a system. Since a source code level is targeted, two applied changes are equal if there is a similarity relation between the delta of both changes. Locating frequently applied changes therefore corresponds to identifying sets of similar code fragments.

A possible approach to locate identical and similar code fragments is by using clone detection techniques. Clone detection techniques are developed to identify duplicated or cloned code fragments within a program since duplication may hinder the program's evolution. Over the years different techniques are proposed to locate clones or fragments which share the same code but may differ in the naming of identifiers. Ducasse for example, proposes a detection technique to locate clones containing a certain amount of identical lines [2]. Baker on the other hand focusses on code fragments in which identifiers, which are likely to change during the duplication process, may differ as long as there is a one to one mapping between the identifiers [1]. Since these techniques are developed to locate similar code fragments in a scalable way, we use them to identify frequently applied changes.

By using clone detection techniques on the changes extracted from the versioning system, we are therefore able to identify frequently applied changes.

In our initial study, we started by extracting from the repository all changes made during the lifetime of a product. For each change the corresponding delta, which consists of the code before and after the change, was added to one, general text-file. This text–file was later analyzed by Kamiya's clone detection tool CCFinder [7] to locate recurring, similar changes. Since a similar change recurs a couple of times in the change history, it correspond to a frequently applied change or FAC.

3. Evaluating the technique

To evaluate the technique, we applied it to study a successful open–source system called Tomcat. After more than three years of development, Tomcat is considered the standard Java servlet container. Ever since the beginning, participating developers can contribute to the project by accessing a central CVS versioning system. Therefore the CVS system contains a wealth of maintenance information under the form of changes. Combined with the knowledge that the project experienced some major redesign phases, Tomcat shows an interesting case to explore the techniques possibilities.

3.1. Finding refactorings using clone detection

In a previous study, we showed how clone detection can be used to detect refactorings between two versions [11]. We observed how scalability hinders the application of this technique since comparing all successive versions of a large system is not feasible. The scalability problem was a direct result of comparing whole systems rather than the changes made between two versions. Since the technique proposed in this paper focusses on the changes, it does not suffer this problem, yet is able to detect similar refactorings.

3.2. Finding FACs

The goal of our initial study was to explore the kind of frequent change sets that can be formed when a parameterized clone detection technique as CCFinder is used [7]. Due to their focus on the detection of similar code fragments, parameterized clone detection techniques are expected to produce the most suitable FACs.

CCFinder is a token based detection technique which searches a specially constructed tree for maximal matches. Due to its token based nature, the detection process is not influenced by the code layout. Crucial in its detection process is the definition of a length threshold. This threshold defines the minimal number of tokens that should match before a token sequence is considered a match. Matches which fail to meet the length threshold, are not considered matching fragments. Increasing the threshold, therefore makes the detection of both positive as false matches less likely. In our study, the impact of this parameter was taken into account by exploring various thresholds. Based on the visual representation of the detected matches, we manually verified the relation between repeatedly found changes. For these matching changes make up a set of frequently applied changes.

The sets of identical changes, constructed with a high threshold value, are small sets of long changes which are almost identical. Accidentally matching, long changes are

less likely when a high threshold value is used. Therefore the changes which are identified as (frequently) recurring changes are related to each other by a copy relation. As we observed in the initial study, there are three possible causes for such relation to exist:

- A first reason is the introduction of duplicated code. Based on former experience, the maintainer applies a previously used solution in a different location. In our study, we for example noticed how the same exception handling code was introduced in different places. However, the maintainers were aware of the problems related to duplicating code since a few versions later, the duplicated statements were replaced by a function.
- Repositioning a code fragment is a second cause for the introduction of copy related changes. Moving code from one class to another results in deleting and adding the same piece of code at various locations. In our study, we for example observed how an utility function of RequestUtil was moved into the Parameter class.
- Temporarily adding a code fragment, is the last cause we observed. We noticed how many of the fragments added in one version, are deleted later, causing a copy relation between the addition and delete. As figure 1 illustrates, many similarities between add and delete changes exist.

Figure 1. As each dot in the plot corresponds to a similarity between an add and a delete change of at least 40 tokens, it illustrates the relation between both categories

High threshold values therefore allow the identification of recurring, product specific changes. By establishing the motivation behind these changes more general maintenance conclusions can be derived. For example we might learn why code is duplicated or when a temporary solution can be suitable. Also note that we used a simple comparison scheme, comparing any part of the delta, with any other delta. A comparison scheme in which only the pre-change code of delta's is compared, would not find many of these copy related changes. For high threshold values, it is therefore better to use a simple comparison scheme which just compares anything.

Low threshold values on the other hand, lead to the identification of frequently applied, generic changes. Due to the reduced impact of the threshold on accidental matches, more changes which are only accidentally syntactically identical, are detected. This leads to sets of frequently applied changes which correspond to low level code changes as e.g. changing a function call or changing a variable. In our case study, we for example had a set of package definition changes, while another set bundled an extensive set of import statement changes. However before low threshold detection can be used in practice, two problems should be solved.

When no special precautions are taken, different kinds of generic changes are identified as one frequently applied change. Since clone detection only demands that the syntactical structure of two fragments matches, changes sharing an identical structure, yet differing in semantics, are wrongly classified as identical changes. Figure 2 which serves as an example, illustrates how renames of function calls and variables are structurally identical. This problem can be solved by taking more specific change information into consideration.

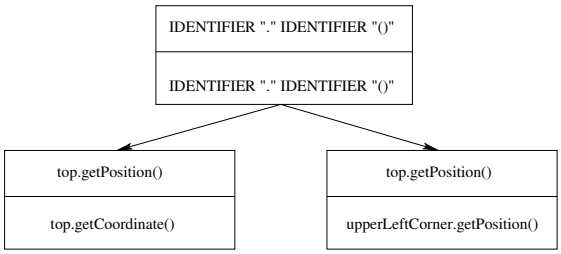


Figure 2. This figure shows how two generic changes can be identical from a syntactical perspective. In this example, renaming a function–call (left) and renaming a variable (right) share the same structure (above), however differ semantically. In the syntactic representation, non-terminals are quoted.

A similar problem was caused because many changes contain similar sub-units which match with each other. In our simple evaluation setting, all changes were added to one text-file which was afterwards analyzed for the presence of similarities. However this allows a statement, part of a larger change, to match a statement of an other change. To avoid this problem, the comparison of changes has to be

done in a more intelligent way. Identical changes are then no longer changes which just share a similarity, but changes in which both the code before the change as the one after the change are similar. In case of a low threshold detection process, more care has to be taken when comparing changes. The frequently applied changes are however more generic and can be used to automatically find generic maintenance strategies.

4. Study of frequent changes

Finding frequently applied changes on itself does not solve any maintenance issues. Each frequently applied change is rather a building block to identify generic maintenance strategies. Based on the kind of sets formed, different approaches can be taken.

Frequently applied changes identified with a high threshold and are therefore specific to one product, can be used as a starting point to study the motivation and success of a change. Afterwards this study may be generalized in a (set of) general rule. The motivation behind the removal of a recently added code fragment in a product, may teach us for example, why a change may fail in general. Similarly, studying the duplication of a solution, may point us to general problems or teach us how duplication grows, which in turn allows us to improve design patterns to cope with this duplication. However, due to their tight product coupling, these high threshold FACs can be used to improve and understand the current product as well.

The generic FACs found with a low threshold on the other hand, can be used to derive maintenance strategies automatically. In such automatic process, FACs are used as building blocks in a data mining process to identify frequent change patterns. All changes, classified according to their change set, are added to a kind of change transaction database. After composing such change database, a data mining process searches for frequent change patterns. However we can not just search all the changes for recurring patterns since change strategies are composed of both generic changes and relations between them. Therefore related changes, for example two changes which share an identical code fragment or because one change introduces a function used by the other, are considered as one change transaction. By comparing these various change transactions, recurring patterns are identified. These change patterns allow us to find generic maintenance operations like the refactorings that are currently described in literature [4]. By evaluating the situation in which these maintenance operations are used, we also might identify when and why they should be applied.

In turn, these maintenance operations can be considered as frequently applied changes and used to identify even higher level change strategies. One possibility is to redefine the data mining process to these higher level FACs. In this redefined process, a transaction groups all the change patterns applied in one version. Change patterns identified by such process may show us how a generic problem situation is removed by performing a sequence of lower level maintenance operations. Software maintainers can use this knowledge when handling a similar problem.

An other possibility is to relate changes with bug reports as proposed by Fischer [3]. By combining this information, frequently occurring bugs as well as their solutions may be identified. Consider for example a case where many bug reports are related to changes belonging to the “change loop condition” FAC. This would empirically show that looping conditions tend to lead to errors and therefore should be thoroughly tested. Similarly changes could be linked with maintenance reports or feature requests, to identify requests with similar solutions. This would not only allow maintainers to apply requests faster, but also helps designers to anticipate changes better.

Frequently applied changes, allow us to start with a set of small, generic changes and incrementally build up a whole set of maintenance operations which enrich our current software maintenance knowledge with wide-spread operations and techniques.

5. Future Directions

In the immediate future, we want to further explore the technique, introduced in this paper. This includes further evaluation of the suitability of various clone detection techniques and clone detection in general. For example, we want to study how the low threshold related problems (3.2) can be reduced or even removed.

Next to improving the technique through evaluation, we will apply it to study software evolution. In this context, we plan to evaluate the techniques and ideas presented in section 4. By means of these techniques, we want to investigate various projects to come to general maintenance strategies.

6. Related Work

To our knowledge, little effort has been spent to compose sets of frequently occurring changes and use those to study software maintenance. One could argue that a runtime change classification scheme as the one by Gustavsson [5], fits this context too since the items in such classification correspond to sets of changes. However FACs work on different levels and work from an exploratory perspective. Furthermore, the technique allows to automatically categorize changes based on any classification scheme.

In the context of studying changes, work has been done to identify the motivation for a change [12]. The goal was

verifying whether a change was made to fix a problem, prepare for future change or to insert new user functionality. Related with this work, Mockus used word frequency analysis of log messages to not only identify the purpose of changes, but relate it to change size and time between changes as well. Mockus and De Hondt, who both studied change log information, state that a textual description of a change is necessary to understand the real motivation behind a change [10, 6]. Our technique, does not differ from this point of view. Change information should be linked with other maintenance information to fully understand the motivation behind these strategies.

Concerning the detection of patterns using data mining, there is some relation to Michail [9] who detects reuse patterns based on a data mining approach. We propose a similar approach to detect change patterns over different versions.

References

[1] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Computing*, 26(5):1343–1362, October 1997.

[2] S. Ducasse, M. Rieger, and S. Demeyer. A language independent approach for detecting duplicated code. In H. Yang and L. White, editors, *Proceedings of Int. Conf. on Software Maintenance (ICSM)*, pages 109–118. IEEE Computer Society, September 1999.

[3] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings Int. Conf. on Software Maintenance (ICSM)*, pages 23–32. IEEE Computer Society, September 2003.

[4] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[5] J. Gustavsson. A classification of unanticipated runtime software changes in java. In *Proceedings Int. Conf. on Software Maintenance (ICSM)*, pages 4–12. IEEE Computer Society, September 2003.

[6] K. D. Hondt and P. Steyaert. Exploiting classification for software evolution. In *ECOOP 2000 Workshop on Objects and Classification*, 2000.

[7] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multi-linguistic token-based code clone detectionsystem for large scale source code. *IEEE Trans. Software Engineering*, 28(7):654–670, July 2002.

[8] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[9] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd Int. Conf. on Software Engineering*, pages 167–176. ACM Press, 2000.

[10] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings Int. Conf. on Software Maintenance (ICSM)*, pages 120–130. IEEE Computer Society, October 2000.

[11] F. V. Rysselberghe and S. Demeyer. Reconstruction of successful software evolution using clone detection. In M. W. G. Tommi Mikkonen and M. Saeki, editors, *Proceedings Int. Workshop on principles of software evolution (IWPSE)*, pages 126–130. IEEE Computer Society, September 2003.

[12] E. B. Swanson. The dimensions of maintenance. In *Proceedings of the 2nd Conf. On Software Engineering*, pages 492–497, 1976.

Mining the Software Change Repository of a Legacy Telephony System

Jelber Sayyad Shirabad, Timothy C. Lethbridge, Stan Matwin
School of Information Technology and Engineering
University of Ottawa, Ottawa, Ontario, K1N 6N5 Canada
{jsayyad,tcl,stan}@site.uottawa.ca)

Abstract

Ability to predict whether a change in one file may require a change in another can be extremely helpful to a software maintainer. Software change repositories store historic changes applied to a software system. They therefore inherently contain a wealth of information regarding (hidden) interactions between different components of the system, including the files that have changed together in the past. Data mining techniques can be employed to learn from this software change experience. We will report on our research into mining the software change repository of a legacy system to learn a relation that maps file pairs to a value indicating whether changing one may require a change in the other.

1. Introduction[1]

In large software systems there are many unknown or undocumented relationships and interactions between different components of the system. Such undocumented relationships are major sources of complexity and cost for maintaining the system.

Source code management systems, along with error tracking and change repositories maintain a comprehensive change history of a system. They inherently store a wealth of information regarding many of the interactions and relationships among different components of the system. Data mining methods convert data containing past experience with a given process into the knowledge about this process. Therefore, source code management systems are a fertile area of application for data mining. The idea here is to learn relationships among software entities from the historic change records.

In Section 2 of this paper we present the notion of Relevance Relations among entities in a system. Section 3 shows how the problem of learning a relevance relation can be mapped to a classification learning problem, while Section 4 describes the measures used to evaluate the quality of the learned classifiers.

As a proof of concept we will learn a relevance relation between file pairs in a legacy system. This relation maps a pair of files to a value indicating whether changing one may require a change in the other. Such a relation can be very helpful to a software maintainer.

To learn this relation we will mine the change repository of our subject legacy system. Sections 5 to 7 provide the details of this process as well as some of the results. The conclusion and future work is presented in section 8.

2. Relevance Among Software Entities

In this section we provide the definitions of Relevance Relation and other concepts closely related to it. We will also discuss a specific example of such relations in the context of software maintenance.

2.1 Relevance Relations

Definition: A *Software Entity* is any semantically or syntactically valid construct in a software system that can be seen as a unit with a well defined purpose[2]..

Examples of software entities include documents, source files, routines, modules, variables etc.

Definition: A *Predictor* is a relation that maps tuples of one or more software entities to corresponding values reflecting a prediction made about the elements of the tuples.

Definition: A *Relevance Relation* (RR) is a predictor that maps tuples of two or more software entities to a value *r* quantifying how *Relevant* i.e. connected or related, the entities are to each other. In other words *r* shows the strength of *relevance* among the entities. Therefore, "Relevant" here is embedded in, and dependent on, the definition of the relation.

For instance, in section 2.2 we will discuss a relevance relation called co-update that maps file pair tuples to one of two relevance values Relevant or Not-Relevant.

The relevance value *r* can be a number between 0 (lack of any relevance) and 1 (100% relevant), in which case the

[1] Due to the space limitation the discussion of related work has been kept to a minimum.

[2] Unless otherwise stated, in this paper an entity means a software entity.

relevance relation is *continuous*, or it can be one of a set of predefined values, in which case the relevance relation is *discrete*.

2.2 A real world example of Relevance Relations

When a maintenance programmer is looking at a piece of code, such as a file or a routine, one of the important questions that she needs to answer is:

"*which other files should I know about, i.e. what other files might be relevant to this piece of code ?".*

Knowing the answer to this question is essential in understanding and maintaining any software system regardless of the type of maintenance activity.

In this paper we will therefore focus on a special kind of relevance relation where the entities are files. We want to learn a special class of Maintenance Relevance Relations (MRR) called the *co update* relation:

co-update(f_i, f_j) $\rightarrow$ {Relevant, Not-Relevant} where,

$i \neq j$ and f_i and f_j are any two files in the system.

co-update(f_i, f_j) $\rightarrow$ Relevant means that a change in f_i may result in a change in f_j, and vice versa .

As it can be seen in this definition, the co-update relation is a discrete relevance relation mapping two entities (files in this case) to one of the two relevance values.

A relevance relation such as co-update could be used to assist a maintenance programmer in answering the above question about files.

3. Relevance Relations and classification learning

While in most cases one can easily specify the behavior of a relevance relation in terms of its domain and range, the actual definition of the relation is unknown. It is also the case that one could provide instances of the relation of interest, without knowing an exact definition for it. For instance by looking at a software change repository we can find past instances of the co-update relation, without knowing the definition of the relation itself. If we knew the definition, we could use it to predict whether for any pair of files in the system a change in one may require a change in another, even if we do not have a record of them being changed together in the past. Therefore, it is natural for one to try to learn from these instances.

The problem of learning a relevance relation can be directly mapped to a classification learning problem. Here the classifier represents the *learned* relevance relation. In classification learning terminology we are trying to learn a *concept* e.g. the co-update relation between files. Figure 1 shows the relation between a relevance relation and a classifier modeling it.

To learn a concept such as the co-update relation, an *induction algorithm* must be provided with pre-labeled (pre-classified) *examples* or *cases* of that concept. The example consists of the description of the concept and an assigned *class* or *label*. An example is described by calculating the value of a list of predefined *attributes* or *features*. The features of an example must describe the entities in the corresponding tuple which is mapped by the relevance relation. For instance an attribute could be the file type of the first file in a co-update tuple with possible values "source" and "header". Another attribute could be the number of routines called by both files in the tuple. In the first case the attribute is based on one of the entities in the tuple, while in the latter case the attribute is based on two entities.

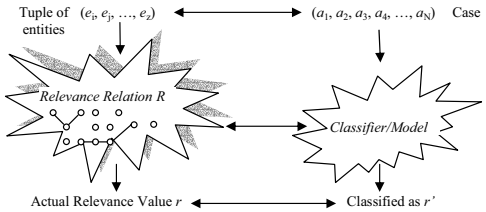


Figure 1. The Mapping between a Relevance Relation and a Classifier

The output of the induction algorithm is a classifier which will model the relevance relation of interest. As Figure 1 suggests, once a model is learned a previously unseen tuple of entities (e_i, e_i,…, , e_z) can be translated to a case with feature or attribute values (a_1, a_2,…, , a_N), and input to the learned model. The output of the model r' is an approximation of the actual relevance value r [3]. A classifier that always correctly classifies the given cases, accurately represents the corresponding relevance relation. However, this is hardly ever the case in a real world setting. In Section 4 we will discuss how we can measure the performance of generated classifiers or models.

Due to software engineers' time restrictions it is not always practical to ask them to provide instances of the co-update relation, therefore we used heuristics instead.

We extract from the software change repository all the updates applied to the system within the time period for which the data will be mined. The *Co-update heuristic* suggests that files changed together by each update be paired and label as Relevant.

The *Not-Relevant heuristic* labels a pair of files as Not-Relevant if these files are not changed by any updates within the time period used for mining

If T=[T_1,T_2] is the period of time to which the Co-update heuristic was applied, T'=[T_3,T_2] the period of time to which the Not Relevant heuristic is applied includes T i.e., $T_3 \leq T_1$

Our methodology allows the set of Not-Relevant tuples

[3] Here we emphasize that any relevance relation can be modeled by a classifier learned from the instances of that relation. For the special case of suggesting software entities that tend to change together one could also employ other techniques such as association rule learning as discussed in [5].

be further refined, if there is additional information, e.g. expert feedback, suggesting a tuple should not be labeled as Not-Relevant.

4. Measuring classifier performance

Figure 2 shows a *confusion matrix*, where the counts of predicted versus actual class of examples used for evaluating a model are tabulated. The following measures can be derived from this matrix.

		Classified As	
		Relevant	Not-Relevant
True Class	Relevant	*a*	*b*
	Not-Relevant	*c*	*d*

Figure 2. A confusion matrix

$$TP_R = \frac{\text{Relevant cases correctly classified}}{\text{Number of Relevant cases}} = \frac{a}{a+b}$$

$$FP_R = \frac{\text{Not - Relevant cases incorrectly classified}}{\text{Number of Not Relevant cases}} = \frac{c}{c+d}$$

$$E_c = \frac{C_{PN} * b + C_{NP} * c}{a + C_{PN} * b + C_{NP} * c + d}$$

In our research we consider the Relevant class to be the positive class and the Not-Relevant class the negative class. TP_R and FP_R are the *True Positive* and the *False Positive* rates for the Relevant class. Readers familiar with the *Recall* measure will recognize that it is the same as true positive rate. E_c is the *Cost Sensitive Error Rate*. It generalizes the formula for *Error Rate* by allowing arbitrary cost factors to be assigned to each of the two possible misclassifications. C_{NP} is the cost factor for misclassifying negative examples e.g. Not-Relevant as positive. C_{PN} is the cost factor for misclassifying positive examples e.g. Relevant, as negative. When the costs for both kinds of errors are set to one, this formula simplifies to the formula for error rate.

All three measures above are normalized. Ideally we would like to have a classifier with a true positive rate of 1, and the false positive and error rates of 0.

5. Attributes used in our experiments

To learn the co-update relation, we have experimented with different sets of attributes. These attribute sets can be divided into two groups:

- Syntactic attributes
- Text based attributes

Syntactic attributes are based on syntactic constructs in the source code such as function calls, variable definitions or type definitions. These attributes are extracted by static analysis of the source code[4]. They also include attributes based on names given to files. Interested readers can find the complete list of these attributes in [4].

Text-based attributes allow us to exploit another source of knowledge about the files modified together: the text of comments and problem reports. Each file is represented by a vector of features that correspond to the words found in the collection of all comments or all problem reports. Such a "bag of words" is a commonly used representation method for documents in information retrieval and machine learning. The Boolean bag of words representation sets a feature to *true* if the corresponding word exists in the document and *false* otherwise.

We have adapted this representation to accommodate file pair tuples as is the case for the co-update relation. After assigning a bag of word feature vector to each file, we create an example for tuple (f_i, f_j, *r*) by creating a new bag of words feature vector which is the intersection (or logical AND) of the feature vectors corresponding to f_i and f_j. Therefore, in the new file pair feature vector a feature is set to true if the corresponding word appeared in the sets of words assigned to both f_i and f_i, otherwise the feature is set to false. The idea here is to find similarities between the two files. Of course the example will be labeled as *r* e.g. Relevant or Not-Relevant.

We have created bag of word feature vectors for files using:

- Source code comments
- Problem reports

In the first case a program file is seen as a document, i.e. assigned a set of words, consisting of the words in its comments.

In the case of problem reports the words in problem reports must somehow be associated with program files. This is achieved by creating a set of words for each file consisting of the words in all problem reports that caused it to change.

6. Experimental setup

The subject for our experiments was a large telephone switching software system (a PBX) developed by Mitel Networks corporation. This software was originally created in 1983 and is still a major source of revenue for our industrial partner. Approximately 1.9 millions lines of high level language (HLL) and Assembler code were distributed in about 4700 source files. The high level language source files which are the subject of this report constitute about 75% of these files.

In our research we used the source code and error tracking and update data maintained in a system called SMS. Using SMS one can view problem reports submitted against the system and updates applied to fix them. By applying the Co-update heuristic we extracted a set of file pairs that were changed together by updates submitted in the 1995 to 1999 time period, i.e. the set of Relevant file

[4]Computing the value of some of these attributes involves steps similar to the ones taken to measure well known static software product metrics such as fan in, fan out, and cyclomatic complexity.

pairs. Using the Not-Relevant heuristic mentioned earlier, we found a set of file pairs that were not changed together during this time period, i.e. the set of Not-Relevant file pairs[5].

The *group size* of an update is the number of files changed by it. Experiments reported in this paper limit the Relevant file pairs to the ones changed by updates where the group size is at most 20. These updates constitute 93% of updates with a group size larger than 1. Our experiments have shown that limiting group size generate better results than not doing so. We generate a new Relevant tuple (and example) for each individual update that change a pair of files together. Table 1 shows the distribution of Relevant and Not Relevant examples used in our experiments. We split these examples to 1/3 *Training Repository* and 2/3 *Testing Repository*.

Table 1. Class distributions

	Relevant	*Not Relevant*	*#Relevant/#Not Relevant*
All	4547	1226827	**0.00371**
Training	3031	817884	**0.00371**
Testing	1516	408943	**0.00371**

As discussed above for each instance of the co-update relation we generate an example by calculating the value for a set of predefined attributes. We use the relevance value of the instance as example's label e.g., Relevant.

Table 1 shows that the number of Not-Relevant examples is about 270 times the number of Relevant examples. This imbalance creates difficulties for most learning algorithms as they are designed to select models with higher accuracy. In this skewed scenario a classifier that classifies every example as Not-Relevant will have a very high accuracy, yet it will be completely useless. To compensate for this, we train our classifiers on far less skewed data sets. These training sets include all the Relevant examples and a sampled subset of Not-Relevant examples in the training repository so that thy will have the following Not-Relevant/Relevant ratios:

1-10,15,20,25,30,35,40,45,50

In other words we learn from 18 *training sets* with the above skewness ratios and test on the complete testing repository that has the original skewness among classes. In the reminder of this paper any reference to a *"ratio N classifier"* means a classifier that is trained on a training set with a skewness ratio *N*.

The learning system used in our experiments is C5.0[6], an advanced version of the C4.5 decision tree learner [3]. Decision tree learning is one of most widely used and well researched areas of machine learning. A decision tree is an explainable model that can be studied and reasoned about by domain experts.

7. Comparing models learned from syntactic and text based features

We have learned models of the co-update relevance relation by conducting three sets of experiments using the 17 syntactic feature set, the source file comment feature set, and the problem report feature set. Each set of experiments generated 18 classifiers corresponding to the above training skewness ratios. For each set of experiments we plotted TP_R of each classifier against its FP_R. This plot is known as the ROC plot. Figure 3 shows the ROC plot generated from these experiments[7]. In an ROC plot the ideal point is (1, 0) where the true positive rate is at its maximum and the false positive rate is at its minimum. A classifier that is on the north-west side of another classifier on the plot, i.e. closer to point (1, 0), is said to be the dominant one.. In the ROC plots shown in this paper the rightmost point corresponds to a classifier learned from a balanced training set, while the leftmost point corresponds to ratio 50 classifier.

As Figure 3 shows the text based attributes generate models that dominate the classifiers generated from syntactic features. The problem report based features generated the best models. Increasing the number of Not-Relevant examples in the training set causes a drop in TP_R and FP_R. The drop in TP_R, which is the undesirable effect, is far less in the case of classifiers learned from problem report based features. A closer look at the ratio 50 problem report based classifier revealed that it achieves a precision of 62% and a recall of 86% for the Relevant class. We believe performance values such as these makes a classifier a good candidate for field deployment.

We also combined text based features used in the classifiers of Figure 3 with syntactic features and repeated our experiments using these combined feature sets. As can be seen in Figures 4 and 5 the classifiers generated from the combined feature sets in most cases dominate the original text based classifiers. The effects are more prominent in the case of source file comment based classifiers, however this should not be very surprising as the problem report based classifiers already show fairly high quality.

Finally, Figure 6 shows the cost sensitive error rate plots for the ratio 50 text based classifiers and two random classifiers used as comparison baselines. In a two class classification problem, a common baseline is a random classifier with a probability of 50% for each class. However since the distribution of our classes is skewed we have also used a classifier with the same skewness as the testing repository. Examples in the testing repository

[5] To further restrict the size of this set the first file in a Not-Relevant pair must also appear as the first file in a Relevant file pair.

[6] We have also experimented with a simple learning algorithm called 1R [1] and Set Covering Machines (SCM) [2].The unsatisfactory, 1R results shows the complexity of the data. Results obtained for SCM were not significantly better than the ones obtained with C5.0.

[7] The axes in these plots are scaled between 0 and 100.

are randomly classified, with the desired probabilities, by these classifiers. To better account for the variation in the randomness, we create one accumulative confusion matrix for each classifier by repeating this 10 times. Figure 6 shows that increasing cost factors C_{NP}.and C_{PN} both increase the cost sensitive error rate, however text based classifiers perform better than both base random classifiers. The problem report based classifier generated the best cost sensitive error rates.

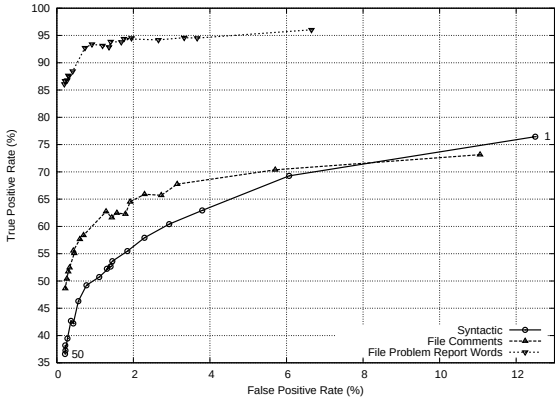


Figure 3. Comparing syntactic and text based features

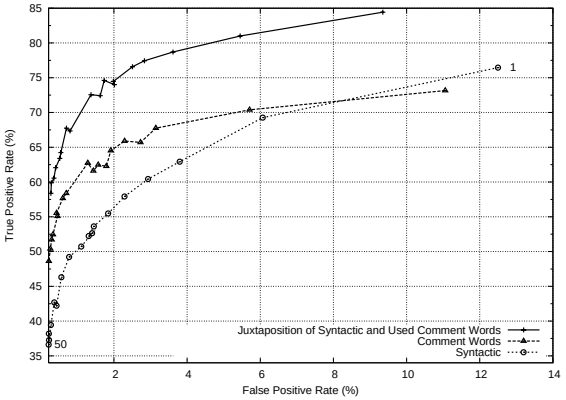


Figure 4. Combining syntactic and source file comment features

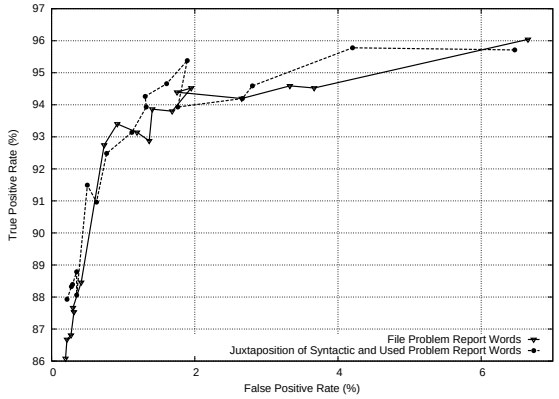


Figure 5. Combining syntactic and problem report features

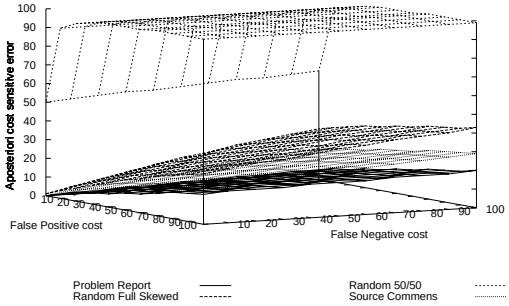


Figure 6. Cost sensitive error rate plots

8. Conclusion and future work

In this paper we presented the notion of Relevance Relation to represent relations among entities in a software system. We showed how classification learning can be used to model relevance relations. As a case study we set out to learn models for the co-update relevance relation between pairs of files in a large legacy system.

We presented results obtained from syntactic and text based feature sets, and their combinations. Our results show one can learn models with performance values that merit their practical use. We further analyzed these models under different misclassification cost assignments to evaluate their quality. The problem report based models generate some of the lowest cost sensitive error rates even in the presence of high misclassification costs. In the future we intend to experiment with other feature sets, and learn other relevance relations in software systems. We also plan to deploy the learned classifiers and evaluate their performance in the field.

References

[1] Holte R.C. 1993. Very Simple Classification Rules Perform Well on Most Commonly Used Datasets. Machine Learning, Vol. 3 pp. 63 91

[2] Marchand M. and Shawe-Taylor J. 2002, The Set Covering Machine, JMLR, Vol. 3, pp. 723 745

[3].Quinlan J.R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Pat Langley, Series Editor

[4] Sayyad Shirabad J., Lethbridge T.C. and Matwin, S. 2003. Mining the Maintenance History of a Legacy Software System. Proceedings of the 19th ICSM, pp. 95-104.

[5] Zimmermann T., Weißgerber P., Diehl S., and Zeller A. 2004. Mining Version Histories to Guide Software Changes. Proceedings of 26th ICSE. (To appear).

Four Interesting Ways in Which History Can Teach Us About Software

Michael Godfrey Xinyi Dong Cory Kapser Lijie Zou
Software Architecture Group (SWAG)
School of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA
email: {migod,xdong,cjkapser,lzou}@uwaterloo.ca

Abstract

In this position paper, we outline four kinds of studies that we have undertaken in trying to understand various aspects of a software system's evolutionary history. In each instance, the studies have involved detailed examination of real software systems based on "facts" extracted from various kinds of source artifact repositories, as well as the development of accompanying tools to aid in the extraction, abstraction, and comprehension processes. We briefly discuss the goals, results, and methodology of each approach.

1. Introduction

This position paper describes four broad approaches to studying software system evolution that yield different perspectives on how and why a system has evolved. These approaches are:

- coarsely-grained longitudinal case studies of growth and evolution,
- finely-grained case studies of *origin analysis* between consecutive versions of a system,
- case studies of code cloning within families of related systems, and
- tracking how build architectures and software manufacturing-related artifacts change over time.

Each of the above approaches involves three basic steps:

1. extraction of raw "facts" from various kinds of source artifact repositories,
2. automated, semi-automated, and manual analysis techniques performed on the facts, and
3. tool-supported exploration, navigation, and visualization to aid in comprehension.

We note that there is sufficient space in this position paper only to outline our results and methodologies. The reader is referred to listed references for more details.

2. Longitudinal case studies of growth

2.1. Goals and Results

Previously, we have studied growth and evolution of several open source software systems, including the Linux operating system [2]. Our original goal was to track how the growth patterns of large open source systems compared to previous studies on (non-open source) industrial systems; in particular, we wished to determine if Lehman's Laws of Evolution [8], which had been derived based on studies of (closed source) industrial software systems, also held for open source systems. We found that they did not hold in several instances; perhaps the most surprising result was that the Linux kernel continued to grow at a geometric rate even after surpassing two million lines of code (2 MLOC) (Fig.1). Lehman's empirical model predicts slowing growth as a software system becomes "large".[1]

2.2. Methodology

For these studies, we analyzed the source code of the Linux operating system in a semi-automated manner. This involved manually downloading and unpacking 96 versions of the kernel source code "tarball", running a set of hand-crafted `bash` and `awk` scripts over them to measure their

[1] Lehman has personally acknowledged that this study does indeed contradict some of his laws, and has said that he will need to reformulate them to take into account the various phenomena of open source software development.

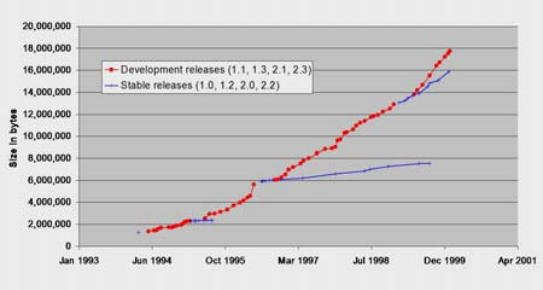


Figure 1. Growth of the compressed `tar` file distribution for the Linux kernel source release; measuring size as lines-of-code, number of modules, *etc.*showed roughly the same geometric growth pattern [2].

size in various ways and at various levels of abstraction, and then analyzing and exploring the results within a spreadsheet.

At the time the study was done, we did not have access to (or know of) a "live" CVS repository that could have aided in automating the analysis tasks. The size of the Linux kernel source itself (over 2 MLOC) and its relative fragility (it can be difficult to configure, build, and compile in an automated way) limited the amount and kinds of analysis we were able to perform. Its fragility meant that we could not reliably use our preferred static analysis tools to get a more sophisticated understanding of its complexity, and its size meant that we could not easily store more than a few unpacked kernels at a time on the file system. Effectively, we "boiled the ocean" of source code down to sets of numbers that could be stored inside a few spreadsheets, and used those as the basis for our analysis.

3. Case studies of *Origin Analysis*

3.1. Goals and Results

A particular problem for program comprehension is accurately modelling the *ontology* of a system's components. That is, the identity of a component is often equated with the name of the containing file or programming language entity (possibly together with its location within, say, a directory hierarchy). If a component is renamed or moved, it is considered that the old component has died and a new one has been born. However, as a system is redesigned and refactored over time, it is fairly common for components to be renamed, moved to another container, and merged into or split from other components. While the name/location-based identity assumption has the advantages of being simple and easy to implement in a tool, a lot of useful knowledge about the system can be lost if the system has undergone internal change and refactoring.

We have therefore sought to develop a set of techniques for performing what we call *origin analysis*, as well as a supporting tool called Beagle [11, 12]. That is, when confronted with a set of programming language entities that appear to be new to a particular version of a software system, we try to determine which of these components really *are* new and which are in fact derived from entities in the previous version of the system.

Origin analysis continues to be an ongoing area of research for us, but we have already performed two detailed case studies. The first explored the incidence of moving and renaming of functions within the GCC compiler suite [11]; Table 1 shows the results of applying origin analysis of the `parser` subsystem of version 1.0 of the EGCS variant of GCC. This subsystem does not exist in the previous evolutionary ancestor (GCC version 2.7.2.3), yet we were able to determine that approximately 46% of the 848 functions originated from various places within the ancestor. The second case study, which is not detailed here, examined the incidence of merging and splitting in the PostgreSQL relational database [12].

File	*# Func*	*# New*	*# Old*	*Overall*
`c-aux-info.c`	9	0	9	Mostly Old
`fold-const.c`	44	15	29	Mostly Old
`objc/objc-act.c`	167	17	150	Mostly Old
`c-lang.c`	16	14	2	Mostly New
`cp/decl2.c`	57	50	7	Mostly New
`cp/errfn.c`	9	9	0	Mostly New
`cp/except.c`	25	20	5	Mostly New
`cp/method.c`	30	26	4	Mostly New
`cp/pt.c`	59	57	2	Mostly New
`except.c`	55	52	3	Mostly New
`c-decl.c`	70	29	41	Half-Half
`cp/class.c`	61	31	30	Half-Half
`cp/decl.c`	134	84	50	Half-Half
`cp/error.c`	31	16	15	Half-Half
`cp/search.c`	81	40	41	Half-Half

Table 1. Summary of origin analysis results for the (apparently) all new parser subsystem of the EGCS 1.0 compiler relative to its immediate evolutionary ancestor (GCC 2.7.2.3).

3.2. Methodology

Origin analysis is comprised of two basic techniques: *entity analysis* and *relationship analysis*. Entity analysis, which is similar to software clone detection, attempts to match entities of the two system versions based on similarity of the entities themselves. We have implemented this as a metrics-based "fingerprint" [7]. A set of metric values (*e.g.*, cyclomatic complexity, fan-in/out) for functions are

precomputed on system check-in by the commercial tool Understand for C++, and the results are stored in a relational database. At querying time, the candidates with the closest Euclidean distance to the metric tuple of the target entity are returned.

Relationship analysis is based on the assumption that if an entity is moved or renamed, there is a high likelihood that it will still engage in many of the same relationships with the same entities as before (*e.g.*, calls, called-by, inherits, uses-var). Various relations are extracted and recorded at system check-in by the cppx fact extractor for C/C++ systems [9], and stored in a relational database. At querying time, the candidates that have the most similar relational images (*e.g.*, call the same functions) are returned.[2]

Effectively, origin analysis reduces the source code down to sets of numerical values and abstracted facts about the software entities and their inter-relationships; we store this information in a relational database, and use a graphical tool to perform directed queries to help establish the most likely "origin" of software entities that appear to be "new".

4. Case studies of code cloning

4.1. Goals and Results

While algorithms and tools for code duplication detection (*i.e.*, "clone detection") have been well studied by the research community, there has been relatively little investigation into what types of clones might exist and how often and in what context they occur within industrial software systems. We feel that these questions are important as they can help us to develop criteria to evaluate the effectiveness of current detection techniques, and provide insight into how these tools might be improved.

Recently, we performed a case study on the incidence of code cloning within the file system component of the Linux operating system [6]. Our initial goal was to investigate how code duplication occurred in a well known industrial software system, and we began to classify the types of clones we found.

Our study led to several broad observations about cloning within the candidate system, and to hypotheses about cloning in general. For example, we found that files that belonged to the same subsystem (in this case, a particular file system implementation, such as `ext3`) often had many instances of cloning within them. Overall, we found that 78% of clone-pairs occurred within the same subsystem; this led us to hypothesize that some degree of system structure might be determinable based on the clone relationships. We also found that subsystems that share higher-than-average amounts of code duplication between them were often in a relationship of one being derived from the other, or the creation of one was heavily based on the other. In Figure 2, we have labelled three points in the graph where the subsystems were in such a relationship.

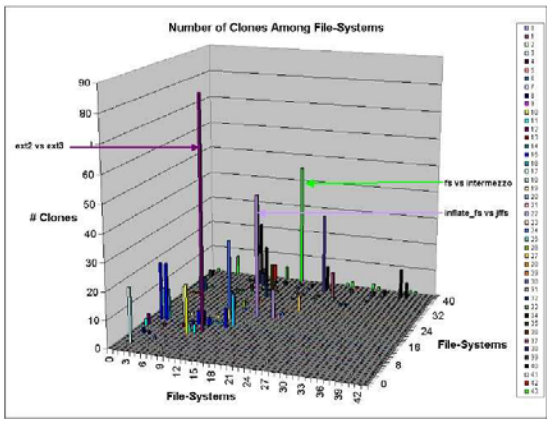


Figure 2. Number of clone pairs between file systems (excluding themselves).

We also observed that clones that existed between files in the same subsystem were usually function clones (according to the definition in the next section), whereas clones between files in different subsystems were usually not function clones. This is another indication that information about system structure may reside within the cloning relationship. We hypothesize that cloning activity may provide strong clues about file relationships.

Finally, we found that clones between files of different subsystems were often the "remains" of functions that had been cloned, but had been changed substantially enough that they could no longer be easily found as function clones. This insight has led us to consider a new vein of investigation; we plan to use change data from CVS repositories to profile how function clones change over time. Our analysis will address discovering which developers tend to introduce cloned code in a software system, who makes changes that drive them to become different, and if bug fixes are consistently made across all clone instances.

4.2. Methodology

To detect clones, we used both parameterized string matching [5, 4] and metrics-based string matching [1].[3] For brevity we will not describe these techniques here; the reader is referred to the cited references. After the initial extraction, we found that the tools returned a total of 5000

[2] Relationship analysis is also key to detecting merging and splitting, but the details are more subtle [12].

[3] We used the CCFinder tool to perform parameterized string matching [5]; we implemented our own metrics-based tool, following the design of others [1, 7].

clone pairs; of these, we determined through inspection that 1996 of these pairs were clearly false-positives and so were removed.

In the study, we identified five types of clones: *function clones*, *initialization clones*, *finalization clones*, *cloned blocks*, and *cloned blocks within the same function*. For the first four clone types, we further subdivided each group into clones in the *same file*, the *same subsystem*, and *different subsystems*. *Function clones* were functions where a minimum of 60% of the code of each function was duplicated between the two; a function clone can be composed of several smaller clone pairs. *Initialization clones* are pieces of source code at the beginning of a function that allocate space for and initialize variables; these clones must start in the first five lines of the function and end within the first half of the function. Analogously, *finalization clones* are pieces of source code which deallocate space and massage data for returning; these clones must start in the last half of the function and end in the last five lines of the function. *Cloned blocks of code* are segments of code that do not fall into any of the other types of clone

After categorization, and for any other empirical results we have presented, we performed manual inspection of a large percentage of the clone pairs in the given study to ensure that they were within the criteria that we specified and that they were accurately found as clones.

5. Longitudinal case studies of software manufacturing-related artifacts

5.1. Goals and Results

Software manufacturing — that is, the creation of software deliverables from source artifacts — is an important part of industrial software development. Large software systems often have complex subparts that engage in subtle relationships with the underlying technologies from which they are built; consequently, many such systems have complex and interesting architectural properties that can only be understood in the context of the various phases of system construction [10]. We consider that modelling and extracting build-time architectural properties of such systems are key to the software comprehension process, and so we have begun studying characteristics of development and maintenance activities that are related to software manufacturing (SM).

Recently, we have begun a project to study the maintenance effort of six open source projects[4] from a software manufacturing perspective (Table 2). We have attempted to measure the maintenance effort of SM-related artifacts for each project along three dimensions: the authorship, the size of the changes, and the frequency of the changes.

Project	*Period studied from*	*# files*	*# CVS records*	*# authors*	*Build tools used*
midworld	05/2002	199	1,677	8	SCons
mycore	07/2001	343	2,116	12	Ant
Apache Ant	01/2000	1,791	28,888	32	Ant
kepler	08/2003	412	1,129	6	Ant + Make
PostgreSQL	07/1996	2,093	59,815	22	Make
GCC	08/1997	17,378	150,423	204	Make

Table 2. Case study project information.

The results of the study suggest that the development and maintenance of SM-related artifacts is a significant activity during software evolution. For example, we found overall that more than half of the project developers contributed changes to the SM-related artifacts (Table 3). This may indicate that changes to the source artifacts often require changes to the SM-related files. In all but one of the systems we studied, SM-related files changed much more than often than the other kinds of source files. Overall, changes to the SM-related files accounted for non-trivial percentages of total system changes, from 3–10% depending on the project.

Project	*# authors*	*# authors who changed SM artifacts*	*Percentage*
midworld	8	5	62.50%
mycore	12	7	58.33%
Apache Ant	32	26	81.25%
kepler	6	4	66.67%
PostgreSQL	22	15	68.18%
GCC	204	154	75.49%

Table 3. Author involvement in SM activities.

As more evidence of the significance of the evolution of SM-related artifacts, we found several versions of systems where insertions and deletions of lines in SM-related artifacts accounted for up to 30% of the total insertion and deletion of lines in the system (Table 4).

Project	*LOC of SM files / total LOC*	*Changed LOC in SM files / total changed LOC*	*Peak*
Apache Ant	0.70%	1.59%	11.92%
PostgreSQL	2.33%	3.70%	24.99%
GCC	4.64%	20.05%	30.24%

Table 4. LOC and changed LOC in SM files.

In an attempt to find causes of the changes to the build system, we calculated the correlation of number of changes to SM-related artifacts to the number of times the file count changed. For half of our case studies, we found correlation stronger than 0.7. We found that none of systems showed a strong correlation to the size of the changes in SM-related artifacts and change in file counts.

[4] We note that one of the systems studied —Apache Ant —also happens to be a system building tool.

5.2. Methodology

For our study, we chose six open source software systems that used CVS as their versioning system. This allowed us to analyze the CVS logs — particularly of those artifacts that are related to SM — and perform statistical analysis on the data. The first step was to separate the SM-related artifacts from other source artifacts. The SM-related artifacts includes configuration scripts and system description files, such as `Makefiles`, Ant `build.xml` files, and `SConscripts`. We considered each CVS "commit" record to be one change, and the number of lines in each CVS commit to be the size of the change. By examining the size and frequency of changes in SM-related artifacts and source artifacts, we are able to address questions such as:

1. How much effort is put into SM-related artifacts?

2. What is the relation between evolution of SM-related artifacts and evolution software system as a whole?

A paper detailing the results of this study is under development.

6. Summary and challenges

This position paper has outlined four approaches to studying software systems using historical data extracted from various kinds of source artifacts. In each case, we performed automated analysis of the artifacts, then used various intermediate tools (such as `grok` [3], `awk`, and `bash` scripts) to created abstracted views that can be explored, navigated, and visualized for program comprehension purposes.

We conclude by noting that performing evolutionary studies of software systems presents several challenges that must be addressed by researchers in the field:

- *Scale* — In our experience, existing analysis tools often function "at the bleeding edge" of what is practical with respect to internal (and external) storage and processing. A typical static analysis performed on a single version of a system often produces voluminous detail and is computationally intensive; performing the same analysis across multiple versions of a system requires that scale issues be addressed seriously.

- *Automation* — Again, in our experience many software analysis tools require significant user intervention to extract accurate facts. This problem is exacerbated over multiple versions, and as new problems arise and have to be dealt with.

- *Artifact linkage and analysis granularity* — The canonical version control repository system, CVS, typically stores only source code, which it treats as plain text. A sophisticated tool for exploring software system evolution requires easy access not only to the "facts" and statistics about the source code that result from the analysis tools, but also to the source code entities themselves; CVS simply does not understand the notion of "method" or "function" embedded within a file.

References

[1] G. Antoniol, U. Villano, E. Merlo, , and M. D. Penta. Analyzing cloning evolution in the Linux kernel. In *Information and Software Technology 44(13)*, 2002.

[2] M. W. Godfrey and Q. Tu. Evolution in open source software: A case study. In *Proc. of 2000 Intl. Conference on Software Maintenance (ICSM-00)*, San Jose, California, October 2000.

[3] R. C. Holt. An introduction to the Grok language. Available at http://plg.uwaterloo.ca/~holt/papers/grok-intro.html, 2002.

[4] T. Kamiya, S. Kusumoto, and K. Inoue. A token-based code clone detection tool: CCFinder and its empirical evaluation. Technical report, Osaka University, 2000.

[5] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. In *Transactions on Software Engineering 8(7)*, pages 654–670. IEEE Computer Society Press, 2002.

[6] C. J. Kapser and M. W. Godfrey. Toward a taxonomy for source code cloning: A case study. In *Presented at First Intl. Workshop on Evolution of Large-scale Industrial Software Applications (ELISA-03)*, Amsterdam, September 2003.

[7] K. Kontogiannis. Evaluation experiments on the detection of programming patterns using software metrics. In *Proc. of 1997 Working Conference on Reverse Engineering (WCRE-97)*, Amsterdam, Netherlands, October 1997.

[8] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. Metrics and laws of software evolution —The nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics-97)*, Albuquerque, NM, 1997.

[9] A. Malton and T. Dean. The CPPX homepage: A fact extractor for C++. Website. http://www.swag.uwaterloo.ca/~cppx.

[10] Q. Tu and M. W. Godfrey. The build-time software architectural view. In *Proc. of 2001 Intl. Conference on Software Maintenance (ICSM-01)*, Florence, Italy, October 2001.

[11] Q. Tu and M. W. Godfrey. An integrated approach for studying software architectural evolution. In *Proc. of 2002 Intl. Workshop on Program Comprehension (IWPC-02)*, Paris, France, June 2002.

[12] L. Zou and M. W. Godfrey. Detecting merging and splitting using origin analysis. In *Proc. of 2003 Working Conference on Reverse Engineering (WCRE-03)*, Victoria, BC, November 2003.

Predicting Source Code Changes by Mining Revision History

Annie T.T. Ying*+, Gail C. Murphy*, Raymond Ng*
Dep. of Computer Science, U. of British Columbia*
{aying,murphy,rng}@cs.ubc.ca

Mark C. Chu-Carroll+
IBM T.J. Watson Research Center+
mcc@watson.ibm.com

(Paper not included in proceedings per authors' request)

Mining Software Usage Data

Mohammad El-Ramly
Department of Computer Science,
University of Leicester, UK
mer14@le.ac.uk

Eleni Stroulia
Department of Computing Science,
University of Alberta, Canada.
stroulia@cs.ualberta.ca

Abstract

Many software systems collect or can be instrumented to collect data about how users use them, i.e., system-user interaction data. Such data can be of great value for program understanding and reengineering purposes. In this paper we demonstrate that sequential data mining methods can be applied to discover interesting patterns of user activities from system-user interaction traces. In particular, we developed a process for discovering a special type of sequential patterns, called interaction patterns. These are sequences of events with noise, in the form of spurious events that may occur anywhere in a pattern instance. In our case studies, we applied interaction pattern mining to systems with considerably different forms of interaction: Web-based systems and legacy systems. We used the discovered patterns for user interface reengineering, and personalization. The method is promising and generalizable to other systems with different forms of interaction.

1. Introduction

An increasing amount of work is published on mining software "development" data, i.e., data produced while developing the software, e.g., CVS archives [16,17]. Also, a lot of work was done by the dynamic analysis community [9,10] on analyzing "runtime" data of software systems, e.g., execution traces of object oriented systems, etc. Little was done to mine software "usage" data, i.e., system-user interaction data collected while the system is running. Usage data is rich of information on how the system is currently used. This position paper makes a case for mining usage data in support of reengineering, reverse engineering and program understanding efforts.

Usage (or system-user interaction) data consists of temporal sequences of events that took place while the users were interacting with the system. We call such sequences "interaction traces". Typically, interaction traces contain interesting patterns of user activities and of the usage of system services in support of these activities. We demonstrate that sequential data mining techniques can be applied to discover these patterns. We used these patterns for interaction reengineering, i.e., reengineering the way the users interact with the system via user interface (UI) reengineering and personalization. Also, such patterns can be used for program comprehension, requirements recovery and reverse engineering tasks.

To validate our argument, we developed a process for mining interaction traces and successfully used it to mine usage data of two types of software systems, with radically different UI styles. The process retrieves a special type of sequential patterns, called interaction patterns. An interaction pattern is a frequently occurring sequence of events that might include up to a certain level of noise in each pattern instance, in the form of spurious events. Noise events can occur anywhere in a pattern instance. A pattern is considered interesting according to a user-defined criterion that defines the minimum pattern length and frequency and the maximum level of noise permitted.

We applied our "interaction pattern mining" process to legacy and Web-based systems. In the first application, we recovered the patterns of the frequent tasks done by the users of a legacy system from traces of its users' dialog with the legacy UI, recorded while the users were doing their regular jobs. These patterns model the currently active and demanded services of the legacy system as accessed via its UI. These service models are used for reengineering the legacy UI and wrapping it with a Web or WAP (Wireless Application Protocol) UI. In the second application, interaction pattern mining was used to discover the interesting navigation patterns in the Web server logs of a focused Web site. A focused Web site supports an ongoing evolving process and its users use it in a consistent way, e.g., navigation activities of different users during a period of time are more or less similar. Examples of such sites are Web sites of university courses. They evolve according to the course schedule and students access them similarly, following to the course-work progress. The discovered patterns can be used in reengineering the Web site UI for faster and easier access. This is done by giving online URL recommendations for current users to make their navigation easier and faster, by suggesting to them the consequent pages accessed by enough recent users who had similar navigation history.

While more case studies are needed, these applications demonstrate the applicability and value of the method and suggest that generalization to other forms of sequential system-user interaction and runtime data is straightforward.

In the following, Section 2 presents the "interaction pattern mining" problem. Sections 3 and 4 describe two applications of it. Section 5 is the summary and conclusion.

2. Interaction Pattern Mining

Mining sequences of data for recurring patterns is a generic problem with instances in a range of domains that are similar in that the data to be mined is represented by sequences, but different in the type of patterns of interest in each domain. Examples include sequential pattern mining of retail industry data [1], discovery of frequent episodes in event sequences [14], and discovering patterns in DNA and protein sequences [4]. Many algorithms to solve these problems emerged from data mining [1,2,3,14] and bio-informatics [11,13] communities.

Interaction pattern mining is similar to other sequential pattern mining problems in that the input data is ordered sequences of event Ids (or URLs or protein labels, etc.), but it is different in terms of the type of patterns sought. This is because interaction pattern mining constrains the maximum level of noise permitted in a pattern instance, but does not care where the noise occurs. This gives flexibility in discovering tasks that include user mistakes, trivial events and/or alternative paths for some subtasks.

To formally define interaction pattern mining,

1. Let A be the ***alphabet*** of events.
2. Let $S = \{s_1, s_2, \ldots, s_n\}$ be a set of sequences. Each ***sequence*** s_i is an ordered set of Ids drawn from A and represents a recorded trace of the runtime behavior of the system under analysis, e.g., an interaction trace.
3. An ***episode*** e, is an ordered set of events occurring together in a given sequence.
4. A ***pattern*** p is an ordered set of events that exists in every episode $e \in E$, where E is a set of episodes of interest according to some user-defined criterion c. E and e are said to "support" p. The individual Ids in an episode e or a pattern p are referred to using square brackets, e.g., $e[1]$ is the first Id of e. Also, $|e|$ and $|p|$ are the number of items in e and p respectively.
5. If a set of episodes E supports a pattern p, then the first and last Ids in p must be the first and last Ids of any episode $e \in E$, respectively, and all Ids in p should exist in the same order in e, but e may contain extra Ids, i.e., $|p| \leq |e| \; \forall \; e \in E$. Formally,
   - $p[1] = e[1] \qquad \forall \; e \in E$,
   - $p[|p|] = e[|e|] \qquad \forall \; e \in E$, and
   - $\forall$ pair of positive integers (i, j), where $i \leq |p|$, $j \leq |p|$ and $i < j$, $\exists \; e[k] = p[i]$ and $e[l] = p[j]$ such that $k < l$.

The above predicate defines the class of patterns that we are interested in, namely, ***approximate interaction patterns*** with at most a predefined number of insertion errors. For example, the episodes {2,***4***,3,4}, {2,***4***,3,***2***,4} and {2,3,4} support the pattern {2,3,4} with at most 2 insertions per episode, which are shown in bold italic font.

6. An ***exact interaction pattern*** q is a pattern supported by a set of episodes E such that none of its instances has insertion errors
   - $q[i] = e[i] \qquad \forall \; e \in E$ and $1 \leq i \leq |q|$
7. The ***support*** of a pattern p, written as *support* (p), is the number of episodes in S that support p.
8. **A *qualification criterion*** c, or simply ***criterion***, is a user defined quadruplet (*minLen*, *minSupp, maxError*, *minScore*). Given a pattern p, the ***minimum length*** *minLen* is a threshold for $|p|$. The ***minimum support*** *minSupp* is a threshold for *support* (p). The ***maximum error*** *maxError* is the maximum number of insertion errors allowed in any episode $e \in E$. This implies that $|e| \leq |p| + maxError \; \forall \; e \in E$. The ***minimum score*** *minScore* is a threshold for the scoring function used to rank the discovered patterns. A scoring function can be defined depending on the application in hand and the nature of the patterns sought.
9. A ***maximal pattern*** is a pattern that is not a sub-pattern of any other pattern with the same support.
10. A ***qualified pattern*** is a pattern that meets the user-defined criterion, c.

Given the above definitions, the problem of interaction pattern mining can be formulated as follows:

Given: (a) an alphabet *A*,
(b) a set of sequences *S*, and
(c) a user criterion *c*
Find all the qualified maximal patterns in *S*.

Solving interaction pattern mining problems is a three steps process. The first is a pre-processing step, needed for cleansing the data in the input sequences, depending on the problem in hand. In the second step, our novel algorithm for interaction pattern mining, Interaction Pattern Miner 2 (IPM2) [8], is applied to the cleansed sequences to discover the patterns that meet a user-defined criterion. The last step is the post processing analysis and comprehension of the discovered patterns. The first and third steps are application and data dependent.

3. Reengineering Legacy User Interfaces Using Interaction Patterns

CelLEST project for semi-automated legacy system UI reengineering [6,12,15] used interaction pattern mining as part of a lightweight automated process for UI reengineering of legacy systems that was designed to automate the then non-automated technology of our

industrial partner Celcorp. [5] CelLEST adopts a semi-automated task-centered lightweight process for wrapping legacy UIs with Web or WAP UIs. The idea is to semi-automatically understand and model the frequently used legacy system services, represented by frequent patterns of user activities. Then, new UIs for the desired platform are generated automatically, that wrap these demanded services. A significant innovation in the project is that the new UI packages an entire legacy system service (or user task) in the suitable UI on the target platform instead of mimicking the legacy interface one by one. For example, a system service that requires accessing 15 legacy screens may be reengineered into one Web form instead of 15 Web pages and forms because this is the natural representation of this task on the Web platform. Hence, the method is described as "task-centered". Service models, which were built from the patterns, are used by the new UI to invoke the legacy system services via the legacy UI as if the new UI is a typical user of the legacy system performing his or her tasks.

In this application, interaction pattern mining was used to recover the frequent patterns of user activities while using the legacy system. These patterns are buried in the huge amount of data exchanged in dialog between the users and the system via its UI. Mining the system-user interaction traces with the legacy UI discovers these patterns. The instances of each pattern are analyzed to infer information about the type and location on the screen of the user inputs. An analyst augments these patterns with extra semantic information to build full-fledged models of the frequent user tasks. Then, these models are automatically translated to abstract UI specifications and then to a UI implementation on the platform of choice: XHTML-enabled (Extensible Hypertext Markup Language) platforms or WAP devices. The automated pattern discovery and analysis process replaced the earlier time-consuming error-prone manual modeling process.

In CelLEST project, we used a host-access middleware that serves as an emulator and recorder to access legacy IBM 3270 systems. Unobtrusively through the middleware, legacy system users can open a session with the legacy application and do their regular jobs. The middleware records their dialog with the system UI in the form of sequences of legacy screen snapshots forwarded to the user' terminal, interleaved with the user actions in response, in the form of keystrokes. Screens and snapshots are like classes and objects; the later are instances of the former. Analysis methods, including feature extraction, clustering and others are used to derive a model of each legacy screen from all its instances in the traces. This model includes, for example, any distinguishing features of the screen like a title or a certain visual distribution of the screen content that occurs on all its recorded instances. In this step, each screen is given a unique Id. So, interaction traces can be abstracted by sequences of Ids as Figure 1 shows. Figure 1 is part of a real interaction trace taken from navigating a legacy library catalog system. Numbers are screen Ids and arrows are user actions.

To explain what type of interaction patterns can be found in these traces, Figure 1 shows two very similar segments of the user dialog with the legacy library system (in the dashed polygons) that occurred apart from each other in the trace. These segments suggest that the user was doing two similar runs of the same task, or alternatively, s/he was using the same legacy system service twice, although the two runs differ in the number of snapshots of screens 6 and 9 accessed. In the actual recorded trace, screen 4 displays the results of issuing a *browse* command to browse the relevant part of the library catalog file. Then, the user decides which items s/he wanted to retrieve from the catalog by issuing a *retrieve* command and s/he receives screen 5. Then, s/he displays brief information about the items using *display* command that displays screen 6. Finally, s/he selects an item using the *display item* command to display its full or partial information (screen 7). After selecting an option from screen 7 (e.g., full details, summary, etc.), screens 8, 9 and 10 display the first, intermediate and last pages of the required details, respectively. The number of snapshots of screens 6 and 9 retrieved varies depending on the item checked. The navigation segment of Figure 1 shows that this task of item information retrieval was done twice.

We applied interaction pattern mining to recorded system-user dialog traces of a number of systems [6]. To further explain the expected outcome of this application, we brief one of the experiments. In this experiment, we collected 5 traces of user interaction with the IBM 3270 connection of a public library system. Each trace represented a session of interaction between a user and the library catalog, during which s/he retrieved information about the library items of interest. The traces had 1657 screen snapshots in total and 27 screens (recall the classes

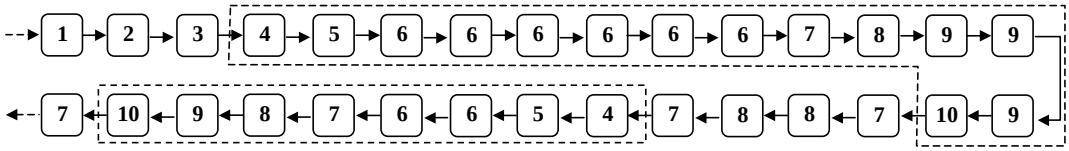


Figure 1. A Segment of a Trace of Interaction with a Legacy Library System

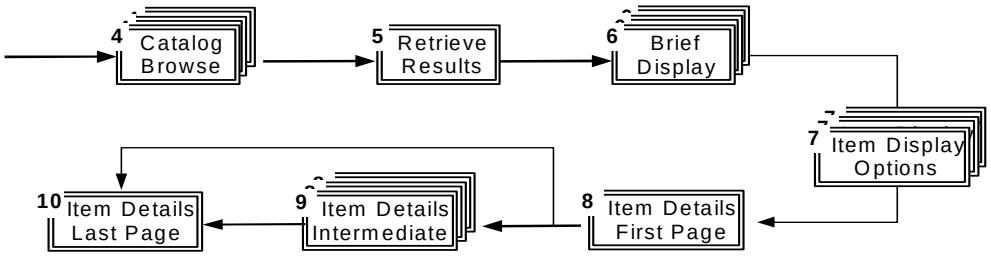


Figure 2. A Diagrammatic Representation of The Pattern 4^{+}-5-6^{+}-7^{+}-8^{+}-9^{*}-10, Corresponding to The Information Retrieval Task Repeated Twice in The Trace Segment of Figure 1.

and object analogy). The segment of Figure 1 is taken from one of the traces after identifying screens and labelling snapshots with their screen Ids. The traces were pre-processed and analyzed using IPM2 algorithm. Then manually, an analyst post-processed the discovered patterns by filtering out trivial patterns. Three patterns of interaction with the library catalog were discovered; each represents an information retrieval task, corresponding to a service of the legacy system. Figure 2 shows one of the interaction patterns discovered, which is {4^{+}-5-6^{+}-7^{+}-8^{+}-9^{*}-10}. n^{+} means one or more snapshots of screen n and n^{*} means zero or more. The two dashed polygons of Figure 1 represent instances of this pattern. Further details of this experiment are in [6].

In CelLEST, these patterns were used for lightweight UI reengineering. Each pattern was enriched with semantic information and then translated to abstract user interface specifications that are executable on multiple platforms, e.g., XHTML-enabled platforms and WAP devices [12]. Hence, a quick UI wrapper can be generated semi-automatically for the legacy services of interest.

Use Case name: Retrieving Information on a Library item
Participating actor: Library System User
Entry condition: The user issues a *browse* command
Flow of events:

1- Flip the catalog pages until the relevant page.
2- Issue a *retrieve* command to construct a results-set for the chosen catalog entry.
3- Display the results set using *display* command and turn its pages until the required item is found.
4- Issue a *display item* command.
5- Specify a display option.
6- Display the item details.

Exit condition: The user retrieves the required Information about the item s/he wants.

Figure 3. A Use Case Model Representation of The Interaction Pattern of Figure 2.

Beyond CelLEST, the enriched patterns can be represented in alternative formats to serve other reverse engineering and reengineering purposes. For example, to integrate the legacy system services with new object oriented applications designed in UML, it could be useful to translate the recovered interaction patterns to use cases to integrate with new UML-based requirements. Figure 3 shows a use case representation of the pattern of Figure 2. These patterns can also be used for re-documentation, and requirement recovery tasks.

4. User Interface Reengineering of Focused Web Sites Using Interaction Patterns

We used interaction pattern mining for Web-usage behavior analysis of focused Web sites and proposed a method for on-the-fly UI reengineering and personalization of such sites. A focused web site supports an ongoing evolving process and is usually navigated in a task-driven as opposed to data-driven way. The users of the Web site usually navigate it to accomplish the same task(s) during a certain period of time. As the process evolves, they shift together to other tasks. Models of these tasks, in the form of interaction patterns, are buried in the Web server logs.

We used a Web site of a computer science university course as an example of focused Web sites. The users of this site usually do similar tasks related to their course work every week, e.g., read and/or download new materials, lab instructions and assignments and related code, etc. Ideally, the discovered interaction patterns correspond to the frequent user tasks of interest, not just to interesting associations of Web pages. In other words, the sequence of navigation matters. Since interaction pattern mining tolerates noise, in the form of insertion errors, it can discover patterns of sequential user navigation with spurious navigation or slight differences. These patterns can serve as basis for online URL recommendations for current users to make their navigation easier and faster, by suggesting to them the further pages accessed by recent users who had similar

navigation sequences. Details of this application are in [7], but highlights are briefed below.

Mining Web logs for interaction patterns is a three steps process. First, preprocessing cleans and standardizes the Web server logs and divides them into sessions. A session is a coherent sequence of Web site navigation activities of the same user. Then, IPM2 is applied to the sequences of URL Ids representing sessions, with a user defined interestingness criterion. Then, post-processing depends on how the extracted patterns will be used.

We designed a method for Web UI reengineering and personalization using these patterns, which generates runtime recommendations for pages that new users of the Web site may want to visit. A runtime infrastructure, capable of user session tracking and relevant pattern selection, is needed in this application. The HTTP protocol is stateless and does not support establishing a long-term connection between the Web server and the client's browser. To address this problem, dynamic page rewriting with hidden fields can be used. When the user first submits a request, the server returns the requested page rewritten to include a hidden field with a session-specific Id. Each subsequent request of the user to the server supplies this Id to the server, enabling it to maintain the user's navigation history. This session-tracking method does not require any information on the client side and can therefore be employed, independent of any user-defined browser settings. Since the server knows the user's Id, it can examine its recent navigation history to identify whether it includes the prefix of any of the collected patterns. If so, the suffixes of the relevant patterns are offered as recommendations for subsequent navigation. Then, page-rewriting technique can easily support the dynamic adaptation of the pages requested by the users with the recommendations on new potential places to visit.

5. Summary and Conclusions

This position paper demonstrated that applying data mining to software usage data reveals valuable information about the system in the form of sequential patterns. Such patterns can be used for a variety of reengineering, program understanding and reverse engineering tasks. In particular, we presented a process for discovering a type of sequential patterns, called interaction patterns and two applications for it. The first is discovering patterns of the frequent user tasks in the recorded traces of system-user interaction with legacy systems. These patterns are used for automated UI reengineering. We also applied interaction pattern mining to discover frequent user navigation patterns from server logs of focused web sites. We proposed a method for lightweight Web site runtime reengineering by introducing on-the-fly URL recommendations based on these patterns.

Interaction pattern mining is a powerful technique for analyzing usage data and can be extended to different types of software runtime sequential data to discover patterns of user and/or system activities.

References

[1] R. Agrawal and R. Srikant, Mining Sequential Patterns. In Proc. of the 11th Int. Conf. on Data Engineering (ICDE), pg. 3-14, 1995.

[2] R. Agrawal and R. Srikant, Mining Sequential Patterns: Generalizations and Performance Improvements. In Proc. of the 5th Int. Conf. on Extending Database Technology (EDBT), 1996.

[3] J. Baixeries, G. Casas and J. Balcazar, Frequent Sets, Sequences, and Taxonomies: New, Efficient Algorithmic Proposals. Report No. LSI-00-78-R, El departament de Llenguatges i Sistemes Informàtics, Universitat Politècnica de Catalunya, Spain, 2000.

[4] B. Brejova, C. DiMarco, T. Vinar, S. R. Hidalgo, G. Holguin, and C. Patten, Finding Patterns in Biological Sequences. Unpublished project report for CS798G, University of Waterloo, Fall 2000.

[5] Celcorp, http://www.celcorp.com/

[6] M. El-Ramly, Reverse Engineering Legacy User Interfaces Using Interaction Traces. Ph.D. Thesis, University of Alberta, Canada, 2003.

[7] M. El-Ramly and E. Stroulia, Analysis of Web-Usage Behavior for Focused Web Sites: A Case Study. J. of Software Maintenance and Evolution: Research and Practice, vol.16, no. 1-2, pg. 129-150, 2004.

[8] M. El-Ramly, E. Stroulia and P. Sorenson, Interaction-Pattern Mining: Extracting Usage Scenarios from Run-time Behavior Traces. In Proc. of the 8th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD 2002), 2002.

[9] ICSE Workshop on Dynamic Analysis, 2003.

[10] ICSE 2nd Int. Workshop on Dynamic Analysis, 2004.

[11] I. Jonassen, Methods for Finding Motifs in Sets of Related Biosequences. Dr. Scient Thesis, Dept. of Informatics, Univ. of Bergen, 1996.

[12] R. Kapoor, Device-Retargetable User Interface Reengineering Using XML. Tech. Report TR01-11, Dept. of Computing Science, Univ. of Alberta, 2001.

[13] A. Floratos, Pattern Discovery in Biology: Theory and Applications. Ph.D. Thesis, Dept. of Computer Science, New York Univ., 1999.

[14] H. Mannila, H. Toivonen and A. Verkamo, Discovery of Frequent Episodes in Event Sequences. Data Mining and Knowledge Discovery, vol.1, no. 3, pg. 259-289, 1997.

[15] E. Stroulia, M. El-Ramly, P. Iglinski and P. Sorenson, User Interface Reverse Engineering in Support of Interface Migration to the Web. Automated Software Engineering, vol.3, no. 10, pg. 271-301, 2003.

[16] T. Ying, Predicting Source Code Changes by Mining Revision History. M.Sc. Thesis, Dept. of Computer Science, University of British Columbia, 2003.

[17] T. Zimmermann, P. Weißgerber, S. Diehl, and A. Zeller, Mining Version Histories to Guide Software Changes. In Proc. of Int. Conf. on Software Engineering (ICSE), 2004.

Defect Analysis

Bug Driven Bug Finders

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

We describe a method of creating tools to find bugs in software that is driven by the analysis of previously fixed bugs. We present a study of bug databases and software repositories that characterize commonly occurring types of bugs. Based on the types of bugs that were commonly reported and fixed in the code, we determine what types of bug finding tools should be developed. We have implemented one static checker, a return value usage checker. Novel features of this checker include the use of information from the software repository to try to improve its false positive rate by identifying patterns that have resulted in previous bug fixes.

1. Introduction

Static analysis of source code to locate bugs is a well-researched area [3][9]. Static analysis has several well-known benefits. Examining the source code without actually executing the code makes the quality of test suites, a hard problem, a moot point. Static analysis also allows code to be tested that is difficult to run in all environments, such as device drivers. There are a number of systems that provide a means to write code snippets that will be used to statically check code for one type of bug or another [5][10].

It is easy for programmers to think about types of bugs that might occur, and then devise a tool to look for these bugs. However, the space of possible tools to build is very large. Instead of creating solutions and looking for bugs, we propose that efforts to build bug-finding tools should start from an analysis of the occurrence of bugs in real software, and then proceed to build tools to locate these bugs. This paper describes a study of bug databases and software repositories to determine what types of bugs static checkers should be looking for by classifying the types of bugs that are frequently reported and fixed in the code.

2. Related Work

While previous work has tried to make general predictions about faults and identify trends across the software project from software repositories, our work is concerned with specific bugs. We determine the types of bug checkers that will be useful for a code base by looking at the history of its development. We also feed data mined from the revision history back into specific bug detectors to make decisions on which flagged errors are more likely to be true errors.

There are a large number of systems in use to statically check source code for bugs. These systems have been very successful in finding various types of bugs [2]. At the very basic end of these systems are compilers that perform type checking. A step beyond these are tools like Lint that have a set of patterns to match against the code to flag common types of programming errors [13]. Systems such as metal [6] allow the user to define what type of patterns the static analysis checker should look for via state machines that are applied to the source code. Simple data flow analysis has also shown to be an effective way to statically detect bugs [10].

While static checkers are effective at finding bugs, they can produce a large number of false positives in their results. Therefore, the ordering in which the results of a static bug checker are presented may have a significant impact on its usefulness. Checkers that have their false positives scattered evenly throughout their results can frustrate users by making true errors hard to find. Previous work on better ordering of results has focused on analyzing the code that contains the flagged error [11]. Unlike previous work, we will look at data collected over the entire project and historical trends to rank our error reports.

The historical data we use are mined from revision histories stored in software repositories. Data from software repositories has been used in a number of ways to guide the software development process. The software repository data has been used to identify high-risk areas of the code based on change histories [4]. It has been claimed that data based on change history is more useful in predicting fault rates than metrics based on the code, such as length [8]. Others have worked to identify relationships between software modules by studying which pieces of the source code are modified together [7][14].

3. Mining Historical Data

The software development process produces a number of artifacts as code is written and maintained. Chief among them are bug databases and source code

Table 1: Bugs Identified in the Bug Database

NULL pointer check	3
Return Value check	4
Logic Errors/Feature Request	34
Uninitialized Variable Errors	1
Error Branch Confusion	2
External Bugs (OS or other software failed)	2
System specific pattern	1
No identified code change	153

repositories containing revision histories. We use both of these artifacts to guide our research.

3.1 The Bug Database

To start our investigation, we reviewed the bug database for the Apache web server, httpd[1]. We studied the current branch of the software, which is version 2.0. We looked at the first 200 bugs that were marked as FIXED and CLOSED. We were interested in identifying the types of bugs that were fixed and matching the bug reports back to specific source code changes to classify the fixed bugs.

Our search through the bug database produced a number of interesting results. The bug reports in the bug database rarely can be tied directly back to a source code change. We were only able to tie 24% of the bug reports marked as fixed directly back to a code change. While a developer can post a comment detailing what code needed to be changed, or denote which CVS commit was created to resolve the bug, this is rarely done. Most bug reports consist of a discussion between the reporter and a developer. If the bug is fixed, the developer often ends the bug report with a short comment that contains some vague notion of where in the code the problem existed. We have also seen cases where the developer will be very specific and explain exactly what needed to be fixed or attaches a *diff*, but these seem to be the exception. Table 1 contains a breakdown of the bugs we were able to identify from the bug database.

The types of bugs we found in the bug database are worth discussing. Most of the bugs we found were logic errors or feature requests. Feature requests are just what is expected, a new feature for the software or porting a feature to a new platform. We categorize bugs where the code is correctly written to do the wrong logic as logic errors. These bugs can arise from the developer misunderstanding the specifications or not understanding how some web browsers act (in the specific case of Apache's httpd). These bugs do not lend themselves to being found via static checking. Bugs of this type are on the order of implementing the incorrect function to calculate a value, but doing the implemented calculation correctly.

Table 2: Bugs identified in the Software Repository

NULL pointer check	28
Return Value Check	29
Uninitialized Variable Errors	3
Failure to set value of pointer parameter	1
Feature Request	1
Error caused by if conditional short circuiting	1
Loop iterator increment error	3
System specific pattern	3

All but three of the bug reports we reviewed in the bug database came from users outside the project. These reports were mostly against a released version of the software, rather than a random CVS dump of the source. Only 2 of the bug reports were marked as being reported against a CVS-HEAD version of the source code. This leads us to believe that most of the simple "statically found" bugs are taken care of by the developers before a release is made. Hence, the users exercise few of these bugs.

3.2 The Software Repository

In order to understand what types of bugs are being committed to the software repository, but not making it into a release, we inspected commit messages in the CVS repository. We looked for commit messages that contained the strings 'fix', 'bug' or 'crash' and did not have a specific bug report listed. In this way we tried to weed out as many of the bugs from the bug database as possible. Moreover, we only looked at files that had a larger number of commits to them, 50 or more. Table 2 shows the breakdown of the bugs we were able to identify from the CVS repository.

The bugs found in the CVS repository were much more amenable to identification by static analysis. While a few continued to be the result of misunderstood specifications or some other logic error, a significant number were also of the kind easily found by static analysis: a problem with the code, not with the algorithm. The two most common types of bugs found in the CVS commits were NULL pointer checks and misuse of function return values. These two types of bugs accounted for 57 of the bugs we identified in the CVS commits.

4. Static Checker

Many of the bugs found in the CVS history are good candidates for being detected by static analysis, especially the NULL pointer check and the function return value check. We chose to develop a return value checker based on the knowledge that this type of bug has been fixed many times in the past. Additionally, a return value

checker can easily take advantage of data in the CVS repository to refine its results.

4.1 Return Value Checker

The return value checker we wrote checks to see if, when a function returns a value, that value is tested before being used. Using a return value can mean passing it as an argument to a function, using it as part of a calculation, dereferencing the value if it is a pointer or overwriting the value. The need for checking the return value is intuitive in C programs since the return value of a function often may be either valid data or a special error code. For example, in the case of returning a pointer the error code is often NULL. This error code could cause problems if the return value is dereferenced without being tested. If an integer value is returned, often -1 or 0 is an error code and these values should not be used in arithmetic. Even though the idea of a return value checker is not new [13], basing the return value checker on aggregate data and bug fix histories makes our approach novel.

Our checker categorizes each error it finds into one of several categories. Errors are flagged for return values that are completely ignored; the return value is never stored by the calling function. Errors are also flagged for return values that are used in some manner before being tested in a control flow statement. See Table 3 for the complete list of categories of errors our checker reports.

4.2 Ranking

The key to our checker is the ranking system used to present the output in a useful manner. Error reports are grouped by the called function. A function is ranked by how often its return value is tested before being used. This is an aggregate number generated by running the checker over all of the code in the current version of the software and tracking, for each function, the number of times the function is called and after how many of these calls the return value is used improperly. An *improper usage* of a return value is defined as either never storing the return value in the calling function or using the return value, as previously defined, before it is tested. We base our ranking on the notion that while developers produce bugs, they generally know how to use the return values of the functions they call and most often do so correctly. The more often a function has its return value checked, the more likely it is to *need* its return value checked. If a function almost always has its return value checked, the instances in which its return value is not checked are highly suspect and are good candidates for being bugs.

We also gather data automatically from the CVS commits to help with the ranking of the error reports. We search the CVS commit history to determine when a bug our checker would find has been fixed. The fact that the developer took the time to change this code suggests that it is an important change to make. We expect that the called function in such a bug fix, the function that previously did not have its return value checked, does need its return value checked before being used. Each such function we find is flagged as being involved in a bug fix in a CVS commit. We refer to these functions as being flagged with a CVS bug fix. We suspect when this function is called the return value has a valid reason to be checked before being used.

Our tool ranks errors involving functions flagged with a CVS bug fix higher than all functions not so flagged. Within each list of functions--with and without CVS bug fixes--the functions are sorted by the percent of their return values checked in the current snapshot of the software. At the top of the list then, are functions that very often have their return value checked and are flagged with a CVS bug fix.

We used a simple heuristic to determine if a CVS commit contains a return value check bug fix. The old and new versions of the committed files are both checked for return value check errors. For a given function in a file and a given function called by that function, if the new version has more return value checks that are not errors than the old version, the commit is said to fix a return value check bug for the called function. Note that simply adding an additional function call that has its return value checked makes it appear that a fix has been made.

5. Case Study

We ran a case study of our checker on the Apache httpd 2.0 source code. This is a large project with a deep CVS history. Our study was confined to the 2.0 branch and did not look into any code that resided solely in the 1.0 branch. The current snapshot contains about 200,000 lines of code and approximately 2,200 unique functions are called. These numbers include the core of the web server and optional modules. Our checker runs on Linux and we only considered modules that would run on such a system. We also included the Apache Portable Runtime (apr and apr-util) since the web server will not compile without it. The APR is a set of libraries produced by Apache to push some of the platform specific wrapper code out of httpd and give the developer a consistent set of APIs to use for common tasks.

In order to search the CVS repository for bug fixes we had to take a number of steps. For each CVS commit, we checked out the version of the code from the repository produced by that commit. We used the *configure* script supplied with the software to generate necessary files, including Makefiles. The Makefiles were used to determine the command line options needed to run the particular source file through our checker.

Table 3: Errors, CVS Bug fix flagged functions

	Checked 99% -51%	Checked 50% - 1%
Ignored (I)	22	33
Argument (A)	13	14
NULL dereference (N)	2	45
Calculation (C)	12	18
Stored, Unused (S)	8	27
Unused on Path (P)	15	9
Stored, Untested (U)	6	7

Table 4: Errors, non-CVS Bug fix flagged functions

	Checked 99% -51%	Checked 50%- 0%
Ignored (I)	67	2803
Argument (A)	48	1439
NULL dereference (N)	21	532
Calculation (C)	10	61
Stored, Unused (S)	32	429
Unused on Path (P)	17	486
Stored, Untested (U)	27	216

We successfully evaluated 5188 CVS commits to determine which functions were involved in a CVS fix to a return value check. There were 3811 more commits made to the CVS repository that we could not run through our checker. Some CVS commits would not configure correctly (1737). Some files contained C constructs that our parser could not handle, most notably having a variable number of arguments to a function (1027). The parser [12] we used was stricter with type checking than gcc. Many statements that would give warnings in gcc give errors in the parser. For instance, passing NULL, an integer, to a function that expects a void* caused the parser to raise an error. A number of commits also had true type errors where there was an actual bug checked in to the repository that resulted in a type error. The number of type errors, which caused a commit not to be checked, was 584. Also, source files raised an internal error in the parser 66 times. We were not able to track down the cause of these internal errors.

5.1 Initial Results

Our checker flagged 7,223 errors in the current snapshot of the httpd source. Each error flagged by the checker is an individual call site that has the return value produced by the called function used improperly. These 7,223 errors represent calls to 866 unique functions.

In searching the CVS commits, we found 75 functions that have a return value check bug fix and are called at least once in the current CVS snapshot. Of those 75, 41 have their return value checked 100% of the time in the current CVS snapshot (55%) and so are involved in no flagged errors. For comparison, 52% of all functions (886) had their return value checked 100% of the time. The remaining 34 functions are involved in 231 errors flagged by our checker. We consider these 231 errors likely candidates to be true errors. Note that this number of 231 does not include errors for functions with none of their return values checked, with large numbers (over 100) of unchecked return values or functions called via function pointers.

Upon inspecting these 231 errors, we believe 61 errors could be true bugs and need further inspection. The 61 bugs found in these errors gives a false positive rate of 74% for this chunk of our results (functions flagged with a CVS bug fix). See Table 3 for the breakdown of these results.

There were 86 functions not flagged with a CVS bug fix but with their return value checked more than 50% of the time in the current software snapshot. These functions account for 222 of the errors flagged by our checker. Since these functions have their return values checked more often than not, we expect these errors also to be likely candidates for being true errors. Upon inspecting these 222 errors, we believe 37 could be true bugs and need further inspection. This chunk of our results produces a false positive rate of 83%. See Table 4 for the breakdown of these results.

Overall we inspected 453 error reports and found 98 that we believe are suspicious and should be marked as a bug. This gives an overall false positive rate of 78%. The remaining 6,770 errors marked by our checker are produced by functions whose return value is checked 50% of the time or less and we expect these errors to be unlikely candidates to be true errors, thus we did not inspect them.

A false positive rate closer to 50% would be more palatable. A threshold for false positives is 50% since we would like a user to be as likely as not to find a bug when inspecting an error reported by our tool. Our technique has not yet achieved this false positive rate. However, a simple Lint-like tool would have had a higher false positive rate as each error report is given equal weight and not ranked in any way. We would have had to review each of the 7,223 errors to find the 98 bugs, which would be 73 false positives for every real bug.

5.2 A Bug Expressed

We were able to crash the httpd server by exploiting a bug found by our tool. The return value of the function ap_server_root_relative() is used directly as an argument to strcmp(). The function ap_server_root_relative() accepts two arguments, a fully qualified directory name and a filename. The return value is a char* that represents a path to a file, basically directory/filename. The return value can be NULL in a number of cases. The easiest way to get the function to return NULL is to have the fully qualified name of the file (plus NULL

terminator) to be larger than 4096 bytes. In this section of the source code, 4096 appears to be the size of all the filename buffers. Obviously, if one passes a directory and filename to the function that has a combined length of more than 4096 the function will return NULL. If this happens when the return value is used directly as an argument to strcmp() httpd will crash.

6. Conclusions

In this paper we have presented a method of creating bug-finding tools that is driven by the analysis of previous bugs. We have shown that the bugs cataloged in bug databases and those found by inspecting source code change histories differ in their types and level of abstraction. Bugs listed in a bug database are generally reported by users outside of the development team and are most often reported against a public release of the software rather than a CVS snapshot. These bugs are also of a more high level nature, involved with algorithmic problems rather than simple coding problems.

We have shown that the past bug history of a software project can be used as a guide in determining what types of bugs should be expected in the current snapshot. Moreover, such data can help to recommend which of a group of bug reports are more likely to be true.

The checker we have implemented checks for function return value usage errors and uses data mined from the revision history of the software to rank the results in a useful way. With our checker we have been able to identify 98 instances in the Apache web server that we believe should be classified as bugs and need further inspection.

In the future we want to identify other static bug checkers that can benefit from information mined from a CVS repository. We also plan to refine our current static checker and run it on other software projects.

7. Acknowledgements

This work was supported in part by DOE Grants DE-FG02-93ER25176, DE-FG02-01ER25510, and DE-CFC02-01ER254489 and NSF award EIA-0080206. We would like to thank Dan Quinlan at Lawrence Livermore National Laboratory for help in using the ROSE parser.

8. References

[1] Apache Web Server, httpd. Available online at http://httpd.apache.org

[2] Ashcraft, K., Engler, D., Using programmer-written compiler extensions to catch security holes. In *Proceedings IEEE Symposium on Security and Privacy*, Oakland, California, May 2002.

[3] Ball, T., Rajamani, S. K., The SLAM Project: Debugging System Software via Static Analysis, In *Proceedings of the 29th Symposium on Principles of Programming Languages* (POPL '02), Jan 2002, Portland, Oregon, USA, pages: 1 – 3.

[4] Bevan, J., Whitehead, E. J., Identification of Software Instabilities, In *Proceedings of 10th Working Conference on Reverse Engineering*, (WCRE '03) Victoria, British Columbia, Canada, Nov 13-17, 2003. pages 134-143.

[5] Engler, D., Incorporating application semantics and control into compilation, In *Proceedings USENIX Conference on Domain-Specific Languages* (DSL'97), October 15-17, 1997.

[6] 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.

[7] Gall, H., Jazayeri, M., Krajewski, J., CVS Release History Data for Detecting Logical Couplings, In *Proceedings of the International Workshop on Principles of Software Evolution* (IWPSE '03), Helsinki, Finland, September 2003, pages 13-23.

[8] Graves, T. L., Karr, A. F., Marron, J. S., Siy, H., Predicting fault incidence using software change history, *IEEE Transactions on Software Engineering*, Vol 26, Issue 7, July 2000. pages: 653 – 661

[9] Heine, D. L., Lam, M. S., A Practical Flow-Sensitive and Context-Sensitive C and C++ Memory Leak Detector In *Proceedings of the Conference on Programming Language Design and Implementation* (PLDI '03), June 2003.

[10] Hovemeyer, D., Pugh, W., Finding Bugs Is Easy, unpublished, http://www.cs.umd.edu/~pugh/java/bugs/docs/findbugsPaper.pdf

[11] Kremeneck, T., Engler, D., Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations, In *Proceedings of 10th Annual International Static Analysis Symposium*, (SAS '03) San Diego, CA, USA, June 2003. pages 295-315.

[12] Quinlan, D., ROSE: A Preprocessor Generation Tool for Leveraging the Semantics of Parallel Object-Oriented Frameworks to Drive Optimizations via Source Code Transformations. In *Proceedings Eighth International Workshop on Compilers for Parallel Computers* (CPC '00), Aussois, France, Jan 4-7, 2000.

[13] *Unix Time Sharing System Programmer's Manual*, AT&T Bell Laboratories, 1979. Seventh Edition, Volume 2a.

[14] Ying, A. T. T., Murphy, G. C., Ng, R. T., Chu-Carroll, M. C., Using version information for concern inference and code-assist. Position paper for *Tool Support for Aspect-Oriented Software Development Workshop at the Conference on Object Oriented Programming, Systems Language and Applications* (OOPSLA '02), Seattle, WA, USA, November 4-8, 2002.

Mining Repositories to Assist in Project Planning and Resource Allocation

Tim Menzies
Department of Computer Science,
Portland State University,
Portland,
Oregon
tim@menzies.us

Justin S. Di Stefano, Chris Cunanan,
Robert (Mike) Chapman,
Integrated Software Metrics Inc.,
Fairmont, West Virginia
justin@lostportal.net, ccunanan@ismwv.com,
Robert.M.Chapman@ivv.nasa.gov

Abstract

Software repositories plus defect logs are useful for learning defect detectors. Such defect detectors could be a useful resource allocation tool for software managers. One way to view our detectors is that they are a V&V tool for V&V; i.e. they can be used to assess if "too much" of the testing budget is going to "too little" of the system. Finding such detectors could become the business case that constructing a local repository is useful.

Three counter arguments to such a proposal are (1) no general conclusions have been reported in any such repository despite years of effort; (2) if such general conclusions existed then there would be no need to build a local repository; (3) no such general conclusions will ever exist, according to many researchers. This article is a reply to these three arguments.

To appear in the International Workshop on Mining Software Repositories (co-located with ICSE 2004) May 2004; http://msr.uwaterloo.ca.

1 Introduction

To make the most of finite resources, test engineers typically use their own expertise to separate critical from non-critical software components. The critical components are then allocated more of the testing budget than the rest of the system. A concern with this approach is that the wrong parts of the system might get the lions-share of the testing resource.

Defect detectors based on static code measures of components in repositories are a fast way of surveying the supposedly non-mission-critical sections. Such detectors can be a V&V tool for V&V; i.e. they can be used to assess if *too much* of the testing budget is going to *too little* of the system. As shown below, satisfactory detectors can be learnt from simple static code measures based on the Halstead [2] and Mccabes [3] features[1]. Such measures are rapid and simple to collect from source code. Further, the detectors learnt from these measures are easy to use.

Our experience with detect detectors has been very positive. Hence, we argue that organizations should routinely build and maintain repositories of code and defect logs. When we do so, we often hear certain objections to creating such repositories. This paper is our reply to three commonly-heard objections. For space reasons, the discussion here is brief. For full details, see [5, 6].

The first objection concerns a **lack of external validity**. Despite years of research in this area, there has yet to emerge standard static code defect detectors with any demonstrable external validity (i.e. applicable in more than just the domain used to develop them). Worse still, many publications argue that building detectors from static code measures is a very foolish endeavor [1, 7].

To counter the first argument, there has to be some demonstration from somewhere that at least once, another organization benefited from collecting such an endeavor. Paradoxically, making such a demonstration raises a second objection against local repository construction. If detectors are externally valid then organizations don't need *new data*. Rather, they can just *import data* from elsewhere. To refute this **buy not build** objection, it must be shown that detectors built from local data are *better than* detectors built from imported data.

Finally, if the proposal to build a repository survives objections one on two, then a third objection remains. Why is it that we make such an argument *now* when so many others have *previously* argued the opposite for so long? That is, it must be explained the **source of opposition** to static defect detectors.

The rest of this paper addresses these objections using the NASA case study described in the next section. Using that data, we show that external valid detectors can be generated. Next, we show that these detectors can be greatly improved using detectors tuned to a local project. Finally, we identity potential sources of systematic errors that may have resulted in prior negative reports about the merits of static code defect detectors.

2 Case Study Material

Our case study material comes from data freely available to other researchers via the web interface to NASA's Metrics Data Program (MDP) (see Figure 1). MDP contains around two dozen

[1] Elsewhere, we summarize those metrics [4]. Here we just say that Halstead measures reflect the density of the *vocabulary* of a function while Mccabe measures reflect the density of *pathways* between terms in the vocabulary.

Figure 1. The MDP repository: `http://mdp.ivv.nasa.gov`.

		module found in defect tracking log?	
		no	yes
signal detected?	no; i.e. $v(g) < 10$	A = 395 $LOC_A = 6816$	B = 67 $LOC_B = 3182$
	yes i.e. $v(g) \geq 10$	C = 19 $LOC_C = 1816$	D = 39 $LOC_D = 7443$

$$\begin{array}{rrr} Acc = & accuracy = & 83\% \\ PF = & Prob.falseAlarm = & 5\% \\ PD = & Prop.detected = & 37\% \\ prec = & Precision = & 67\% \\ E = & effort = & 48\% \end{array}$$

Figure 2. A ROC sheet assessing the detector $v(g) \geq 10$. Each cell {A,B,C,D} shows the number of modules, and the lines of code associated with those modules, that fall into each cell of this ROC sheet.

static code measures for thousands of modules based on the Halstead and McCabe measures. The data also include defect counts seen in up to eight years of project data.

From that data, various *data mining* [9] techniques have been applied to automatically build detectors. The output of these learners were compare to detectors generated by a DELPHI approach; i.e. asking experienced test engineers what thresholds they use to identify problematic code. These DELPHI predictors return "true" or "false" if some code measure passes some value.

The LSR and M5 data miners build predictors for the number of defects expected in new modules [9]. LSR uses linear standard regression to fit a single multi-dimensional linear model to the continuous detect data. For example, LSR generates equation such as Equation 1 below:

$$\begin{array}{rcl} defects_1 & = & 0.231 + (0.00344 * N) + (8.88e-4 * V) \\ & & -(0.185 * L) - (0.0343 * D) - (0.00541 * I) \\ & & +(1.68e-5 * E) + (0.711 * B) \\ & & -(4.7e-4 * T) \\ c_1 & = & -0.3616 \end{array} \quad (1)$$

Here, $\langle N, V, L, D, I, E, B, T \rangle$ are the *derived Halstead metrics* discussed in [4] and c_1 is the *correlation* of $defects_1$ to the actual error per module count. Correlation is discussed further below.

While LSR generates one equation, the M5 data miner can generate systems of equations. M5 is an extension of LSR that divides the data into a small number of regions and fits one linear model to each region.

Two other data miners used in this study were the J48 [9] ROCKY [4]. J48 is a standard decision tree learner and ROCKY is a home-brew learner than used a Gaussian approximation to propose interesting divisions of numeric data. ROCKY generates detector for each number attribute a of the form $a \geq N$ where $\geq N$ covers $\alpha\%$ of the Gaussian area. N is set such that:

$$\alpha \in \{0.05, 0.1, 0.15, \ldots, 0.95\} \quad (2)$$

ROCKY is a very simple learner that was run on subsets of the available data; i.e. just on the Mccabes data; just on the Halstead data; or just on simple lines of code (LOC) counts per module.

ROCKY and J48 process *discrete* defect classes. To generate discrete defect data, we took numeric defect counts per module and declared predicted defects "true" if $\#defects \geq 1$. In order to compare M5 and LSR to ROCKY and J48, the M5 and LSR output was converted to discrete booleans as follows. If M5 or LSR's predictions passes some threshold T, then predicted defects was set to "true". Our experiments repeated that test for:

$$T \in \{0.3, 0.6, \ldots, 3\} \quad (3)$$

The predictors generated by these methods were assessed via several assessment metrics. The *accuracy*, or Acc, of a detector as the number of true negatives and true positives seen over all events. In terms of the cells $\langle A, B, C, D \rangle$ shown in Figure 2, accuracy is $Acc = \frac{A+D}{A+B+C+D}$.

Apart from accuracy, several other measures are of interest. The *probability of detection*, or "PD", is the ratio of detected signals, true positives, to all signals; i.e. $PD = \frac{D}{B+D}$ (PD is also called the $recall$ of a detector). Also, the *probability of a false alarm*, or "PF", is the ratio of detections when no signal was present to all non-signals: i.e. $PF = \frac{C}{A+C}$. Further, the $precision$ of a detector comments on its correctness when it is triggered; i.e. $prec = \frac{D}{C+D}$.

Another statistic of interest is the $effort$ associated with a detector. If the detector is triggered, then some further assessment procedure must be called. For the particular static code defect detectors discussed in this paper, we will assume that this effort is proportional to the lines of code in the modules. Under that assumption, the $effort$ for a detector is what percentage of the lines of code in a system are selected by a detector; i.e. $effort = E = \frac{LOC_C + LOC_D}{LOC_A + LOC_B + LOC_C + LOC_D}$.

$Correlation$ is a statistic representing how closely two variables co-vary. Let a_i and p_i denote some actual and predicted values respectively. Let n and $\overline{x}$ denote the number of observations and the mean of the n observations, respectively. Then:

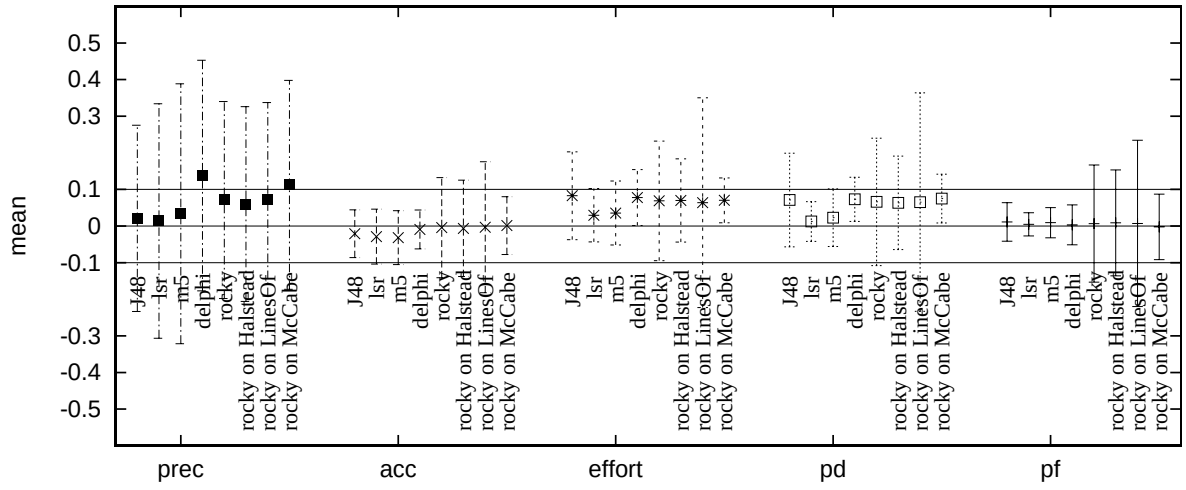


Figure 3. Between project stabilities in defect detectors. Mean μ and standard deviation σ of changes in defect detector statistics. Dots denote mean (μ) values. Whiskers extend from $\mu + \sigma$ to $\mu - \sigma$.

$$S_{PA} = \frac{\sum_i (p_i - \overline{p})(a_i - \overline{a})}{n-1};\ S_p = \frac{\sum_i (p_i - \overline{p})^2}{n-1};\ S_a = \frac{\sum_i (a_i - \overline{a})^2}{n-1}$$
$$correlation = c = \frac{S_{PA}}{\sqrt{S_p S_a}}$$

Correlation varies from -1 (perfect negative correlation) through 0 (no correlation) to +1 (perfect positive correlation). For example, the following equation, found via LSR using just lines of code LOC counts, has a very different correlation c to Equation 1 shown above:

$$\begin{aligned} defects_2 &= 0.0164 + 0.0114 * LOC \quad (4) \\ c_2 &= 0.65 \end{aligned}$$

3 Lack of External Validity?

To test the external validity of our detectors, we took five NASA applications, then learnt detectors from each of them using $\langle DELHI, LSR, M5, J48, ROCKY \rangle$. Because of Equation 2 and Equation 3, this resulted in hundreds of detectors. All these detectors were then applied to the other four applications.

As predicted by the **lack of external validity** objection, the detectors behaved differently when applied to the different applications. Figure 3 shows the mean and standard deviation of the *differences* in the values when *the same* detector was applied to *different* applications. For some learners and some assessment metrics, the observed standard deviations were quite large. For example, precision varied wildly and the variance in detectors built from module $linesofcode$ was always large.

However, in stark contrast to the **lack of external validity** objection, the differences were mostly very small. For example, with the exception of precision, most of the differences in the means were ≤ 0.1; and some learners consistently generated detectors with a very small variances (e.g. LSR,J48,DELPHI). To place Figure 3 in perspective $\langle pd, pf, acc, effort, prec$ all vary from zero to one so the *difference* between two (e.g.) pd values can vary from -1 to +1.

4 Buy, not Build?

The results Figure 3 come from a very varied set of applications. While all the studied applications used C or C++, they were built at four different locations around the country by five different teams for five very different application areas (ground station telemetry processing, flight software for earth orbiters, simulation tools for making predictions about hardware behavior, etc).

If defect detectors are so stable across domains, then the **buy not build** objection states that we need not build our own local repository. Instead, we need only reuse detectors learnt elsewhere.

Figure 4 is our reply to this objection. That figure shows results from learning detectors at *different* times in the life cycle of *the same* application. Defect logs were extracted at 6,12,18,24, and 48 months into the development of one of our applications. Defect detectors learnt at $time < X$ were applied to source code developed at $time \geq X$. Figure 4 shows the mean and standard deviations of the *differences* in $\langle pd, pdf, effort, acc, prec \rangle$ seen when *the same* detector was applied at *different times* to the *the same* application. Compared to Figure 3, the mean differences and standard deviations are greatly reduced.

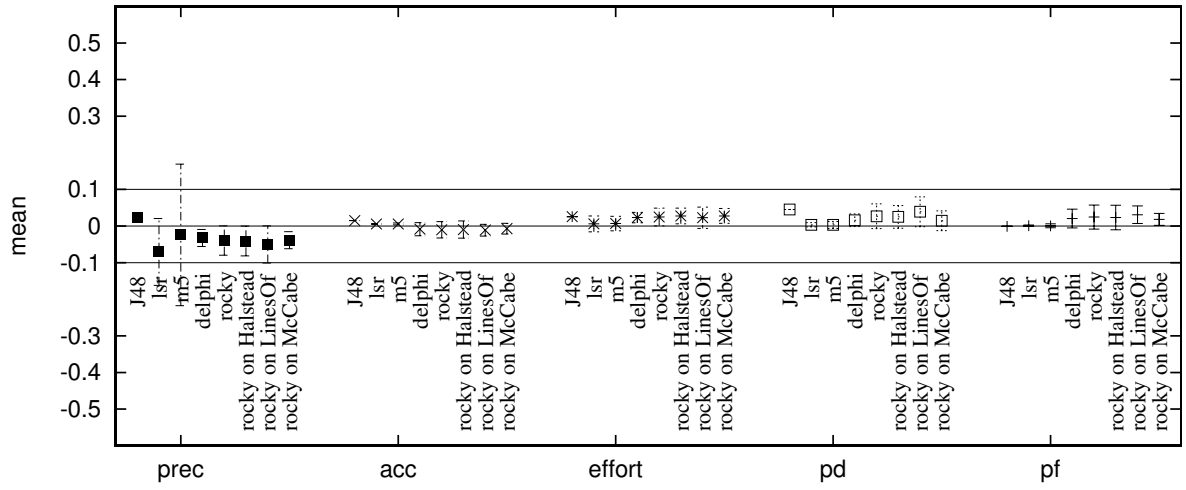


Figure 4. Within-project stabilities in defect detectors (same format as Figure 3.

5 Source of Opposition?

Figure 3 showed that defect detectors from other applications are *stable*; i.e. provide nearly the same results when applied to the current applications. Further, Figure 4 shows that detect detectors learnt from an historical record of the current application are *stabler*. This report is hence very positive on the merits of using repositories to build static code defect detectors.

Other researchers are not as positive. This section reviews some of those critiques and offers several source of systematic errors that may explain prior negative results in this area.

There are many reasons to doubt the merits of static code measures such as the Halstead/Mccabe metrics. Such metrics collected from a single module know neither (a)how often that module will be called nor (b)the severity of the problem resulting from the module failing nor (c)the connections *from* this module *to* other modules. Also, static code measures are hardly a complete characterization of the internals of a function. Fenton offers an insightful example where *the same* functionality is achieved using *different* programming language constructs resulting in *different* Mccabe measurements for that module [1]. Worse still, certain empirical evidence suggests that the Mccabe metrics might be no more informative than more simplistic measures. For example, Fenton & Pleeger note that cyclomatic complexity is highly correlated with lines of code [1]. Sheppard & Ince remarks that "for a large class of software it is no more than a proxy for, and in many cases outperformed by, lines of code" [7].

In reply to this pessimism, we take care to distinguish between *primary* and *secondary* detect detectors. We endorse standard practice in which test engineers *primarily* use their domain knowledge and the available documentation to identify the modules that require most of their attention. Our detectors are only *secondary* tools to quickly survey the parts of the system that were ruled out by the primary methods. If our secondary detectors trigger, then we would suggest that test engineers divert a little of the resources allocated by the primary method to check some other regions.

Primary detectors need a high probability of detection (pd). For all the reasons listed by Fenton & Pleeger and Sheppard & Ince, it is clear that defect detectors learnt from Halstead/Mccabe-style static code measures may not yield high pds. However, static code defect detectors are satisfactory secondary detectors. An important property of a secondary detector is a low probability of false alarm pf. Such low false alarms are required to ensure test engineers are not inappropriate distracted from their inspections of modules selected by the primary detectors. The bottom plot of

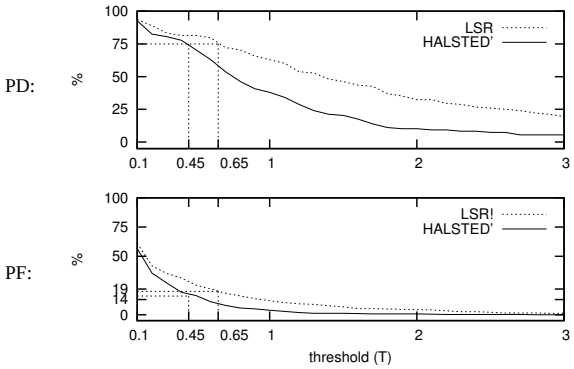


Figure 5. Effort, probability of false alarm, and probability of detection seen using $defects_i \geq T$ where $defects_i$ is one of Equation 1 (the "HALSTEAD" curves) or Equation 4 (the "LSR" curves) and T controls when the detector triggering (see the discussion around Equation 3).

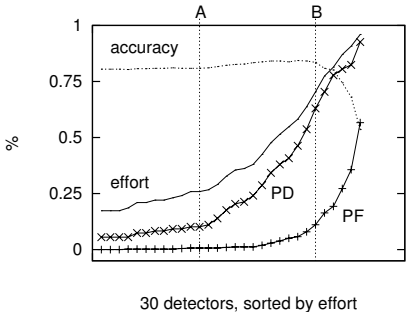


Figure 6. Each x-axis point x describes the $pf, pd, effort, accuracy$ of one detector.

Figure 5 shows the range of pf as a function of a detection threshold constant T. Note that by selecting T appropriately, detectors can be created with a pf that are so low that, if they are triggered by module X, then it becomes nearly certain that there is a defect in module X.

Empirically, detectors with low pfs also have low pds. For example, in Figure 5, a detector with a $pd = 75\%$ has a pf of around 20% and higher pfs have higher pds. This is to be expected since, as discussed above, static code defect detectors are ignorant of many features of an application. Hence, it is important that secondary detectors are paired with primary detectors since the latter has the greatest chance of finding bugs in the inspected regions. Similarly, primary detectors should be paired with low pf secondary detectors. While we *hope* test engineers are the most effective defect detectors, the available empirical evidence is, at best, anecdotal [8]. As shown here, much is known about the $\langle pf, pd, accuracy, effort, precision\rangle$ of static code measures. Further, they are cheap to build and easy to run over large code libraries. Hence, they are a useful way to check that aren't test engineers look in the wrong place.

As to Fenton & Pleeger's and Sheppard & Ince's comments about the merits of lines of code vs more complex measures such as Halstead/Mccabe, we saw above that lines of code can generate detectors with large variances across different applications (recall the between-application results of Figure 3). Also, we take issue with their use of correlation to assess detectors. Recall that Figure 5 results come from two equations with very different correlations to number of defects: -0.3616 and 0.65 for Equation 1 and Equation 4 and (respectively). Either equation can reach some desired level of detection, *regardless of their correlations*, merely by selecting the appropriate threshold value. For example, a $pd = 75\%$ can be reached using either method by setting $T \geq 0.65$ or $T \geq 0.45$.

More generally, we have found several commonly use assessment metrics to be uninformative about defect detectors. The problematic assessment measures are correlation, precision, and accuracy. Figure 3 showed that precision can vary wildly while other measures are more stable. Figure 5 showed that correlation can be insensitive to other measures like pd. Figure 6 shows a problem with accuracy. In that figure, hundreds of our detectors are shown sorted on increasing effort. Consider the detectors marked A and B on Figure 6. These two detectors have nearly the same accuracy, yet with $effort$s, PDs, and PFs that vary by factors as high as 4. That is, accuracy can be uninformative regarding issues of pf, pd, and $effort$.

In summary we are positive about static code defect detectors and others are not for several reasons. Firstly, we as negative as others about the merits of static code measures as a *primary* defect detection method. However, we are very positive about using static code defect detectors with low pfs as *secondary detectors* which can *augment* some other detection method. Secondly, we can demonstrate stable pf results across multiple applications. That is, if our secondary detectors trigger then it is highly unlikely that they are incorrectly reporting a detect. Thirdly, prior criticisms may be passed on problematic assessment measures such as correlation. We recommend using pf, pd, and $effort$ to assess detectors.

A drawback to this analysis is the sample size. While our work is based on a larger sample that some other publications in this area, more data is always better. We plan to frequently re-sample NASA's metrics data repositories to check our conclusions. This ability to revisit and revise old conclusions about software engineering is an important benefit of public domain code+defect repositories such as NASA's MDP program.

References

[1] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach*. International Thompson Press, 1997.

[2] M. Halstead. *Elements of Software Science*. Elsevier, 1977.

[3] T. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, Dec. 1976.

[4] T. Menzies, J. D. Stefano, K. Ammar, K. McGill, P. Callis, R. Chapman, and D. J. When can we test less? In *IEEE Metrics'03*, 2003. Available from `http://menzies.us/pdf/03metrics.pdf`.

[5] T. Menzies and J. S. D. Stefano. How good is your blind spot sampling policy? In *2004 IEEE Conference on High Assurance Software Engineering*, 2003. Available from `http://menzies.us/pdf/03blind.pdf`.

[6] T. Menzies, J. S. D. Stefano, C. Cunanan, and R. M. Chapman. The business case for defect logging. In *IEEE Transactions Software Engineering (in preperation)*, 2004.

[7] M. Shepped and D. Ince. A critique of three metrics. *The Journal of Systems and Software*, 26(3):197–210, September 1994.

[8] F. Shull, V. B. ad B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada*, pages 249–258, 2002. Available from `http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps`.

[9] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

Acknowledgements: The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility under NASA contract NCC2-0979 and NCC5-685.

Bug Report Networks: Varieties, Strategies, and Impacts in a F/OSS Development Community

Robert J. Sandusky Les Gasser Gabriel Ripoche
Graduate School of Library and Information Science
University of Illinois at Urbana-Champaign
{sandusky,gasser,gripoche}@uiuc.edu

Abstract

Our empirical research has shown that a predominant structural feature of defect tracking repositories is the evolving "bug report network" (BRN). Community members create BRNs by progressively asserting various formal and informal relationships between bug reports (BRs). In one F/OSS bug repository under study, participants assert two formal relationships (duplications and dependencies) and various informal relationships (like "see also" references).

BRNs can be interpreted as (1) information ordering strategies that support collocation of related BRs, decreasing cognitive and organizational effort; (2) sense-making strategies wherein BRNs provide more refined representations of software and work-organization issues; (3) social ordering strategies that rearrange collective relationships among community members. This paper presents findings from an investigation of the nature, extent, and impact of BRNs in one large F/OSS development community. We investigate whether and how specific classes of BRNs influence problem management within the community, and identify several new research questions.

1. Introduction

We are conducting empirical investigations into how F/OSS development communities manage software problems. The goal of our research is to develop models of how software problems are managed by large, distributed software development organizations. We aim to identify factors, such as *information*, *activity*, and *process*, which help explain better or worse software problem management (SWPM) performance, with the goal of both understanding such distributed collective practices and improving software production. The early stages of our work include qualitative analysis of the information used and activities performed by members of this community. We use this qualitative analysis to identify concepts, phenomena, and relationships between them as revealed through the examination of the bug reports created and managed by this community. The factors we identify can then be related to each other, hypotheses can be created, and the hypotheses can subsequently be tested in order to isolate the factors that affect SWPM performance. In addition, the seeds created from this human-based mining and analysis can be "computationally amplified," forming the basis of broader automated extraction of process models from very large corpuses of problem data [1].

The negative financial and social impacts of low quality software have been well documented [2, 3]. Previous research on software quality has focused on the development of metrics [4] and defect prediction models [5]. Other research has identified relationships between organizational structure, processes, and quality [6, 7, 8]. The SWPM process itself has been studied less frequently [9]. Our research approach, while grounded in empirical data, acknowledges the contributions of research on process and organizational issues.

Figure 1 shows the main elements of the bug report repository used by one community we are studying. The repository itself is a relational database system and a set of associated scripts that interact with the database and provide a Web-based user interface. The repository contains more than 235,000 records, referred to as bug reports (BRs). Each BR consists of (1) a number of fixed, vocabulary-controlled fields (e.g., status, resolution, severity), (2) several short-length text fields (e.g., keywords, summary), (3) attachments (e.g., screenshots, code patches), (4) a sequential series of text comments that are time-stamped and show the identity of the submitter, and (5) optional indications of relationships between BRs (e.g., duplication, dependency, and informal citations).

One of the most notable structural features of this community's bug report repository is the *bug report network* (BRN). A bug report network is created when members of the software community assert *duplication*, *dependency*, or *reference* relationships among bug reports. Duplication and dependency are both formal, symmetrical types of relationships with an explicit and codified representation in the bug reports. Community

members frequently create informal relationships, like "see also" references, by referring to other bug reports when they are adding text comments to existing bug reports. Sixty-five percent of the bug reports in this repository are associated with other bug reports using one of these three types of relationships.

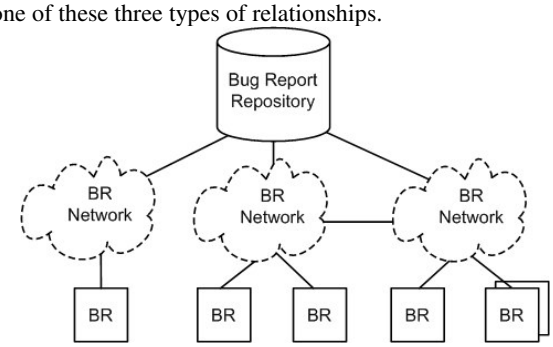


Figure 1. Bug report repository elements

Bug reports are first-class database objects, but bug report networks are not. Figure 1 shows three typical BRN patterns. The leftmost BRN represents the 35% of the bug reports that have zero formal or informal relationships to other bug reports: each of these bug reports forms a trivial BRN. The middle BRN shows two bug reports associated by a dependency or informal relationship. The rightmost BRN shows a more complex set of relationships. The doubled bug report on the right represents a bug report with a duplicate relationship to the second bug report behind it. The duplicated bug report is also associated by either a dependency or informal relationship with the other bug report in the network. Note that it is also possible for BRNs to be connected to each other, as indicated by the line connecting the middle and rightmost networks.

We use the following definitions, based upon the definitions stated by the community in their SWPM documentation, to identify the formal relationships between bug reports:

- **Duplicate**: A bug report is marked as a *duplicate* if the problem represented by the bug report is believed to be already represented by another bug report. A duplicate relationship is a formal, symmetrical relationship between two bug reports.
- **Dependency**: A bug report is marked as a *blocker* of another bug report if resolution of the software problem it represents blocks development and/or testing work on the problem represented by the other bug report. A bug report is marked as *dependent on* another bug report if the problem it represents can't be fixed until the problem represented by the other bug report is fixed. Bug reports that are dependent on each other have a formal, symmetrical "blocks" / "depends on" relationship.

Community members also frequently assert informal references between bug reports. While it is possible to automatically extract instances of informal relationships from the repository, the nature and purpose of these references vary considerably and are most reliably understood by reading the bug reports and understanding the contexts in which the citations are made. Here are some examples of these informal references:

```
- This looks related to #X
- See comments on X -- same applies here
  I think.
- My fix for X kinda helps fixing this
  too.
- Should bug X be added to this?
```

2. Method

A random sample of 385 BRs was systematically drawn from a population of more than 182,000 bug reports opened over a five year period. The bug report is the primary unit of analysis in this study. The sample size was determined using an approach reported by Powell [10] (p.75). A conservative sample size was suitable here because we did not have complete information about the variability of all characteristics of the bug reports at the time the sample was drawn.

2.1. Qualitative analysis

Each bug report in the sample was treated as a text and was read and analyzed using a content-analytic approach [11]. Concepts, phenomena, and relationships between phenomena were identified and refined as they emerged from the bug reports during data analysis using grounded theory [12]. References to other BRs were noted (their location within the BR, reference type, BR serial number) as each BR in the sample was analyzed.

2.2. Automatic processing

Another characterization of bug report networks was attempted using automatic extraction of relationships in a snapshot of over 130,000 bug reports originating from the same bug report repository. Two types of relationships were considered:

- *Formal* relationships identified by specific fields ("blocks" and "depends on") or computer generated output inserted as comments in the bug report's discussion.
- *Informal* relationships in the form of references made by participants in their comments (e.g.: "`See bug #X`", "`Looks like bug Y`", etc.), which were mined using regular expressions.

The processing yielded relationship matrices from which BRNs could be identified. However, the automated processing was less accurate than content analysis. When we compared the automatic and manual processes, we found that the automatic process completely identified all the "informally" connected BRs 40% of the time. Also, our current extraction approach does not allow for the distinction of the various types of relationships that are being established. Results from the qualitative analysis are being used to improve the regular expressions used to automatically identify the informal relationships.

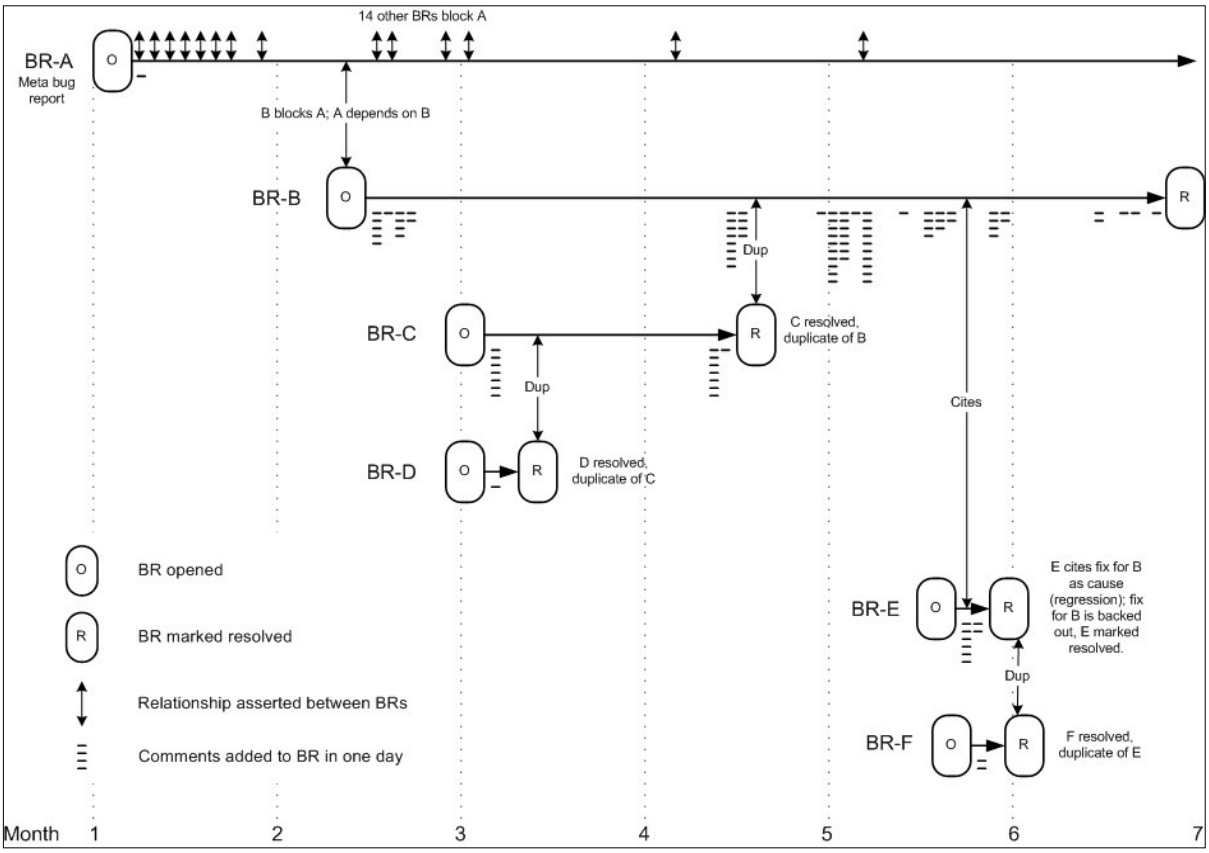


Figure 2. Bug report network

3. Anatomy of a bug report network

Figure 2 represents the bug report network associated with one "critical" severity bug report drawn from the bug report repository under study. This bug report network, consisting of six bug reports, illustrates a number of different relationships that often occur in this repository. The x-axis represents time over a 6-month period; the relationship of the objects in the diagram to the timescale is approximate. The columns of dashes below each bug report's lifeline represent the count of comments added to a bug report on a single day and are shown to provide a sense of the level of activity associated with each bug report throughout the bug report's life.

Bug report "B" (BR-B in the diagram) is the central report in this network. BR-B was opened with a "critical" severity level because it represented a bug that caused the software system to crash. As soon as it was opened, it was associated as "blocking" the resolution of BR-A. BR-A already existed, and was defined as a *meta* bug used to collocate a group of 15 (including BR-B) bug reports representing bugs that caused crashes in this part of the overall system. (Note that it would be possible to look at BR-A as the central BR in a different BRN: this is an example of how BRNs can be connected to each other as shown in Figure 1. See discussion of meta bug report networks below.)

The chain of duplicate relations between BR-B, BR-C, and BR-D is of interest. BR-C was opened several weeks after BR-B, during a six-week period when BR-B was not

very active (no comments added). BR-D was opened later the same day BR-C was opened. BR-D was quickly identified as representing the same bug as BR-C and marked "resolved/duplicate." BR-C was not identified as representing the same phenomena as BR-B until about six weeks after BR-C was opened.

The relationships between BR-B, BR-E, and BF-F are also of interest. The level of activity on BR-B was high during month 5. At one point, a patch for the bug represented by BR-B was introduced. This change caused the bug represented by BR-E (a type of bug and bug report identified as a "regression") to occur. BR-F was opened a couple of hours after BR-E was opened, and was immediately recognized as a duplicate of BR-E. The bad patch associated with BR-B was quickly backed out to resolve the problem associated with BR-E. BR-E was then marked "resolved/fixed."

4. Varieties, strategies, and impacts

The kinds of relationships found between bug reports, their frequency of occurrence, BRN strategies, implications of the construction and use of BRNs, and future work are discussed in this section.

4.1. Varieties

Almost two-thirds (65%) of the 385 bug reports in the sample have either a formal or informal relationship with at least one other bug report. Table 1 shows the frequency with which different types of relationships occur within the sample of 385 bug reports.

Table 1. Frequency of relations in sample

Duplicates	
BRs with one or more duplicate BRs	10%
BRs resolved as a duplicate of another BR	33%
Dependencies	
BRs "blocking" one or more BR	12%
BRs "dependent on" one or more BR	7%
Informal	
BRs with "informal" relation to one or more BR	33%

Community members sometimes create bug reports that, instead of representing problems (bugs), anchor a collection of bug reports having common characteristics (e.g., all the high priority bug reports that should be fixed prior to the next software release). Community members refer to these anchor bug reports as "meta" or "tracking" bug reports. In the BRN illustrated in Figure 2, BR-A is a "meta" bug report used to create a network of bug reports representing system crashes of a similar type. Creation of a meta bug report and its associated BRN represents a specific social and information management adaptation made by community members to increase the utility of the bug report repository.

4.2. Strategies

Constructing BRNs is an information structuring strategy. Individual bug reports, first-class database objects, are composed over time into a new form of information, the BRN. Creating a BRN collocates a group of bug reports that would otherwise remain scattered and disassociated from each other. A BRN thus adds virtual structure to the bug report repository. Collocating information by adding or imposing structure is a complexity management / complexity reduction technique.

Asserting that a bug report is a duplicate of another bug report, for example, shrinks the set of bug reports that must be worked on. Shrinking the set of bug reports to work on reduces the complexity of the field of work. However, identification of duplicates is costly because members of the community must identify duplicates manually. There is also danger of mis-identification: bug reports that are not true duplicates (false positives) will be ignored because their status is "resolved." It's also clear, because of late-marked duplication and duplication time inversion, that "undiscovered" duplicate bug reports exist and multiple groups of people may be duplicating effort by working on two bug reports that represent the same issue (see, for example, the time period between the opening of BR-C and its resolution as a duplicate of BR-B in Figure 2.)

We also suspect that there are patterns of BRNs, for example patterns in the kinds of links that appear, and patterns in the types of links that are sanctioned and even crystallized into standard categories and supporting tools, such as dependencies and duplicates. There may also be patterns in how such networks are formed.

4.3. Impacts

As BRN construction orders information, it also orders social relations. BRs are specifications/codifications of social relationships, such as roles (reporter, assigned-to, cc: list member) and dynamic and patterned interactions (e.g. dialogues, question-response-elaboration sequences; negotiation; coordination of work, etc.). This means that as information is ordered through BRN creation/extension/modification, social relations are also being ordered. The impacts of this kind of social reordering might vary. In some cases, time-to-resolution may be improved by bringing more resources to bear upon a problem. In other cases, performance might deteriorate if, for example, the cost of coordinating the activities of more people slows progress toward resolution.

When a bug report is marked "resolved/duplicate" this means the bug *report* is resolved but it does not mean that the underlying bug itself has been resolved. "Resolved/duplicate" means that the resolver(s) believe

this is a duplicate report of a phenomenon that already has an effective representation elsewhere in the repository. It doesn't even mean that that the resolved bug report can now be ignored, since we have seen instances of late-identification of duplicates (e.g., BR-C in Figure 2) in which accumulated knowledge and dialogue may still be relevant to the resolution of the other bug reports in the BRN. Thus the semantics of the "resolved" keyword are clearly complex.

4.4. Future Work

Our work on understanding and identifying bug report networks has just begun. Many challenges remain, including:

- Identifying the situations in which BRNs are helpful (or unhelpful) in managing software problems; understanding the extent to which complex BRNs are taken into account by community members during problem resolution.
- Determining if BRNs are present in all bug report repositories; how the capabilities of different repositories and the conventions developed by the different communities influence the use of BRNs.
- Quantifying the range of complexity of BRNs in this and other bug report repositories; identifying the most useful metrics for measuring the size and complexity of BRNs (for example, a BRN can be thought of as a graph, with each bug report as a vertex in the graph).
- Developing useful representational forms (e.g., Figure 2) for BRNs that can contribute to our understanding and increase the utility of BRNs as a tool for SWPM.
- Determining how the inclusion of a BR in a BRN affects the community's SWPM performance (e.g., testing for a correlation between BRN membership and time to resolution).

Automatic extraction and representation of BRNs will be an important part of addressing the research questions raised here. The practical application of results of this research to software engineering practice also depends upon the development of effective and scalable automatic extraction and representation techniques. Challenges related to automatic extraction and representation include:

- Improving techniques for automatically extracting and representing BRNs from a bug report repository.
- Develop computational tools to discover and formalize the latent, undiscovered relationships between bug reports.

5. Conclusion

The analysis performed so far demonstrates that bug report networks are common in the bug report repository studied here: 65% of the bug reports sampled are part of a BRN. Members of this community commonly use the formal, symmetrical relationships of duplication and dependency as well as a wide variety of informal relationships. BRNs are a common and powerful means for structuring information and activity. BRNs, however, have not yet been the subject of concerted research by the software engineering community. The continuation of this stream of research will result in a more complete understanding of the contribution BRNs make to effective software problem management.

6. References

[1] Gasser, L., & Ripoche, G. (2003). Distributed collective practices and F/OSS problem management: perspectives and methods. *CITE'03,*Troyes, France, December 2003.

[2] NIST. (2002). *The economic impacts of inadequate infrastructure for software testing: final report.* May 2002. Planning report 02-3. Gaithersburg, MD: NIST.

[3] Leveson, N. & Turner, C.S. (1993). An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7), 18-41.

[4] Osterweil, L. (1996). Strategic directions in software quality. *ACM Computing Surveys*, 28(4), 738-750.

[5] Fenton, N. E., & Neil, M. (1999). A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5), 675-689.

[6] Conway, M.E. (1968). How do committees invent? *Datamation*, 14(4), 28-31.

[7] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1053-1058.

[8] Herbsleb, J., Zubrow, D., Goldenson, D., Hayes, W., & Paulk, M. (1997). Software quality and the capability maturity model. *Communications of the ACM*, 40(6), 30-40.

[9] Crowston, K. (1997). A coordination theory approach to organizational process design. *Organization Science*, 8(2), 157-175.

[10] Powell, R.R. (1991). *Basic research methods for librarians*. (2nd ed.). Norwood, NJ: Ablex.

[11] Weber, R. P. (1990). *Basic content analysis*. (2nd ed.). Newbury Park, CA: Sage.

[12] Strauss, A., & Corbin, J. (1990). *Basics of qualitative research: grounded theory procedures and techniques*. Newbury Park, CA: Sage.

A Tool for Mining Defect-Tracking Systems to Predict Fault-Prone Files

Thomas J. Ostrand
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
ostrand@research.att.com

Elaine J. Weyuker
AT&T Labs - Research
180 Park Avenue
Florham Park, NJ 07932
weyuker@research.att.com

Abstract

In earlier research we identi ed characteristics of les in large software systems that tend to make them particularly likely to contain faults. We then developed a statistical model that uses historical fault information and le characteristics to predict which les of a system are likely to contain the largest numbers of faults. Testers can use that information to prioritize their testing and focus their efforts to make the testing process more ef cient and the resulting software more dependable. In this paper we describe a proposed new tool to automate this prediction process, and discuss issues involved in its design and implementation. The goal is to produce an automated tool that mines the project defect tracking system and that can be used by testers without requiring any particular statistical expertise or subjective judgements.

Keywords: Prediction Tool, Software Faults, Software Defects, Fault-prone, Software Testing

1 Background

Change tracking systems are commonly used within software development projects to allow the development group to record software faults (also known as defects) and the actions taken to repair them, as well as other changes made to a software system for any reason at all. If the change information is recorded accurately and kept up-to-date, the team is always aware of the system's status, knows what problems are currently outstanding, and can make sound decisions about whether the product is ready for release. Fault history also provides valuable data for analysis of project trends. A new project can be evaluated against the fault patterns of previous related projects, and changes made to the project's methods, tools, or organization if fault rates are significantly higher than expected.

Many large AT&T software projects use an integrated change tracking/version control system, which makes both the project code and the change information accessible to developers and testers. The version control part of the system maintains the code base of the development project. A system administrator sets up an initial framework of the project in the system, and programmers then write code and check it in under version control. When the programmer wants to modify code, s/he checks out the needed files, makes the changes, and then checks the code back in.

The change tracking part of the system maintains a database of *modi cation requests* (MRs) that have been written for the project under development. An MR can be written by any member of the project team, for reasons that include creating initial code for the project, adding code to implement new functionality, performing maintenance updates to the project, and reporting failures or incorrect behavior. Typically, developers write the first three types of MRs, and testers write the last type, although developers may also detect and report incorrect behavior. Failure-reporting MRs are sent to the developer team, which may use the description of the problem to determine the most appropriate developer to fix the problem.

In order to avoid confusion between references to a software system that is using the change tracking system, and references to the change tracking system itself, in the remainder of this paper we use the term *system* to refer to the change tracking system, and we use the term *project* to refer to a system under development.

Change tracking involves accessing a database that records relevant information about any changes that are made to the software. We are particularly interested in changes that are made because of the presence of faults, and we will therefore sometimes refer to the relevant portions of the combined change tracking/version control system simply as the *defect tracking system*. For every detected failure, an entry is made in the defect tracking system describing the problem and the change made to fix it, including a text description of the problem, the names of the MR writer and the developer who ultimately fixed the problem, and the specific files that were modified in response to the MR.

Our work has used the information in the AT&T defect tracking databases of several large software projects to develop a model that can be used to guide the testing of future releases of the projects. We analyzed the historical change information for these projects, categorizing faults and identifying code characteristics that are associated with faults. Using the fault information from past project releases, we constructed a statistical fault prediction model for the files of future releases [3]. As described in that paper, for one large AT&T project with 17 successive quarterly releases, the model identified 20% of the files in each new release that contain between 71% and 92% of the faults identified in that release. The average over all of the releases considered was 83%.

This prediction can be very useful to help system testers focus their efforts on the parts of the project where defects are most likely to be located. Of course, it does not remove the obligation to assure that all parts of the project have been adequately tested, but knowing that most of the problems will show up in a relatively small portion of the code means that it should be profitable to test that portion earlier and more intensively than other parts. Time available for testing is always limited, and testing the most fault-prone areas early should reveal more faults earlier. As a result, more of the precious testing time can be allocated to areas that may otherwise not be covered as thoroughly, possibly leading to software with higher reliability with the same use of testing resources. Another possibility is that testing time could actually be reduced, as the tester might uncover the same faults more quickly, leading to the same degree of reliability sooner and more cheaply than would otherwise be possible.

The fault prediction is done using a negative binomial regression model whose independent variables are characteristics of the individual files in a release. The variables include the number of lines of code in the file; whether the file is new, changed, or unchanged; the file's age (the number of releases it has previously appeared in); the number of faults found in the previous release; and the file's source language.

2 A Proposed Fault Prediction Tool

Once the mathematical framework of the model has been established, it is straightforward but tedious to apply it to the specific values for a particular release, to derive the fault-proneness predictions for individual files in the next release. We therefore propose constructing a testing tool that incorporates the prediction model, and that will allow testers to generate an ordered list of the most fault-prone files without requiring the intervention or assistance of a professional statistician, or any particular understanding of the statistics involved .

The tool assigns each file of the next release an expected number of faults, and then ranks the files in decreasing order of that number. In our earlier case study [3], we found that the total number of faults predicted for even relatively small sets of files became quite accurate. The tool will allow the tester to request sets of files that are specified either by the percentage of the project that they constitute, or by the expected percentage of faults that they will contain.

For example, the tester could ask for a list of the 20% of the next release's files that are predicted to have the most faults, as was shown for the sample software project analyzed in [3]. However, since the predictor ranks all the project's files in order of their number of predicted faults, the user can just as easily ask for the top T% of that listing for any value T.

Conversely, the user could request a listing of the minimum set of files that are predicted to contain at least P% of the faults in the release. In [3], we found that over the seventeen project releases, the top 20% of files selected by the model included from 71% to 92% of the actual faults in that release, with an overall average of 83%. If we had always wanted to identify a minimal set of files that were projected to contain at least 80% of the faults, then for some releases the model would have identified fewer than 20% of the files, and for other releases more than 20% of the files.

3 Tool Components

The tool will have three main components.

The first component extracts information about faults in the current release from the change tracking database. This information includes the names of specific files that were changed to repair each fault. This component requires an accurate determination of which entries in the database actually represent faults. We found that it was often difficult to accurately identify which changes were made because of faults and which were made for some other reason such as a new feature or a planned enhancement.

Although the same change control system is used by many different AT&T projects, different project teams use the system in different ways, and provide different information regarding the motivation behind a change. For example, one team we worked with uses an MR field that specifies either the group that raised the issue behind a change, or indicates that the change was due to maintenance or an enhancement. Maintenance and enhancement changes are clearly not faults. When the group that raised the issue was the system test group, the MR was almost certainly created because of a failure, and the change was almost certainly to repair a fault. In other cases, such as a customer-requested change, it was sometimes difficult to determine whether or not a change was due to a fault without reading through the entire text of the modification request entry.

Without an explicit identification of fault/no-fault in each report, the best approach is to analyze the text description of the change report. The reporter is expected to write a brief description of the original problem, which is usually (although not always) adequate to determine if the change is fault-based. If the description is ambiguous, and the reporter is accessible, then a personal interview can usually resolve the situation. But this requires a large amount of human effort and can be prohibitively expensive if there are many faults listed in the defect tracking system.

In a second study that we carried out [3], there were approximately 500 change reports to analyze, and the description analysis approach was feasible. However, in our original study [2], many thousands of MRs were entered into the change tracking system, and over 5000 of them were determined to be faults for the first twelve releases of the project. This was certainly too many to analyze individually, or to question each of the reporters. For that project, after discussions with the project test manager, we defined a heuristic that we used to determine what was a fault. We will further discuss the problem of how to decide whether a change report represents a defect in the next section.

After the faults have been identified, the tool's second component goes to the code base to extract properties of the files. The model uses properties of all files in the project, not just those that have been modified. Since only a small percentage of the files are usually modified in a given release, the tool can maintain a table of file properties for the entire project, and update only the entries for the files that have been modified as a result of fault detection. The file characteristics used by the model include the file size, the number of faults in previous releases, the file's change status, the file's age, and the programming language used. The first three of these can change during development and debugging of a release, and should be checked and updated for any files that are mentioned in a change report. The file's age simply increases by one with each successive release, and the programming language does not change.

Extracting all this data about files and faults is facilitated by the fact that the defect tracking system and the version control system are integrated in a single system.

Our current model does not use other static properties such as cyclomatic number [1] and inter-module coupling. Our original case study found that including the cyclomatic number in the model did not improve its predictive ability over that of lines of code. We have not yet assessed the predictive ability of module coupling. If additional studies determine that using these characteristics improve the model's accuracy, they may be incorporated into future versions, and can also be checked from the code base.

The tool's third component makes the fault predictions, using the fault and file information gathered in the previous steps, as well as the model's earlier calculations. Once the file characteristics are known, they can be supplied as input to the model, and the predictions for the next release produced.

4 Designing the Predictor Tool

As presently implemented, the fault-proneness prediction is a multi-step process with human intervention at several key points. These include identifying the particular project MRs that represent faults; identifying the files that have been modified to fix a fault MR; and obtaining properties of the modified files.

The testing tool will replace the human intervention with scripts that implement these required activities. In this section we discuss the issues that have to be solved to accomplish this.

4.1 What is a Defect?

As mentioned above, the tool's database extractor component needs a means to determine which change reports represent defects. We originally thought we could base this decision on the report's *category*. Different software projects use this field in different ways, but it always partitions the changes into the three classes of *enhancement, maintenance* and *modi cation*. One project uses the category to provide additional information about modifications, by identifying where in the development/testing process the need for a change was originally determined. The possible values are *development, system test, user acceptance test* and *customer*.

At first glance, it seems clear that enhancement and maintenance changes should not be counted as faults, while the modification changes are all faults.

Unfortunately, we have found that certain fields in the change report, including the *category*, are frequently not filled out accurately. Testers are always under pressure to complete work quickly, and testing time is always at a premium. We found that creators of modification requests often leave the default values in place for fields of the MR that they believe are not important.

Since the category field is frequently unreliable, we have begun to identify faults by examining the job category of the person who created the modification request. If this person is an integration or system tester, then the modification request invariably represents a fault. When developers initiate an MR, it may be necessary to understand the culture of the development group and the nature of the requested change, or it may be necessary to simply read all requests initiated by developers to determine whether they are new features, enhancements, or actual faults.

Developers can report two types of problems. The first type occurs when a developer writes a modification request

against his or her own code, after the developer discovers a problem during unit testing. In some projects, developers note all such faults, and the resulting changes are recorded in the database. More commonly, unit testing changes are never recorded; the developer simply keeps the code checked out during the unit testing phase, makes all changes to the single checked-out version, and only checks the code in once, when it's judged ready for integration. If reporting unit test faults is part of the development group culture, then the change reports based on them should be included as input to the predictor.

Developers can also write a modification request when they detect a problem or inconsistency in the project specification. In this case, the developer is actually testing and finding a defect in the specification. There may not be any code yet written, and hence none to be changed, or the developer might have written code originally based on a mistaken understanding of the specification. These specification MRs represent real faults that have to be corrected, and they should be included as input to the predictor.

However, many changes initiated by developers are not defect fixes, but are enhancements, new features, or modifications to keep the project consistent. Separating these from the fault-based changes requires more detailed analysis of the change description, or a face-to-face meeting with the developer.

This situation leads us to the third, and best, means of identifying a change report as originating with a discovered defect: ask the person who has written the change report. To help collect this answer as painlessly as possible, the management of one project has agreed to augment its MR creation form with an explicit fault-classification field with 3 possible values: "Fault", "no fault", "unknown". To avoid false answers generated by testers or developers in a hurry, the field has no default value. Future users of the change report system would be expected to fill out this field. This simple addition to the MRs should both improve the accuracy of the data, and hence the accuracy of the prediction, and also simplify the data mining process, making it possible to automate this portion of the data extraction.

Since the fault-classification field has been added only very recently, we do not yet have any analysis results for change reports that use it. However, we do believe that it will be an essential part in fulfilling our goal of building a tool to automate the identification of the most fault-prone files of a project.

4.2 Obtaining Properties of Files

Obtaining file properties is a two-step process: first, the fault MRs have to be interrogated to identify the proper files, and second, those files have to be located in the code base and analyzed.

The AT&T defect tracking system is a proprietary relational database that makes information available as either a text file report or an Excel spreadsheet. The Excel format allows users to extract certain large classes of change reports and feed their information directly into a spreadsheet. Once the entries are in the spreadsheet, it is quite simple to create tables that show relations between different attributes of the change reports, and that can summarize totals of different types of reports. Unfortunately, the current version of the extraction into Excel only includes those fields of the change report that have a fixed number of possible values. While this covers many of the change report fields, it does not include a list of files that have been changed, since there can be arbitrarily many of them.

The text reports are produced by a set of specialized commands that can search for data and create text files with varying levels of detail. These commands can be used to access all the database's information, but apart from specifying which details should be included, the user has little control over the format of the text report. This means that post-processing programs must be written to extract the particular details that are needed for our fault-proneness prediction. In particular, we have to search through the report to find the text strings that name the files that are associated with each fault. We presently do this search with stand-alone shell scripts that are run by a human on the change system's text reports. The tool will integrate the change system's commands with these scripts to remove the necessity for human intervention.

Once the files are identified, we again use stand-alone scripts initiated by a human to extract their static properties. The scripts are run over the latest build of the project code, and include simple line and character counts. More complex code metrics, like the cyclomatic number, have also been computed, although our current prediction model does not use them. A possible structure for the predictor tool is to run these code analyses independently, to build and maintain a table of their values. The tool can then use the table entries whenever they are needed to produce fault-proneness results.

5 Using On-Demand Fault Predictions

The original intent of the fault-proneness predictor was to provide guidance for testers when testing starts on a new release, helping them to focus their efforts on a small percentage of the files that the model predicts are most likely to contain faults. In this scenario, the fault-proneness prediction for release N+1 would be made as close as practical to the beginning of system testing for N+1, to take maximum advantage of the fault data generated by the testing of release N.

However, we can also envisage a tool that would be

able to produce the model predictions on demand at any time during the development and testing of a given release, based on the latest information that has been entered into the change database, and the latest configuration of the code.

The availability of on-demand fault-proneness prediction would allow testing guidance based on release N to be given during the testing of release N. A possible scenario would be to run the on-demand prediction every night after the system testing phase has started. This would augment all the prior fault data with the current day's failure and fault information, and would provide the testers each morning with fresh guidance on where to concentrate testing efforts. The daily prediction should not be used to completely revise the project's original test plan. Rather, its information should be viewed as supplemental, giving advice about potential serious trouble areas in the code.

6 Summary

The tool proposed in this paper is an outgrowth of our studies of defects and characteristics of files containing defects reported for large AT&T software projects. It provides an automated framework for the fault-proneness prediction model that we developed earlier, and will allow testers and developers to generate and use the prediction results without the assistance or intervention of specialists.

The issues encountered in designing the tool include how to identify modification requests that represent software defects, how to make use of data extracted from the repository to interrogate the development project's code base, and how to present the tool's capabilities to its potential users in the most useful way.

The tool design is presently at a very early stage, but the success of the prediction model in our case studies encourages us to believe that this will eventually become a highly useful element of the system tester's toolkit.

References

[1] T.J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, Vol 2, 1976, pp. 308-320.

[2] T. Ostrand and E.J. Weyuker. The Distribution of Faults in a Large Industrial Software System. *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA2002)*, Rome, Italy, July 2002, pp. 55-64.

[3] T. Ostrand, E.J. Weyuker, and R.M. Bell. Where the Bugs Are. *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA2004)*, Boston, MA, July 2004.

Towards Understanding the Rhetoric of Small Changes
-- Extended Abstract --

Ranjith Purushothaman
Server Operating Systems Group
Dell Computer Corporation
Round Rock, Texas 78682
ranjith_purush@dell.com

Dewayne E. Perry
Electrical & Computer Engineering
The University of Texas at Austin
Austin, Texas 78712
perry@ece.utexas.edu

Abstract

Understanding the impact of software changes has been a challenge since software systems were first developed. With the increasing size and complexity of systems, this problem has become more difficult. There are many ways to identify change impact from the plethora of software artifacts produced during development and maintenance. We present the analysis of the software development process using change and defect history data. Specifically, we address the problem of small changes. The studies revealed that (1) there is less than 4 percent probability that a one-line change will introduce an error in the code; (2) nearly 10 percent of all changes made during the maintenance of the software under consideration were one-line changes; (3 the phenomena of change differs for additions, deletions and modifications as well as for the number of lines affected.

1. Introduction

Change is one of the essential characteristics of software systems [1]. The typical software development life cycle consists of requirements analysis, architecture design, coding, testing, delivery and finally, maintenance. Beginning with the coding phase and continuing with the maintenance phase, change becomes ubiquitous through the life of the software. Software may need to be changed to fix errors, to change executing logic, to make the processing more efficient, or to introduce new features and enhancements.

Despite its omnipresence, source code change is perhaps the least understood and most complex aspect of the development process. An area of concern is the issue of software code degrading through time as more and more changes are introduced to it – code decay [5]. While change itself is unavoidable, there are some aspects of change that we can control. One such aspect is the introduction of defects while making changes to software, thus preventing the need for fixing those errors.

Managing risk is one of the fundamental problems in building and evolving software systems. How we manage the risk of small changes varies significantly, even within the same company. We may take a strict approach and subject all changes to the same rigorous processes. Or we may take the view that small changes generally have small effects and use less rigorous processes for these kinds of changes. We may deviate from what we know to be best practices to reduce costs, effort or elapse times. One such common deviation is not to bother much about one line or other small changes at all. For example, we may skip investigating the implications of small changes on the system architecture; we may not perform code inspections for small changes; we may skip unit and integration testing for them; etc. We do this because our intuition tells us that the risk associated with small changes is also small.

However, we all know of cases where one line changes have been disastrous. Gerald Weinberg [9] documents an error that cost a company 1.6 billion dollars and was the result of changing a single character in a line of code.

In either case, innocuous or disastrous, we have very little actual data on small changes and their effects to support our decisions. We base our decisions about risk on intuition and anecdotal evidence at best.

Our approach is different from most other studies that address the issue of software errors because we have based the analysis on the property of the change itself rather than the properties of the code that is being changed [7]. Change to software can be made by addition of new lines, modifying existing lines, or by deleting lines. We expect each of these different types of change to have different risks of failure.

Our first hypothesis is specific to one-line changes, namely that the probability of a one-line change resulting in an error is small. Our second hypothesis is that the failure probability is higher when the change involves

adding new lines than either deleting or modifying existing lines of code.

To test our hypotheses, we used data from the source code control system (SCCS) of a large scale software project. The Lucent Technologies 5ESS™ switching system software is a multi-million line distributed, high availability, real-time telephone switching system that was developed over two decades [6]. The source code of the 5ESS project, mostly written in the C programming language, has undergone several hundred thousand changes.

Our primary contribution in this empirical research is an initial descriptive and relational study of small changes. We are the first to study this phenomenon. Another unique aspect of our research is that we have used a combination of product measures such as the lines of code and process measures such as the change history (change dependency) to analyze the data. In doing so, we have tried to gain the advantages of both measures while removing any bias associated with each of them.

While several papers discuss the classification of changes based on its purpose (corrective, adaptive, preventive) there is virtually no discussion on the type of change: software can be changed by adding lines, deleting lines or by modifying existing lines. As a byproduct of our analyses, we have provided useful information that gives some insight into the impact of the type of change on the software evolution process.

2. Background – Change Data Description

In the 5ESS, a *feature* is the fundamental unit of system functionality. Each feature is implemented by a set of Initial Modification Requests (IMRs) where each IMR represents a logical problem to be solved. Each IMR is implemented by a set of Modification Requests (MRs) where each MR represents a logical part of an IMR's solution. The change history of the files is maintained using the Extended Change Management System (ECMS) (as shown in Figure.1 [3][5][7]) for initiating and tracking changes and the Sources Code Control System for managing different versions of the files. The ECMS records information about each MR. Each MR is owned by a developer, who makes changes to the necessary files to implement the MR. The changes themselves are maintained by SCCS in the form of one or more *deltas* depending on the way the changes are committed by the developer. Each delta provides information on the attributes of the change: lines added, lines deleted, lines unchanged, login of the developer, and the time and date of the change.

While it is possible to make all changes that are required to be made to a file by an MR in a single delta, developers often perform multiple deltas on a single file for an MR. Hence there are typically many more records in the delta relation than there are files that have been modified by an MR.

The 5ESS™ source code is organized into subsystems, and each subsystem is subdivided into a set of modules. Any given module contains a number of source lines of code. For this research, we use data from one of the subsystems of the project. The Office Automation (OA) subsystem contains 4550 modules that have a total of nearly 2 million lines of code. Over the last decade, the OA subsystem had 31884 modification requests (MR) that changed nearly 4293 files. So nearly *95 percent of all files were modified* after first release of the product.

Change to software can be introduced and interpreted in many ways. However, our definition of change to software is driven by the historic data that we used for the analysis: A change is *any alteration to the software recorded in the change history database* [5]. In accordance with this definition, in our analysis the following were considered to be changes:

- One or more modifications to single/multiple lines;
- One or more new statements inserted between existing lines;
- One or more lines deleted; and,
- A modification to a single/multiple lines accompanied by insertion or/and deletion of one or more lines.

The following changes would qualify to be a one-line change when an MR consists of either:

- One or more modifications to a single line;
- One or more lines replaced by a single line;
- One new statement inserted between existing lines; or,
- One line deleted.

Previous studies such as [14] do not consider deletion of lines as a change. However, from preliminary analysis, we found that lines were deleted for fixing bugs as well as making modifications. Moreover, in the SCCS system, a line modification is tracked as a line deleted and a line added. Hence in our research, we have analyzed the impact of deleting lines of code on the software development process.

3. Approach

In this section, we document the steps we took to obtain useful information from our project database. We first discuss the preparation of the data for the analysis and then explain some of the categories into which the

data is classified. The final stage of the analysis identifies the logical and physical dependencies that exist between files and MRs.

3.1 Data Preparation

The change history database provides us with a large amount of information. Since our research focuses on analyzing one-line changes and changes that were dependent on other changes, one of the most important aspects of the project was to derive relevant information from this data pool. While it was possible to make all changes that are required to be made for a MR in a file in a single delta, developers often performed multiple deltas on a single file for an MR. Hence there were lot more delta records than the number of files that needed to be modified by MRs.

In the change process hierarchy, an MR is the lowest logical level of change. Hence if the MR was created to fix a defect, all the modifications that are required by an MR would have to be implemented to fix the bug. Hence we were interested in change information for each effected file at the MR level. For example, in Table 1, the MR *oa101472pQ* changes two files. Note that the file *oaMID213* is changed in two steps. In one of the deltas, it modifies only one-line. However, this cannot be considered to be a one-line change since for the complete change, the MR changed 3 lines of the file. With nearly 32000 MRs that modified nearly 4300 files in the OA subsystem, the aggregation of the changes made to each file at the MR level gave us 72258 change records for analysis.

Table 1: Delta relation snapshot

DELTA relation				
MR	**FILE**	**Add**	**Delete**	**Date**
Oa101472pQ	oaMID213	2	2	9/3/1986
Oa101472pQ	oaMID213	1	1	9/3/1986
Oa101472pQ	oaMID90	6	0	9/3/1986
Oa101472pQ	oaMID90	0	2	9/3/1986

3.2. Data classification

Change data can be classified based on the purpose of the change and also based on how the change was implemented. The classification of the MRs based on the change purpose was derived from the work done by Mockus and Votta [3]. They classified MRs based on the keywords in the textual abstract of the change. For example, if keywords like 'fix', 'bug', 'error', and 'fail' were present, the change was classified as corrective. In Table 2 we provide a summary of the change information classified based on its purpose. The naming convention is similar to the work done in their original paper.

However, there were numerous instances when changes made could not be classified clearly. For example, certain changes were classified as 'IC' since the textual abstract had keywords that suggested changes from inspection (I) as well as corrective changes (C). Though this level of information provides for better exploration and understanding, in order to maintain simplicity, we made the following assumptions:

- Changes with multiple 'N' were classified as 'N'
- Changes with multiple 'C' were classified as 'C'
- Changes containing at least one 'I' were classified as 'I'

Table 2: Change Classification (purpose)

ID	Change type	Change purpose
B	Corrective	Fix defects
C	Perfective	Enhance performance
N	Adaptive	New development
I	Inspection	Following inspection

Changes which had 'B' and 'N' combinations were left as 'Unclassified' since we did not want to corrupt the data. Classification of these as either a corrective or adaptive change would have introduced validity issues in the analysis. Based on the above rules, we were able to classify nearly 98 percent of all the MR into corrective, adaptive or perfective changes.

Table 3: Change classification (implementation)

ID	Change Type	Description
C	Modify	Change existing lines
I	Insert	Add new lines
D	Delete	Delete existing lines
IC	Insert/Modify	Inserts and modifies lines
ID	Insert/Delete	Inserts and deletes lines
DC	Delete/Modify	Deletes and modifies lines
DIC	All of the above	Inserts, deletes and modifies lines

Another way to classify changes is on the basis of the implementation method into insertion, deletion, or modification. But the SCCS system maintains records of only the number of lines inserted or deleted for the change and not the type of change. Modifications to the existing lines are tracked as old lines being replaced by new lines (delete and insert). However, for every changed file SCCS maintains an SCCS file that relates the MR to

the insertions and deletions made to the actual module. Scripts were used to parse these files and categorize the changes made by the MR into inserts, deletes or modifications. Table 3 lists different types of changes based on their implementation method.

3.3 Identifying file dependencies

Our primary concern was in isolating those changes that resulted in errors. To do so, we identified those changes that were *dependencies – changes to lines of code that were changed by an earlier MR*. If the latter change was a bug fix our assumption was that the original change was in error. The one argument against the validity of this assumption would be that the latter change might have fixed a defect that was introduced before the original change was made. However, in the absence of prima facie evidence to support either case, and since preliminary analysis of some sample data did not support the challenging argument, we ruled out this possibility. In this report, we will refer to those files in which changes were made to those lines that were changed earlier by another MR as *dependent files*.

The dependency, as we have defined earlier, may have existed due to bug fixes (corrective), enhancements (perfective), changes from inspection, or new development (adaptive). 2530 files in the OA subsystem were found to have undergone dependent change. That is nearly 55 percent of all files in the subsystem and nearly 60 percent of all changed files. So, *in nearly 60 percent of cases, lines that are changed were changed again.* This kind of information can be very useful to the understanding of the maintenance phase of a software project. We had 51478 dependent change records and this data was the core of our analysis.

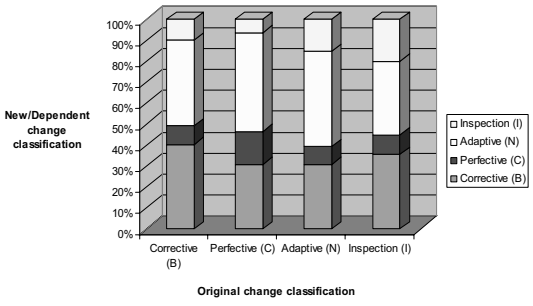


Figure 1: **Distribution of change classification on dependent files**

In Figure 1, we show the distribution of change classifications of the dependent files across the original files. The horizontal axis shows the types of changes made to the dependent files originally. In the vertical axis, we distribute the new changes based on their classification based on the implementation type. From the distribution it can be noted that most bug fixes were made to code that was already changed by an earlier MR to fix bugs. At this point of time, we can conclude that *roughly 40 percent of all changes made to fix errors introduced more errors*.

It is also interesting to note that nearly 40 percent of all the dependent changes were of the adaptive type and most perfective changes were made to lines that were previously changed for the same reason, i.e., enhancing performance or removing inefficiencies.

4. Analysis Summary and Next Steps

We have found that the probability that a one-line change would introduce at least one error is less than 4 percent. This result supports the typical risk strategy for one line changes and puts a bound on our search for catastrophic changes.

Interestingly, this result is very surprising considering the intial claim: "*one-line changes are erroneous 50 percent of the time*" [21]. This large deviation may be attributed to the structured programming practices and development and evolution processes involving code inspections and walkthroughs that were practiced for the development of the project under study. Earlier research [9] shows that without proper code inspection procedures in place, there is a very high possibility that one-line changes could result in error.

In summary, some of the more interesting observations that we made during our analysis include:

- Nearly 95 percent of all files in the software project were maintained at one time or another. If the common header and constants files are excluded from the project scope, we can conclude that nearly 100 percent of files were modified at some point in time after the initial release of the software product.
- Nearly 40 percent of the changes that were made to fix defects introduced one or more other defects in the software.
- Nearly 10 percent of changes involved changing only a single line of code; nearly 50 percent of all changes involved changing fewer than 10 lines of code; nearly 95% of all changes were those that changed fewer than 50 lines of code.
- Less than 4 percent of one-line changes result in error.
- The probability that the insertion of a single line might introduce a defect is 2 percent; there is nearly a 5 percent chance that a one-line modification will cause a defect. There is nearly a 50 percent chance of

at least one defect being introduced if more than 500 lines of code are changed.

- Less than 2.5 percent of one-line insertions were for perfective changes, compared to nearly 10 percent of insertions towards perfective changes when all change sizes were considered.
- The maximum number of changes was made for adaptive purposes, and most changes were made by inserting new lines of code.
- There is no credible evidence that deletions of fewer than 10 lines of code resulted in defects.

To fully understand these effects of small changes in particular, and changes in general, this study should be replicated across systems in different domains and of different sizes.

5. Acknowledgements

We wish to thank Harvey Siy, Bell Laboratories, Lucent Technologies, for sharing his knowledge of the 5ESS change management process. We would also like to thank Audrus Mockus, Avaya Research Labs, and Tom Ball, Microsoft Research, for their contributions and suggestions.

6. References

[1] Fred Brooks, "The Mythical Man-Month", Addison-Wesley, 1975

[2] Dieter Stoll, Marek Leszak, Thomas Heck, "Measuring Process and Product Characteristics of Software Components – a Case study"

[3] Audris Mockus, Lawrence G. Votta, "Identifying Reasons for Software Changes using Historic Databases", In International Conference on Software Maintenance, San Jose, California, October 14, 2000, Pages 120-130

[4] Todd L Graves, Audris Mockus, "Inferring Change Effort from Configuration Management Databases", Proceedings of the Fifth International Symposium on Software Metrics, IEEE, 1998, Pages 267-273

[5] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.S. Marron, Audris Mockus, "Does Code Decay? Assessing the Evidence from Change Management Data", IEEE Transactions on Software Engineering, Vol. 27, No. 1, January 2001

[6] Dewayne E. Perry, Harvey P. Siy, "Challenges in Evolving a Large Scale Software Product", Proceedings of the International Workshop on Principles of Software Evolution, 1998 International Software Engineering Conference, Kyoto, Japan, April 1998

[7] Audris Mockus, David M. Weiss, "Predicting Risk of Software Changes", Bell Labs Technical Journal, April-June 2000, Pages 169-180

[8] Rodney Rogers, "Deterring the High Cost of Software Defects", Technical paper, Upspring Software, Inc.

[9] G. M. Weinberg, "Kill That Code!", Infosystems, August 1983, Pages 48-49

[10] David M. Weiss, Victor R. Basili, "Evaluating Software Development by Analysis of Changes: Some Data from the Software Engineering Laboratory", IEEE Transactions on Software Engineering, Vol. SE-11, No. 2, February 1985, Pages 157-168

[11] Myron Lipow, "Prediction of Software Failures", The Journal of Systems and Software, 1979, Pages 71-75

[12] Swanson. E. B., "The Dimensions of Maintenance", Procedures of the Second International Conference on Software Engineering, San Francisco, California, October 1976, Pages 492-497

[13] Todd L. Graves, Alan F. Karr, J.S. Marron, Harvey Siy, "Predicting Fault Incidence Using Software Change History", IEEE Transactions on Software Engineering, Vol. 26, No. 7, July 2000, Pg 653-661

[14] H.E. Dunsmore, J.D. Gannon, "Analysis of the Effects of Programming Factors on Programming Effort", The Journal of Systems and Software, 1980, Pages 141-153

[15] Ie-Hong Lin, David A. Gustafson, "Classifying Software Maintenance", 1988 IEEE, Pages 241-247

[16] Dewayne E. Perry, Harvey P. Siy, Lawrence G. Votta, "Parallel Changes in Large Scale Software Development: An Observational Case Study", ACM Transactions on Software Engineering and Methodology 10:3 (July 2001), pp 308-337.

[17 Les Hatton, Programming Research Ltd, "Reexamining the Fault Density – Component Size Connection", IEEE Software, March/April 1997, Vol. 14, No. 2, Pages 89-97

[18] Victor R. Basili, Barry T. Perricone, "Software Errors and Complexity: An Empirical Investigation", Communications of the ACM, January 1984, Vol 27, Number 1, Pages 42-52

[19] Dewayne E. Perry and W. Michael Evangelist. ``An Empirical Study of Software Interface Errors", *Proceedings of the International Symposium on New Directions in Computing*, IEEE Computer Society, August 1985, Trondheim, Norway, pages 32-38

[20] Dewayne E. Perry and W. Michael Evangelist. ``An Empirical Study of Software Interface Faults --- An Update", *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, January 1987, Volume II, pages 113-126.

[21] Anecdotally related in an email conversation.

Process and Community Analysis

Data Mining for Software Process Discovery in Open Source Software Development Communities

Chris Jensen, Walt Scacchi
Institute for Software Research
University of California, Irvine
Irvine, CA, USA 92697-3425
{cjensen, wscacchi}@ics.uci.edu

Abstract

Software process discovery has historically been an intensive task, either done through exhaustive empirical studies or in an automated fashion using techniques such as logging and analysis of command shell operations. While empirical studies have been fruitful, data collection has proven to be tedious and time consuming. Existing automated approaches have expedited collection of fine-grained data, but do so at the cost of impinging on the developer's work environment, few of who may be observed. In this paper, we explore techniques for discovering development processes from publicly available open source software development repositories that exploit advances in artificial intelligence. Our goal is to facilitate process discovery in ways that are less cumbersome than empirical techniques and offer a more holistic, task-oriented view of the process than current automated systems provide.

1. Introduction and Beginnings

Software process models represent a networked sequence of activities, object transformations, and events that embody strategies for accomplishing software evolution [10]. Software process discovery seeks to take artifacts of development (e.g. source code, communication transcripts, and so forth), as its input and elicit the networked sequence of events characterizing the tasks that led to their development. This process model may then be used as input to other process engineering techniques such as redesign and re-engineering.

Open source software development (OSSD) communities are a rich opportunity for software process discovery and analysis with the benefit that so much of their process-relevant data is publicly available. Though many researchers have sought non-automated means of software process modeling, often there is so much information that it becomes intractable to subsume unaided, thus motivating the push for tools to assist in process discovery. In our past efforts [6], we have shown the feasibility of automating the discovery of software process models by using manual simulation of how such automated techniques might operate as a basis to substantiate that discovery and modeling of software development processes in large OSSD communities such as Mozilla, Apache, NetBeans, and Eclipse (consisting of tens of thousands of developers continuously contributing software artifacts to the community repository) is both plausible and amenable to automation. In this paper, we explore techniques for searching OSSD Web repositories for process data, relating these data in the form of process events, and assigning them to meaningful orders as a process model in an attempt to reduce the manual effort necessary to discover and model software processes.

We take, as our process meta-model, that of Noll and Scacchi [8]. Software processes are composed of events: relations of agents, tools, resources, and activities organized by control flow structures dictating that sets of events execute in serial, parallel, iteratively, or that one of the set is selectively performed.

It has been shown [6] that OSSD community Web repositories encode process data in terms of the structure of the community repository, its content, and its usage and update patterns. OSSD artifacts vary along these three dimensions over time, and this variance is the source of process events. To effectively discover a software process, we must be able capture these data and their changes. This may be done through combined application of text and link analysis techniques, as described below. We propose the use of text analysis techniques for extracting instances of process meta-model entities from the content of the community repositories, followed by link analysis to assert relationships between the mined entities in the form of process events. Next, we apply usage and update patterns to guide integration of the results of text and link analysis together in the form of a process model (see Figure 1). Finally, we conclude with addressing the knowable validity of discovered software process models and future directions for continuing work.

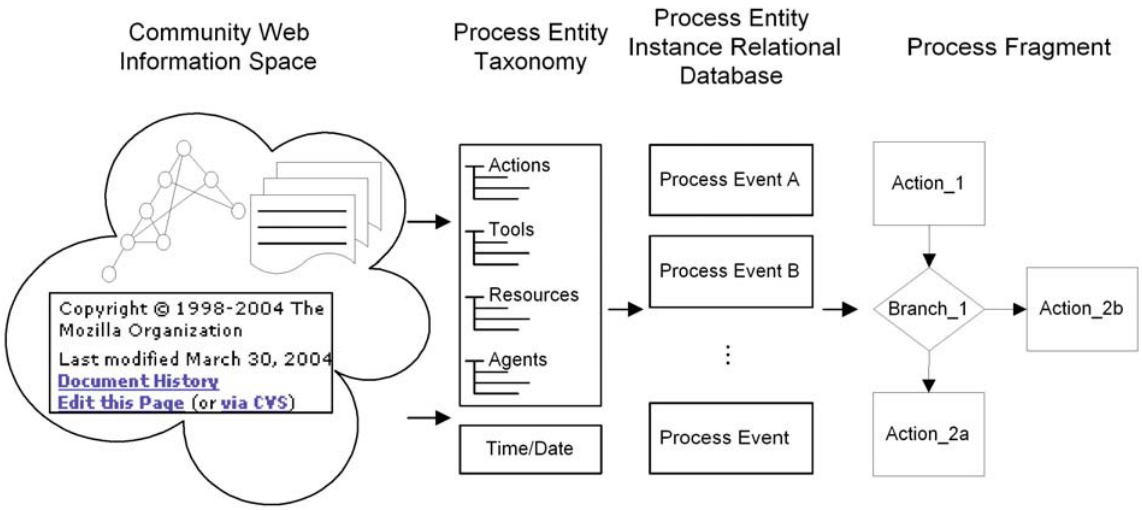


Figure 1: Web artifacts are filtered through a process entity taxonomy to extract atomic process action events, sequenced using temporal indications within the artifacts and reconstructed into a process using PRM

2. Text Analysis

The bulk of the process data is found within the content of Web artifacts. Much of the mapping consists of text extraction, matching between text strings in artifacts such as web pages and email messages and a taxonomy of process related keywords [5]. In the case of web content, we are especially looking for items like date stamps on email messages to place the associated events in time, document authors, and message recipients. This matching is done using a name recognizer.

An inherent challenge to name recognition is that many classes of lexical items we desire to recognize are open sets since we cannot enumerate all possible proper names they contain. Further, name classification suffers from synonymy and polysemy- the same concept represented using different terms, and different concepts represented using the same term, respectively. This frequently occurs between OSSD communities, using terms such as release *manager* rather than release *coordinator* to describe the same role. Fortunately, these are well known problems in text analysis and most text analysis systems provide some support for managing them. The SENSUS ontology system [3] is one such system that attempts to automate much of the domain modeling work allegedly covering most areas of human expertise. This automation is critical considering lexicographical differences across and evolution within communities.

Different types of content yield different opportunities for gathering data. Common to most open source communities are mailing lists and discussion forums, source repositories, community newsletters, issue repositories, and binary release sections, among others. The mere presence of these suggests certain activities in the development process. They also signal what types of data may be contained within. If we just look at source code repositories, we can derive a process specification of a limited set of activities- those that involve changes to the code. Similarly, issue and defect databases tell us that some testing is done on which the issue reports are based. In some communities, issue reports are also used to file feature requests. Such information may also be found within discussion forums or email lists.

Although it may seem tempting to attempt to tailor analysis of artifacts to their type (e.g. email message, defect report, etc) to capitalize on the structure of the artifact type thereby facilitating analysis. While this approach would potentially lead to increased performance in analysis of artifacts conforming to the structure expected by the artifact model, this structure varies widely between communities. To achieve high performance using artifact structure models requires development of models, not only for each artifact type in a community repository, but also for each artifact type used by all repositories under study.

It is interesting to note that we may uncover "how-to" guides or other partial process prescriptions in examining the community repository. Like all content, these may not accurately reflect the process as it is currently enacted, if they ever did. This

suggests the need for probabilistic methods for modeling software development processes to filter noise within a process instance and accounting for variance across instances.

By itself, the result of text extraction gives us the raw ingredients of a process model. We look to link analysis to put these ingredients together into atomic process events.

3. Link Analysis

Text extraction allows us to ask questions such as who is collaboration with whom. From this information, we can construct a social network [Madey, et al] for the community. Social networks may identify developers that frequently collaborate, but they do not tell us what the developers are doing, and, more importantly, how they are doing it. One way to associate what and how information is through the use of probabilistic relational modeling (PRM).

Probabilistic relational modeling [4] is somewhat inspired by entity relationship modeling used to describe databases. In the classical example, we might have tables of movie actors, movies, and roles actors have played in movies and want to learn relationships between them. Conceptually, this is no different from linking process agents playing a role to complete an action (using various tools that consume and produce resources). Probabilistic relational modeling allows inference about individual process entities while taking into account the relational structure between them, unlike traditional approaches that assume independence between entities. Why is this the right approach? Software processes driven by the choice of tools used in development. Tools either dictate what and when activities are performed, or tools are selected to support desired activities, and to an extent, suggest methods of completing activities (i.e. enforce process compliance). Developer roles emerge to perform these activities and carry out supplemental work not performed by development tools. Further, process entity instances arising from text analysis have other relationships. They are related contextually to other entities in the artifacts in which they are found. They are also related to artifacts hyperlinked to those in which they are present. Such contextual relationships arising from the logical structure of the repository are also good candidates for probabilistic relational modeling. Indeed, doing so allows us to form process events whose entities span multiple artifacts.

To learn relationships between process entities, we must know the context of the entity with respect to others. This context can be represented in two ways. Extracting the URL of the artifact in which each entity is located allows us to cross-reference that entity with others in the same artifact, as well as other artifacts in which that entity is located. Additionally, if we look at the creation date of the artifact in which it was located, we may be able to intuit that those instances that are temporally distant may signal an activity of lengthy duration multiple instances of the same activity. This determination, however, is the work of usage and update pattern analysis.

4. Usage and Update Patterns

Usage patterns, like content size, are indicators of which areas of the Web space are most active, which reinforces the validity of the data found therein and also claims of what activities in the process may be occurring at a given time. Web access logs, if available, provide a rich source of data. Web page hit counters and last update statistics are also useful for this purpose.

Cadez [1] and Hong, et al [2] demonstrate two techniques for capturing Web navigation patterns, however neither can be done in a strictly noninvasive manner. The first uses server logs and cannot provide tours of the repository and the latter requires members to access the community Web through a proxy server used to track tours. Nevertheless, if we can map tours of the community Web to process events, we can get a sense of which activities are dependent on which other activities, which can be done in parallel, which sequences are done iteratively.

Fortunately, most large OSSD communities use content managing tools to perform versioning of not only product source code, but of other artifacts in the repository, as well. By analyzing changelogs we can learn the frequency of Web updates, in addition to the agent performing the update, and to some extent, the tools used to create the artifact, given its type. Work by Ripoche and Gasser [9] does this to an extent, studying defect resolution status in open source defect repositories. The approach may be generalized, extended with using the text and link analysis techniques given above, and applied to other types of artifacts, though with somewhat less precision due to the inferential nature of process entity relationship construction.

Unfortunately, revision histories are not always available. Since OSSD repositories are publicly accessible, it is possible to spider the Web repository periodically to track changes externally via *diff* tools, though information regarding the precise time of

update and author would be lost. As an ethical matter, periodic spidering increases the load on the server that, for large repositories, is potentially burdensome.

By examining usage and update patterns, it is possible for us to detect process control flow structures. If we merely order then by time, the set of process events discovered is sequential. Iterations can be teased out of the sequence by considering patterns of repeated tours and updates of and to the Web. Activities being performed in parallel may also be discerned by examining non-intersecting concurrent usage and update patterns. Further, by analyzing the variance between iterations of the same task, we can identify sets of alternate activities, if the variance is small.

5. Process Model Verification

A critical question of software process discovery, regardless of automation, is how we may discern if the process discovered is a correct reflection of the process enacted by the community. The likelihood of arriving at an accurate model increases with the amount of data examined, within the limitations of the techniques applied. This is because the confidence of an asserted relationship between process entities increases with more positive instances of those relationships. Likewise, weak relationships are rejected due to insufficient evidence. At the same time, relationships between entities cannot be discovered if the entities are not in the list of process-relevant terms we look for during text extraction. Thus, the process model obtained is only as good as the taxonomy.

6. Conclusion

In this paper, we have presented a novel approach to discovering software processes from OSSD Web repositories, combining techniques for text analysis, link analysis, and of repository usage and update patterns. Though we have focused our discussion on open source repositories, given the availability of the artifacts, we believe that these techniques can be applied to closed source software repositories, and given the appropriate domain information, other types of processes, as well. Our hope is that in doing so, we may increase understanding of the process techniques that have led to their success.

7. Acknowledgments

The research described in this report is supported by grants from the National Science Foundation #ITR-0083075 and #ITR-0205679 and #ITR-0205724. No endorsement implied. Contributors to work described in this paper include Mark Ackerman at the University of Michigan Ann Arbor; Les Gasser at the University of Illinois, Urbana-Champaign; John Noll at Santa Clara University; and Margaret Elliott at the UCI Institute for Software Research.

8. References

[1] Cadez, I.V., Heckerman, D., Meek, C., Smyth, P., and White, S. Visualization of Navigation Patterns on a Web Site Using Model Based Clustering. In Proc. 2000 Knowledge Discovery and Data Mining Conference, 280-284. (2000).

[2] Hong, J. Heer, S. Waterson, and J. Landay, WebQuilt: A proxy-based approach to remote web usability testing, ACM Transactions on Information Systems, 19(3), 263-285. (2001).

[3] Hovy, E.H., A. Philpot, J.-L. Ambite, Y. Arens, J.L. Klavans, W. Bourne, and D. Saroz. 2001. Data Acquisition and Integration in the DGRC's Energy Data Collection Project. In *Proceedings of the dg.o 2001 Conference*. Los Angeles, CA.

[4] Getoor, L., Friedman, N., Koller, D., Taskar B. Learning Probabilistic Models of Link Structure, Journal of Machine Learning Research, 2002.

[5] Jensen, C. Applying a Reference Framework to Open Source Software Process Discovery. In Proceedings of the First Workshop on Open Source in an Industrial Context OOPSLA-OSIC03, Anaheim, CA October 2003.

[6] Jensen, C. and Scacchi W. Simulating an Automated Approach to Discovery and Modeling of Open Source Software Development Processes. In Proceedings of ProSim'03 Workshop on Software Process Simulation and Modeling, Portland, OR May 2003.

[7] Madey, G., Freeh, V., and Tynan, R. "Modeling the F/OSS Community: A Quantitative Investigation," in *Free/Open Source Software Development*, ed., Stephan Koch, Idea Publishing, forthcoming.

[8] Noll, J. and Scacchi, W. Specifying Process Oriented Hypertext for Organizational Computing. *Journal of Network and Computer Applications* 24, (2001). 39-61.

[9] Ripoche, G. and Gasser, L. "Scalable Automatic Extraction of Process Models for Understanding F/OSS Bug Repair", submitted to the 2003 International Conference on Software & Systems Engineering and their Applications (ICSSEA'03), CNAM, Paris, France, December 2003.

[10] Scacchi, W. Process Models in Software Engineering, in J. J. Marciniak (ed.), Encyclopedia of Software Engineering, 2nd. Edition, 2002.

Applying Social Network Analysis to the Information in CVS Repositories

Luis Lopez-Fernandez, Gregorio Robles, Jesus M. Gonzalez-Barahona
GSyC, Universidad Rey Juan Carlos
{llopez,grex,jgb}@gsyc.escet.urjc.es

Abstract

The huge quantities of data available in the CVS repositories of large, long-lived libre (free, open source) software projects, and the many interrelationships among those data offer opportunities for extracting large amounts of valuable information about their structure, evolution and internal processes. Unfortunately, the sheer volume of that information renders it almost unusable without applying methodologies which highlight the relevant information for a given aspect of the project. In this paper, we propose the use of a well known set of methodologies (social network analysis) for characterizing libre software projects, their evolution over time and their internal structure. In addition, we show how we have applied such methodologies to real cases, and extract some preliminary conclusions from that experience.

Keywords: *source code repositories, visualization techniques, complex networks, libre software engineering*

1 Introduction

The study and characterization of complex systems is an active research area, with many interesting open problems. Special attention has been paid recently to techniques based on network analysis, thanks to their power to capture some important characteristics and relationships. Network characterization is widely used in many scientific and technological disciplines, ranging from neurobiology [14] to computer networks [1] [3] or linguistics [9] (to mention just some examples). In this paper we apply this kind of analysis to software projects, using as a base the data available in their source code versioning repository (usually CVS). Fortunately, most large (both in code size and number of developers) libre (free, open source) software projects maintain such repositories, and grant public access to them.

The information in the CVS repositories of libre software projects has been gathered and analyzed using several methodologies [12] [5], but still many other approaches are possible. Among them, we explore here how to apply some techniques already common in the traditional (social) network analysis. The proposed approach is based on considering either modules (usually CVS directories) or developers (commiters to the CVS) as vertices, and the number of common commits as the weight of the link between any two vertices (see section 3 for a more detailed definition). This way, we end up with a weighted graph which captures some relationships between developers or modules, in which characteristics as information flow or communities can be studied.

There have been some other works analyzing social networks in the libre software world. [7] hypothesizes that the organization of libre software projects can be modeled as self-organizing social networks and shows that this seems to be true at least when studying SourceForge projects. [6] proposes also a sort of network analysis for libre software projects, but considering source dependencies between modules. Our approach explores how to apply those network analysis techniques in a more comprehensive and complete way. To expose it, we will start by introducing some basic concepts of social network analysis which are used later (section 2), and the definition of the networks we consider 3. In section 4 we introduce the characterization we propose for those networks, and later, in section 5, we show some examples of the application of that characterization to Apache, GNOME and KDE. To finish, we offer some conclusions and discuss some future work.

2 Basic concepts on Social Network Analysis

The Theory of Complex Networks is based on representing complex systems as graphs. There are many examples in the literature where this approach has been successfully used in very different scientific and technological disciplines, identifying vertices and links as relevant for each specific domain. For example, in ecological networks each vertex may represent a particular specie, with a link between two species if one of them "eats" the other. When dealing with social networks, we may identify vertices with persons or groups of people, considering a link when there is some kind of relationship between them.

Among the different kinds of networks that can be con-

sidered, in this paper, we use affiliation networks. In affiliation networks there are two types of vertices: *actors* and *groups*. When we represent the network in terms of actors, each vertex is associated with a particular person and two vertices are linked together when they belong to the same group of people. When we represent the network in terms of groups, each vertex is associated with a group and two groups are linked through an edge when there is, at least, one person belonging to both at the same time.

Social networks can be directed (when the relationship between any two vertices is one way, like "is a boss of") or undirected (when it is bidirectional, like "live together"). In addition, they can be weighted (each edge has an associated numeric value) or unweighted (each edge exists or not).

3 Definition of the networks of developers and modules

In the approach we propose, for each project we build two networks using the commit information of the CVS system. Both correspond to the two sides of an affiliation network obtained when we consider commiters and modules in libre software projects. In both cases we consider weighted undirected networks as follows:

- **Commiter network**. Each vertex corresponds to a particular commiter (usually, a developer of the project). Two commiters are linked when they have contributed to at least one common module, being the weight of the corresponding edge the number of commits performed by both developers to all common modules.
- **Module network**. Vertices represent a software module of the project. Two modules are linked when there is at least one commiter who has contributed to both of them. Edges are weighted by the total number of commits performed by common commiters to both modules.

The definition of what is a module will be different from project to project, but usually will correspond to top level directories in the CVS repository. In the case of both networks, the weight of each edge (*degree of relationship*) reflects the *closeness* of two vertices. The higher it is, the stronger the relationship between the given two vertices. We may also define the *cost of relationship* between any two vertices as the inverse of the *degree of relationship*. That *cost of relationship* is a measure of the "distance" between them, in the sense that the higher this parameter the more difficult to reach one vertex from the other. For this reason we use the *cost of relationship* as the base for defining a distance in our networks. Given a pair of vertices i and j, we define the distance between them as $d_{ij} = \sum_{e \in P_{i,j}} c_e$, where P_i, j is the set of all the edges in the shortest path from i to j, and c_e is the *cost of relationship* of edge e of such path.

4 Characterization of the networks considered for each project

For our analysis, we have considered a number of parameters characterizing the topology of the networks. In particular, we use the following definitions (which are common in the analysis of affiliation networks):

- **Degree of a vertex** (k): number of edges connected to that vertex. In the case of commiter networks, for each commiter it represents the number of *companion* commiters, contributing to the same modules as the given one. In the case of module networks, it is the total number of modules with which the given one shares commiters.
- **Weighted degree of a vertex**: sum of the weights of all edges connected to that particular vertex. This can be interpreted as the degree of relationship of a given vertex with its direct neighborhood.
- **Distance centrality of a vertex** [13] (D_c): proximity to the rest of vertices in the network. It is also called *closeness centrality*: the higher its value, the closer that vertex is to the others (on average). Given a vertex v and a graph G, it can be defined as:

$$D_c(v) = \frac{1}{\sum_{t \in G} d_G(v,t)}, \tag{1}$$

  where $d_G(v,t)$ is the minimum distance from vertex v to vertex t (the sum of the costs of relationship of all edges in the shortest path from v to t). The distance centrality can be interpreted as a measurement of the influence of a vertex in a graph: the higher its value, the easiest it is for that vertex to spread information into that network. Let's observe that when a given vertex is "far" from the others, it has a low degree of relationship (i.e. a high cost of relationship) with the rest. In that case the term $\sum_{t \in G} d_G(v,t)$ will be high, meaning that the vertex is not placed in a central position in the network, being its distance centrality low. This parameter can be used to identify modules or commiters which are *well related* in a project.
- **Betweenness centrality of a vertex** [4, 2]: The betweenness centrality of a vertex B_c is a measurement of the number of shortest paths traversing that particular vertex. Given a vertex v and a graph G, it can be defined as:

$$B_c(v) = \sum_{s \neq v \neq t / in G} \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{2}$$

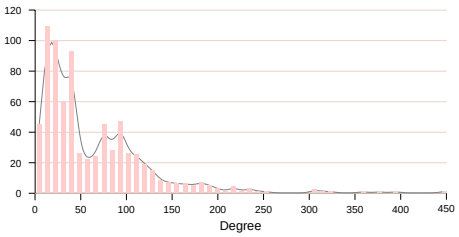


Figure 1. Distribution of the degrees of commiters in Apache, circa February 2004

where $\sigma_{st}(v)$ is the number of shortest paths from s to t going through v, and σ_{st} is the total number of shortest paths between s and t. The betweenness centrality of a vertex can be interpreted as a measurement of the importance of a vertex in a given graph, in the sense that vertices with a high value of this parameter are intermediate nodes for the communication of the rest. In the case of weighted networks, multiple shortest paths between any pair of vertices are highly improbable. So, the term $\frac{\sigma_{st}(v)}{\sigma_{st}}$ takes usually only two values: 1, if the shortest path between s and t goes through v, or 0 otherwise. Therefore, the betweenness centrality is just a measurement of the number of shortest paths traversing a given vertex.

- **Clustering coefficient of a vertex** [14]: The clustering coefficient c of a vertex measures the connectivity of its direct neighborhood. Given a vertex v in a graph G, it can be defined as the probability that any two neighbors of v be connected. Hence

$$c(v) = \frac{E(v)}{k_v(k_v-1)}, \tag{3}$$

where k_v is the number of neighbors of v and $E(v)$ is the number of edges between those neighbors. A high clustering coefficient in a network indicates that this network has a tendency to form cliques. Observe that the clustering coefficient does not consider the weight of edges.

- **Weighted clustering coefficient of a vertex** [10]: The weighted clustering coefficient c_w of a vertex is an attempt to generalize the concept of clustering coefficient to weighted networks. Given a vertex v in a weighted graph G it can be defined as:

$$c_w(v) = \sum_{i \neq j \in N_G(v)} w_{ij} \frac{1}{k_v(k_v-1)}, \tag{4}$$

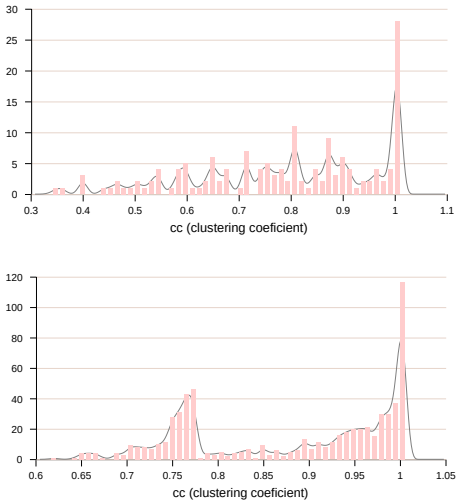


Figure 2. Clustering coefficient of modules in Apache (top) and GNOME (bottom), circa February 2004 (distribution)

where $N_G(v)$ is the neighborhood of v in G (the subgraph of all vertices connected to v), w_{ij} is the degree of relationship of the link between neighbor i and neighbor j ($w_{ij} = 0$ if there are no link), and k_v is the number of neighbors. The weighted clustering coefficient can be interpreted as a measurement of the local efficiency of the network around a particular vertex. For our networks, remark that the term $\sum_{i \neq j \in N_G(v)} w_{ij}$ can be seen as the total *degree of relationship* in the neighborhood of vertex v, while $\frac{1}{k_v(k_v-1)}$ is the total number of relationships that could exists in that neighborhood.

5 Case studies: Apache, GNOME and KDE modules

Apache, GNOME and KDE are all well known libre software projects, large in size (each well above the million lines of code), in which several subprojects (modules) can be identified. They have already been studied (for instance in [11] and [8]) from several points of view. We have used them to apply our methodology, and in this section some results of that application are shown (just an example of how a project can be characterized from several points of view).

In figure 1 the distribution of the degree of relationship for each commiter in the Apache project is shown as an ex-

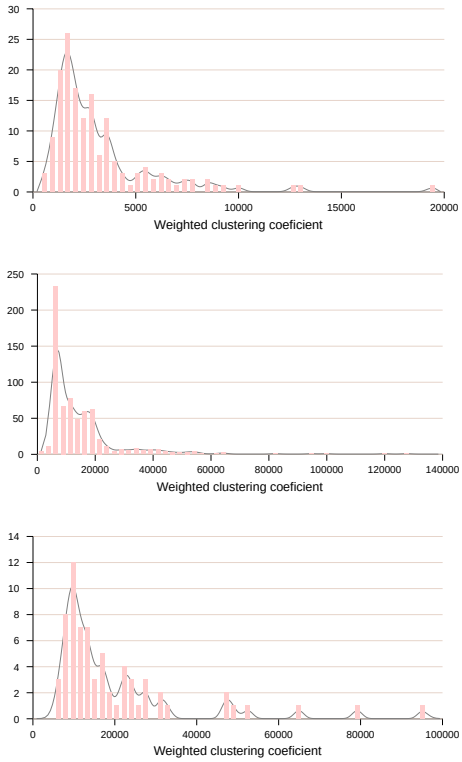


Figure 3. Weighted clustering coefficient of modules in Apache (top), GNOME (middle), and KDE (bottom), circa February 2004 (distribution)

ample of how developers can be characterized by how they relate to each other. It is easy to appreciate how that distributions shows two peaks, one between 20-40 and other around 70-90. Only a handful of developers has direct relationship with more than 200 companions.

In figure 2 the distribution of the clustering coefficient of modules in Apache and GNOME is compared. Although in both cases there is a peak in 1 (meaning that in many cases the direct neighborhood of a module is completely linked together), there is an interesting peak in GNOME around 0.77, which should be studied but probably corresponds to a sparse-connected cluster.

Figure 3 shows how, despite differences in the distribution of the clustering coefficient, the distribution of the weighted clustering coefficient has more similar shapes, with a quick rise from zero to a maximum, and a slower, asymptotic decline later. This would mean than in the three projects most nodes (those near the peak) are in clusters with a similar interconnection structure.

As a final example, on the evolution of a project, figure 4 shows the distribution of the connection degree of four snapshots of the Apache project. It can be seen how there is a tremendous growth in the connection degree of the most connected module (from 34 in 2001 to more than 100 in 2004), while the shape of the distribution changes over time: from 2001 to 2002 a two-peak structure develops, which slowly changes into a one-peak distribution through 2003 and 2004.

For lack of space we do not offer it here, but the analysis of the top modules and developers for each parameter considered gives a lot of insight on which ones are helping to maintain the projects together, to deal with information flows, or are the aggregators of clusters.

6 Conclusions and further work

In this paper we have shown a methodology which applies affiliation network analysis to data gathered from CVS repositories. We also offer some examples of how it can be applied to characterize libre software projects. From a more general point of view, we have learned (demonstration not shown in this paper) that in the three analyzed cases (Apache, GNOME and KDE), both the commiters and the modules networks are small-world networks, which means that all the theory developed for them applies here.

Our group is still starting to explore the many paths open by this methodology. Currently, we are interested in analyzing a large number of projects, looking for correlations which can help us to make estimations and predictions of the future evolution of projects. We are also looking for characterizations of projects based on the parameters of the curves that interpolate the distributions of the parameters we are studying. And of course, applying other techniques

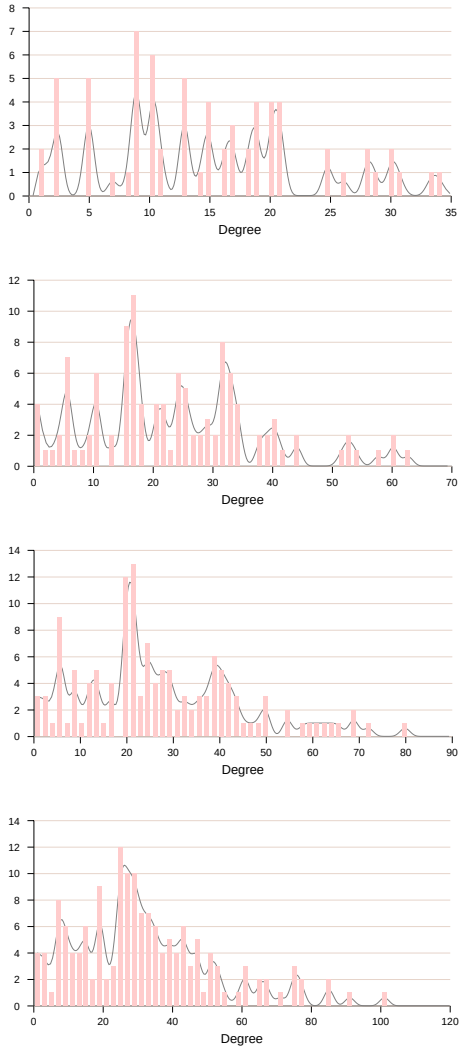


Figure 4. Connection degree of modules in Apache circa February from 2001 (top) to 2004 (bottom) (distribution)

usual in small-world and other social networks.

We feel that these research paths will allow for the more complete understanding of how libre software projects differentiate from each other, and also will help to identify common patterns and invariants.

References

[1] R. Albert, A. L. Barabsi, H. Jeong, and G. Bianconi. Power-law distribution of the world wide web. *Science*, 287, 2000.

[2] J. Anthonisse. The rush in a directed graph. Technical report, Stichting Mathemastisch Centrum, Amsterdam, The Netherlands, 1971.

[3] Cancho and R. Sole. The small world of human language. *Proceedings of the Royal Society of London. Series B, Biological Sciences*, 268:2261–2265, Nov. 2001.

[4] C. Freeman. A set of measures of centrality based on betweenness. *Sociometry 40, 35-41*, 1977.

[5] D. Germn and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, Portland, Oregon, 2003.

[6] R. A. Ghosh. Clustering and dependencies in free/open source software development: Methodology and tools. *First Monday*, 2003. `http://www.firstmonday.dk/issues/issue8_4/ghosh/index.html`.

[7] V. F. Greg Madey and R. Tynan. The open source development phenomenon: An analysis based on social network theory. In *Americas Conference on Information Systems (AMCIS2002)*, pages 1806–1813, Dallas, TX, USA, 2002. `http://www.nd.edu/~oss/Papers/amcis_oss.pdf`.

[8] S. Koch and G. Schneider. Effort, cooperation and coordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27–42, 2002.

[9] R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins. The web and social networks. *IEEE Computer*, 35(11):32–36, 2002.

[10] V. Latora and M. Marchiori. Economic small-world behavior in weighted networks. *Euro Physics Journal B 32, 249-263*, 2003.

[11] A. Mockus, R. Fielding, and J. Herbsleb. A case study of open source software development: The Apache server. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pages 263–272, Limerick, Ireland, 2000.

[12] G. Robles-Martinez, J. M. Gonzalez-Barahona, J. Centeno-Gonzalez, V. Matellan-Olivera, and L. Rodero-Merino. Studying the evolution of libre software projects using publicly available data. In *Proceedings of the 3rd Workshop on Open Source Software Engineering, 25th International Conference on Software Engineering*, pages 111–115, Portland, Oregon, 2003.

[13] G. Sabidussi. The centrality index of a graph. *Psychometirka 31, 581-606*, 1996.

[14] D. Watts and S. Strogatz. Collective dynamics of small-world networks. *Nature 393, 440-442*, 1998.

Mining a Software Developer's Local Interaction History

Kevin A. Schneider, Carl Gutwin, Reagan Penner and David Paquette
Department of Computer Science, University of Saskatchewan
57 Campus Drive, Saskatoon, SK S7N 5A9 Canada
{kas, gutwin, rpenner}@cs.usask.ca, dnp972@mail.usask.ca

Abstract

Although shared software repositories are commonly used during software development, it is typical that a software developer browses and edits a local snapshot of the software under development. Developers periodically check their changes into the software repository; however, their interaction with the local copy is not recorded. Local interaction histories are a valuable source of information and should be considered when mining software repositories.

In this paper we discuss the benefits of analyzing local interaction histories and present a technique and prototype implementation for their capture and analysis. As well, we discuss the implications of local interaction histories and the infrastructure of software repositories.

1. Introduction

We are interested in mining local interaction histories of a software development team to help coordinate their activities and to coordinate the change and use of project artifacts.

A software developer's interaction with a software repository includes editing source code but also involves actions to browse or locate source code. We are interested in recording and analysing this interaction, which we refer to as the developer's *local interaction history*. Our principle motivation is to use this information to support awareness in team based software development.

Developers normally change a local copy of the software under development. Periodically, the developer will synchronize their changes with the shared software repository. Although a portion of the developers' interaction with the local software artifacts may be recorded for the purpose of undoing changes and for recovering from previously saved versions, the interaction is not recorded in the shared repository and is incomplete when considering awareness support.

In our approach, as a developer changes software artifacts the different versions are recorded in a shared 'shadow' repository and analysed with respect to the structure of the software. Hierarchical containment of language entities (the structure of the software) is modeled separately so that we can track changes across the language entities. For example, we can track changes to a *method* across *classes* and *packages*. We use this strategy to monitor API (application programming interface) change and usage.

Mining local interaction histories has a number of potential applications, including:

- **Coordinating team member activities.** Monitoring changes to an API and monitoring API usage may be useful in supporting team awareness during software development. (The focus of this paper and our current prototype implementation.)
- **Identifying refactoring patterns.** Analysing local interaction histories may be useful for identifying novel refactoring patterns and coordinating refactorings that affect other team members.
- **Coordinating multiple file undos.** Tracking changes with respect to the structure of a software system may provide software development guidance when undoing a set of changes.
- **Identifying browsing patterns.** Local interaction history includes the developer's searching, browsing and file access activities. Analysing this browsing interaction may be useful in supporting a developer locate technical expertise or exemplars.
- **Project Management.** Recording the changes a developer makes to software with respect to communication logs or project plans may prove to be fruitful for organizing and managing a software project.

The next section discusses background and related work, focusing on coordination and communication issues in software development. Subsequent sections describe our approach and prototype. The implications of mining local interaction histories and the infrastructure of software repositories is discussed with our future research directions in the paper's conclusion.

2. Background and Related Work

Collaborative software development presents difficult coordination and communication problems, particularly when teams are geographically distributed [6, 8, 10, 12, 13]. Even though projects can be organized to make individual developers partly independent of one another, dependencies cannot be totally removed [10]. As a result, there are often situations where team members duplicate work, overwrite changes, make incorrect assumptions about another person's intentions, or write code that adversely affects another part of the project.

These problems often occur because of a lack of awareness about what is happening in other parts of the project. Unfortunately, current development tools and environments do not make it easy to maintain awareness of others' activities [1]. Awareness is a design concept that holds promise for significantly improving the usability of collaborative software development tools.

2.1. Collaboration in Software Development

Collaboration support has always been a part of distributed development – teams have long used version control, email, chat groups, reviews, and internal documentation to coordinate activities and give and gather information – but these solutions generally either represent the project at a very coarse granularity (e.g. CVS [3]), require considerable time and effort (e.g. reading documentation), or depend on people's current availability (e.g. IRC).

Researchers in software engineering and CSCW have found a number of problems that still occur in group projects and distributed software development. They found that it is difficult to: determine when two people are making changes to the same artifacts [10]; communicate with others across timezones and work schedules [6]; find partners for closer collaboration or assistance on particular issues [12]; determine who has expertise or knowledge about the different parts of the project [13]; benefit from the opportunistic and unplanned contact that occurs when developers are colocated [8]. As Herbsleb and Grinter [8] state, lack of awareness – "the inability to share at the same environment and to see what is happening at the other site" (p. 67) is one of the major factors in these problems.

2.2. Group Awareness

In any group work situation, awareness of others provides information that is critical for smooth and effective collaboration. This is *group awareness*: the understanding of who is working with you, what they are doing, and how your own actions interact with theirs [7]. Group awareness is useful for many of the activities of collaboration – for coordinating actions, managing coupling, discussing tasks, anticipating others' actions, and finding help.

In a software project, knowledge of others' activities, both past and present, has obvious value for project management, but developers also use the information for many other purposes that assist the overall cohesion and effectiveness of the team. For example, knowing the specific files and objects that another person has been working on can give a good indication of their higher-level tasks and intentions; knowing who has worked most often or most recently on a particular file indicates who to talk to before starting further changes; and knowing who is currently active can provide opportunities for real-time assistance and collaboration.

On software projects, awareness information is currently difficult to obtain from development environments: although some of the facts exist (e.g. from CVS logs) there are currently no low-effort means for gathering them. A few research systems do show awareness information (particularly TUKAN [12] and Plantír [11]), but little support exists in more widespread environments.

3. Project Watcher

ProjectWatcher is a prototype system that gathers information about project artifacts and developer's actions with those artifacts, and that visualizes this awareness information in the Eclipse [5] development environment (Figure 1). ProjectWatcher consists of two main parts – the mining component and the visualization plugins.

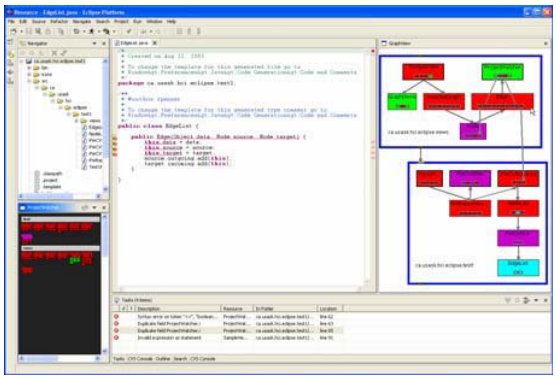


Figure 1. ProjectWatcher in Eclipse. Visualizations are at lower left and upper right.

The mining component analyzes the source code of a project to produce facts for use by the ProjectWatcher visualization plugin. The mining component gathers information on the structure of the project and also on the current and historical activity of the project team members.

To be able to gather developer activity information, a shadow CVS repository of the project is maintained (Figure 2). User edits are auto-committed to the shadow repository as developers edit source code files. Although Eclipse provides a local history of changes, we require that the changes be available to other developers in the software development team and so publishing them in the shadow repository gives us that facility. As well, we are able to record actions along with changes to software artifacts, and we are able to commit changes at different time intervals.

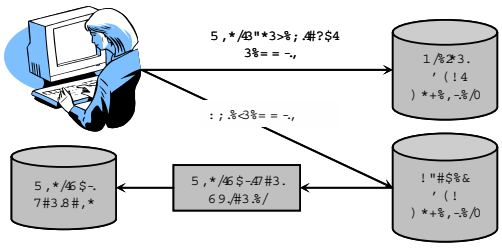


Figure 2. Capturing User Edits. A shadow software repository is used to record the activities of a software developer.

The user edits mining component analyzes the shadow CVS repository to obtain facts about who has been editing the class methods and when. A version of a file is created each time it is auto-committed to the shadow repository. The mining component analyses the differences between versions to track API usage and API change.

The mining component is implemented in two stages and may either be run on the shadow software repository or on the shared software repository (Figure 3). Stage one uniquely names all entities in the project while extracting the entity and relationship facts. This process is accomplished with a TXL program using syntactic pattern matching [2, 4]. At this point, the method call facts are not uniquely identified since we do not have sufficient information to identify which package or class the method being called belongs to. This resolution is accomplished by stage two, the method call resolver.

The method call resolver extracts facts from the project source code and integrates them with the facts extracted from stage one. Next, the method call facts are analyzed to determine which package and class the method that was called belongs to. This process involves resolving the types of variables and return types of methods that are passed as arguments to method calls. The types of all the arguments are identified, and then scope, package, class, and method facts are analyzed to determine which package and class the method belongs to. To resolve calls to the Java library, the full Java API is first processed by the ProjectWatcher min-

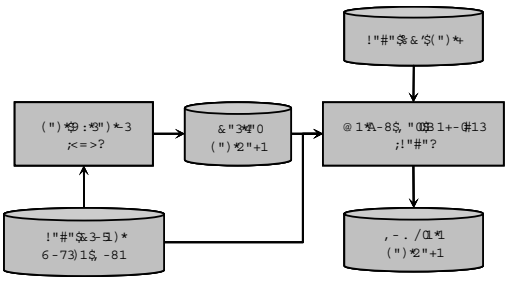


Figure 3. Mining User Edits. In a two stage process, package, class and method facts are extracted and combined with Java API facts. The facts are used by the visualization component to convey API use and API change information.

ing component (this is only done once for all projects). Not all calls may be resolved, however for our purpose the accuracy of the method call resolver is adequate.

The complete factbase contains uniquely identified facts indicating all packages, classes, methods, variables, and relationships for a Java project and all user edits. These facts are used by the visualization plugin to show activity and proximity information. The time and space required for fact extraction and factbase storage depends on the size of the code. For example, ProjectWatcher has been tailored for Java, and mining the Java Development Kit 1.4.1 results in 202 package facts, 5,530 class facts, 47,962 method facts, and 106,926 call facts.

4. Awareness Visualization

4.1. Activity Awareness

ProjectWatcher visualizes team members' past and current activities on project artifacts. The visualization uses the ideas of interaction history [9] and overviews: the interaction history is a record all of the actions that a person undertakes with a project artifact (gathered unobtrusively by the mining component as people carry out their normal tasks); the overview representation is a compact display of all the project artifacts, that can be overlaid with visual information about the interaction history. Although some tools such as CVS front-ends do have limited visualization (e.g. by colour on the project tree), our goal here is to collect much more information about interaction, and provide much richer visualizations that will allow team members to gather more detailed awareness information.

ProjectWatcher plugins use the extracted fact base to create a visual model of what each developer is doing in that

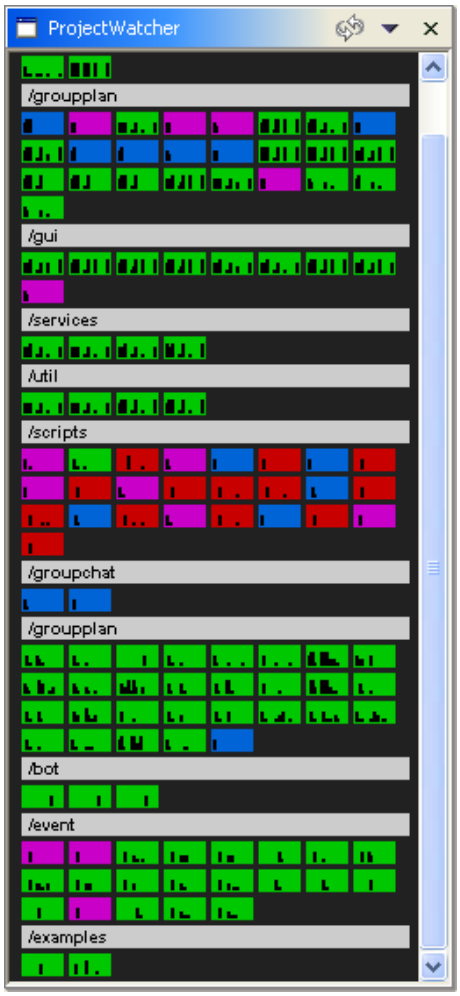


Figure 4. Project overview plugin showing packages (grey bars) and classes within each package (coloured blocks). Colour indicates who edited the class most recently. Black marks inside class blocks chart edits since project start.

project space. In the overview plugin (Figure 4), project artifacts are shown in a simple stacked fashion that displays packages, files, classes, and methods. Artifacts are always stacked by creation date, so that their location in the overview can over time be learned by the user. On this basic (but space-saving) representation, we overlay awareness information. First, each developer is assigned a unique colour, and this colour can be added to the blocks in the overview based on a set of filters. Common filters include who has modified artifacts most recently, or modified them most often. Second, we show a summary of the activity history for each artifact with a small bar graph drawn inside the object's rectangle; bars represent amount of change to the class since its creation. Finally, more information about an artifact can be obtained by holding the cursor over a rectangle: for example, the name of the class and a more detailed bar graph, along with details about the state of the class compared to the CVS repository.

4.2. Proximity Awareness

Following on from a basic understanding of others' activities is the question of proximity – that is "who is working near to me?" in terms of the structures and dependencies of the software system under development.

The notion of distance to another person has not been studied extensively, although it has been explored previously in Schümmer's TUKAN [12]. We have developed a visualization tool (Figure 5) that makes it easier to see proximity-based groups. Once actions are mapped to the dependency structure, the graph is presented in visual form with people's locations and proximities made explicit.

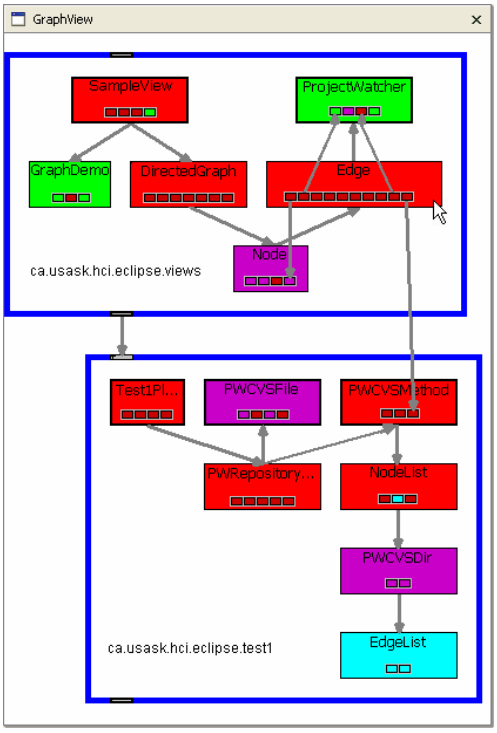


Figure 5. ProjectWatcher graph view

5. Conclusion

We have presented a system for mining local interaction histories to help address some of the awareness problems experienced in distributed software development projects. The system observes a software developer's activities in a software development environment and records those actions in relation to the artifact-based dependencies extracted from source code. Visualization plugins represent this information for developers to see and interact with. Although our prototypes have limitations (particularly in terms of project size), they can provide developers with much-needed information about who is working on the project, what they are doing, and how closely linked two developers are.

Our experience suggests a number of directions for mining software repository research, including:

- **Content.** Research on awareness often monitors a software development teams' interaction with a shared software repository. Unfortunately, the granularity of check-in and check-out is usually too coarse to adequately monitor change. This suggests that the content of shared software repositories should also include local interaction histories.

- **Rapid incremental processing.** For our purposes it is important that the computation of source facts and their resolution be relatively efficient to support interactive visualizations.

- **Robustness.** Our analysis may process source that is currently being edited and so the source may not be well-formed. We require that fact extraction and resolution needs to support analysis under ongoing change.

Our future plans with the system involve both improvements and new directions. With the current system, we plan to continue refining our representations and filters to determine how the information can be best presented to developers. We currently visualize source code that is in the process of being edited, and therefore the source code may be inconsistent, incomplete and frequently updated. We are investigating techniques for improving the robustness and performance of the mining component and visualizing partial information given these circumstances.

Longer range plans involve extensions to the basic ideas of project artifacts and interaction histories. We plan to extend our artifact collection to include entities other than those in source code. Many other project artifacts exist, including communication logs, bug reports and task lists. We hope to establish additional facts to model these artifacts and to use the new artifacts and their relationships in the awareness visualizations.

We can also extend our use of the interaction histories to other areas. For example, recording developers' interaction history and extracting method call facts from the source code provides us with basic API usage information. We can present this information in a future plugin to provide awareness of technology expertise. A developer wishing to know how to use a particular Java API feature may be presented with a list of developers who have used the feature frequently or recently. Alternatively, the visualization plugin may present this information overlaid on the project's dependency structure.

Acknowlegment

The authors would like to thank IBM Corporation for supporting this research.

References

[1] M. C. Chu-Carroll and S. Sprenkle. Coven: brewing better collaboration through software configuration management. In *Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 88–97. ACM Press, 2000.

[2] J. R. Cordy, T. R. Dean, A. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Journal of Information and Software Technology*, 44(13):827–837, October 2002.

[3] CVS. Concurrent Versions System. Available online at http://www.cvshome.org/.

[4] T. R. Dean, J. R. Cordy, K. A. Schneider, and A. Malton. Using design recovery techniques to transform legacy systems. In *ICSM*, pages 622–631, 2001.

[5] Eclipse. Available online at http://www.eclipse.org/.

[6] R. E. Grinter, J. D. Herbsleb, and D. E. Perry. The geography of coordination: dealing with distance in r&d work. In *Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 306–315, 1999.

[7] C. Gutwin and S. Greenberg. A descriptive framework of workspace awareness for real–time groupware. *Computer Supported Cooperative Work*, 11(3):411–446, 2002.

[8] J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, pages 63–70, 1999.

[9] W. C. Hill, J. D. Hollan, D. Wroblewski, and T. McCandless. Edit wear and read wear. In *Proceedings of CHI'92*, pages 3–9. ACM Press, 1992.

[10] R. E. Kraut and L. A. Streeter. Coordination in software development. *Communication of the ACM*, 38(3):69–81, 1995.

[11] A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: raising awareness among configuration management workspaces. In *Proceedings of ICSE 2003*, pages 444–454, 2003.

[12] T. Schümmer. Lost and found in software space. In *Proceedings of the 34th HICSS*, 2001.

[13] B. Zimmermann and A. M. Selvin. A framework for assessing group memory approaches for software design projects. In *Proceedings of the conference on Designing interactive systems*, pages 417–426. ACM Press, 1997.

Software Reuse

LASER: A Lexical Approach to Analogy in Software Reuse

Rushikesh Amin, Mel Ó Cinnéide and Tony Veale
Department of Computer Science,
University College Dublin,
Belfield, Dublin 4,
Ireland.
{rushikesh.amin,mel.ocinneide,tony.veale}@ucd.ie

Abstract

Software reuse is the process of creating a software system from existing software components, rather than creating it from scratch. With the increase in size and complexity of existing software repositories, the need to provide intelligent support to the programmer becomes more pressing. An analogy is a comparison of certain similarities between things which are otherwise unlike. This concept has shown to be valuable in developing UML-level reuse techniques. In the LASER project we apply lexically-driven Analogy at the code level, rather than at the UML-level, in order to retrieve matching components from a repository of existing components. Using the lexical ontology WordNet, we have conducted a case study to assess if class and method names in open source applications are used in a semantically meaningful way. Our results demonstrate that both hierarchical reuse and parallel reuse can be enhanced through the use of lexically-driven Analogy.

1. Introduction

Software reuse, in its broadest terms, has been viewed as the reapplication of knowledge from one software system to another [4]. Software reuse reduces development time and cost while improving quality. Most engineering disciplines are based on the reuse concept, from components to formulas and ideas themselves [9]. Nonetheless, in terms of reliability and maintenance effort, software engineering compares badly to other, more physical forms of engineering such as chip design, which ultimately employ the same logical concepts.

In general, a reuse environment comprises a repository of reusable components, together with a mechanism for their retrieval, adaptation and integration. One model of software reuse is the so-called 3C model, involving the notions of *concept*, *content*, and *context* [10]. In this model, the software engineer must specify the conceptual requirements of the component they are seeking. Components are described in terms of their content (data structure used, etc.). In the description of both content and concept, it may also be necessary to provide contextual information. Such an approach has the disadvantage that it involves considerable work in specifying this additional information. Also, the repository of reusable components will continue to grow, thus making the task of finding and choosing an appropriate component more difficult [6].

Analogy involves a structural comparison of two concepts that appear substantially different on the surface but which exhibit important causal or semantic symmetries. Computational models of analogy have shown themselves to be valuable in the development of UML level reuse techniques; examples include *ReBuilder* [2, 3]. Analogy typically has both a linguistic and a conceptual dimension, the latter communicated by the former. Words and their meanings thus play an important role in the effectiveness and comprehension of analogies. In this research we examine whether software artifacts like Java programs exhibit the same reliance on lexical expression for their meaning. It seems clear that good developers choose class, method and variable names that are lexically expressive about the goals of the code. If this is true, we can apply lexical analogy techniques at code level rather than UML-level in order to retrieve matching components from the reuse environment.

The object-oriented paradigm facilitates reuse of code by packaging the most reusable structure and behaviours into distinct classes. The programmer can extend the basic functionality of these classes and/or modify it to get the desired functionality. Our research currently focuses on software written in the Java programming language for a number of reasons. Firstly, Java provides an elegant means of encouraging class creation, extension, composition and abstraction. Secondly, various conventions for Java code suggest that the linguistic elements of Java programs, such as method, class and variable names, are more likely to cor-

respond to meaningful words or phrases in a natural language like English. As such, the innovative core of this research project is the belief that natural language techniques can be productively applied to artificial language constructs like software code to yield a meaningful conceptual representation. By focusing on the actual code level of a new software design rather than the abstract component level, we can exploit a rich vein of lexical-driven opportunities for reasoning about a developer's goal and requirements. While it is true that the general structure of software code does not conform to a natural-language grammar, most modern programmers do employ many of the principles of natural language expression in the naming of classes and methods.

In this paper we present our ongoing work on the application of Analogy to the software reuse domain. We have completed a preliminary study by parsing *JRefactory* [1], an open source refactory tool, to determine if class names and method names follow the natural language technique. We also researched how hierarchical reuse and parallel reuse can be enhanced using a lexically-driven analogical approach. As a basis for our work we are using WordNet, a broad coverage knowledge-base of the English lexicon. In the next section we introduce related work, while in section 3 a brief description of WordNet is given. Section 4 describes analogical reasoning as it is used in LASER. A case study is presented in section 5 and finally, directions for future research and conclusions are discussed in section 6.

2. Related Work

ReBuilder [2, 3], a software tool being developed in the AI Lab of the University of Coimbra, is innovative in the way it uses WordNet to index and retrieve software cases. ReBuilder allows analogical retrieval and mapping between UML descriptions of software systems, and uses analogical transfer to flesh out a new software design based on structural parallels with a pre-existing design. Developers explicitly associate software designs with one or more nodes in the taxonomy, so when the new development project is initiated, the taxonomy can be searched for similar precedents. However, the benefits of this reuse scheme can only be reaped by those developers who take the time to tell ReBuilder how to appropriately annotate and index their software for future retrieval. We also exploit WordNet, but at a much finer level of analysis. By prying into the internal lexical structure of software components, to analyse the names given to the methods and classes and relate these to concepts in the WordNet taxonomy, we create a whole spectrum of new possibilities.

CodeFinder, together with PEEL (Parse and Extract Emacs Lisp), is a software tool that supports the process of finding components for reuse [5]. Repositories are initially seeded semi-automatically with structure and index terms by PEEL. These retrieval structures are used by CodeFinder to allow the user to find semantically-related components. The initial retrieval structures are likely to be incomplete, but CodeFinder also enables the user to add new structure and index terms to the repository to improve future component retrieval. In this way, CodeFinder supports the user in finding reusable components in less-than-perfect repository structures.

3. WordNet

WordNet, a broad coverage knowledge base of the English lexicon from Princeton University [8], is the linguistic core underpinning LASER. Language is inherently ambiguous, especially at the lexical level, and so WordNet is built around the notion of a *synset*. A synset is an indirect means of denoting a concept or specific word sense, by providing a set of synonyms that can each denote that sense. For our initial experiments with LASER we exploit the large number of synsets defined by WordNet, as well as two semantic relations, *is-a* and *part-of*, that WordNet uses to connect these synsets. For example, `Student` *is-a* `Person` and `Classroom` is *part-of* `School`. WordNet will serve as the knowledge-base component in LASER, allowing the system to predict potential parent classes using lexico-conceptual knowledge. For example WordNet can be used to lookup all synsets containing the word `client` to find synonyms and hypernyms like `Customer` and `Person`, as well as neighbours like `Patient`. This will allow LASER to establish reuse connections between a new class called `Client` and existing classes called `Customer`, `Person` or `Patient`.

4. Analogical Reasoning in LASER

Analogy in its simplest form can be defined as a comparison between pairs and is widely used as a problem solving method. The "Structure Mapping" approach in ReBuilder views analogy as a suggestion of a class diagram based on the query diagram [2, 3]. As its name suggests, structure mapping does not consider the lexical labels assigned to elements in the each structure, assuming instead that the meaning of these elements derives not from names but from their structural relationships with other elements. However, in object-oriented code, class structures are usually annotated with meaningful lexical labels for a reason: to allow human comprehension and insight [7]. Analogical reasoning can be used in two of the most useful reuse techniques in object oriented programming, *hierarchical reuse* and *parallel reuse*. The first, *hierarchical reuse*, occurs when one class is an extension of another class. For example, consider a

context in which a programmer is about to extend a super class for a class called `Client`.

```
public class Client extends _
```

By using analogical reasoning we can implement different strategies to suggest new superclass for the class `Client`. Using WordNet as a knowledge base, we can suggest a superclass that has similar meaning as `Client`, for example `Person`, `Customer` etc.

Analogical reasoning also facilitates the second, and the most ambitious, form of reuse, *parallel reuse*, in which the system recognizes that a developer is about to implement a component that already exists in a similar form. For example, analogical reasoning can be used to determine that a new class, `Client`, is structurally similar to another class called `Patient`, from another application. This recognition of this similarity is both lexical and structural: lexical in the sense that `Client` and `Patient` are taxonomically similar terms in WordNet, occupying neighboring area of the taxonomy under a common parent class called `Person`; and structural in the sense that similarity can be measured by recognizing common class members (such as variables with lexically similar names and types) and by recognizing the lexically similar names of any superclasses.

A major use of analogical reasoning is to perform analogical transfer (also called *candidate inference*), in which a target structure is enriched with structure that is mapped from a more elaborate source. In most structure-mapping systems, transfer is performed using structural criteria only, which can lead to incongruous completion patterns. We intend to conceptually ground the transfer process by making it lexically-driven: a combination of lexical and structural evidence will be required to motivate any transfer of structure from a past design to the current development context.

In order to explain the potential of analogical reasoning in software reuse, we present an example of how such reasoning might be performed. Consider a context, in which a developer begins to create an application for a School, starting with the creation of a class called `Student`. This nascent basis may allow for an existing software design for a Hospital to be retrieved, based on the similarity (in lexical and structural terms) of the `Student` and `Patient` classes. Structure-mapping between the nascent elements of the school system and the existing case for a hospital might then suggest that the school system needs classes for `Desk, Room, Class, Teacher, Principal` and `TeachingAssistant`. LASER may create stubs for these classes automatically and add them to the current project specification. However, while there is lexical evidence for the utility of a `Room` class in the School domain, there is no such evidence for the necessity of a `Desk`, `Class`, `Nurse` or even `Doctor` class. Therefore, LASER may also suggest to the developer to create a `Room` class only.

The structural evidence for a `Nurse`, `Doctor` and `Bed` class is not entirely discarded, however: it is merely put to one side to await a more appropriate reuse opportunity. For example, should the developer begin to create a class for `Teacher`, lexical analysis using WordNet will reveal that this class corresponds to a known taxonomic subordinate of `Professional`, as does `Doctor`. At this point it becomes sensible for LASER to suggest to the developer that the `Doctor` class be reused. Since this is an example of *parallel reuse* (`Teachers` are like `Doctors`, but not a kind of `Doctor`), LASER can suggest two alternative strategies: first, a new generic class `Professional` can be created, and the bulk of the `Doctor` class can be lifted to this new parent class, so that it can be reused via class extension by the new class `Teacher`; or secondly, the structure of the `Doctor` class can be directly imported to provide a code skeleton around which to create the `Teacher` class. In a similar way, the class for `Bed` can be reused as the basis for a new class `Desk`, and if the conceptual structure is available to make such a suggestion, `Nurse` can be reused as the basis for `TeachingAssistant`.

5. Case Study

In this section we demonstrate that *hierarchical reuse* and *parallel reuse* in object oriented code can be enhanced through lexically-driven analogy. Our fundamental assumption is the belief that most programmers follow natural language conventions in naming classes and methods. As a pilot study, the *JRefactory* package [1], which contains 1259 classes, was parsed for the purpose of collecting and storing class and method names. We used WordNet to lexically ground class names and method names used by the programmer. The distinction between names that are linguistically sensible and those that are not is a fuzzy one: many names contain a linguistic root that can be extracted with some basic language heuristics. To calculate the accuracy of such a heuristic WordNet match with a particular class or method name, we initially decompose the name using the capitalisation convention used as standard in Java We have also included some common Java words in our experiment that are not part of standard WordNet database: *AST*, *int*, etc. An example of how we interpret a class name is as follow:

Class Name	Match after decomposition			Average match
MyBeerCase	My=100%	Beer=100%	Case=100%	100%
MyBeerPZK	My=100%	Beer=100%	PZK=0%	66.66%

Table 1: Average accuracy match for each class with WordNet

Once names have been decomposed following the capitalisation convention, we then compare each component word against the WordNet synset index and calculate an average accuracy for each name as follows:

$$P_{avg} = \sum_{i=1}^{N} (f_e/N) \quad (1)$$

Where P_{avg} is Accuracy of match per class, f_e is accuracy of match with WordNet for each Word and it could be either 0 or 100 and N=Number of Words.

For MyBeerCase it is (100+100+100)/3=100%.
For MyBeerPZK it is, (100+100+0)/3= 66.66%. The final result is calculated as follows.

$$R_{avg} = \sum_{i=1}^{N} (P_{avg}/N) \quad (2)$$

Where R_{avg} is the average accuracy of percentage match with WordNet.
P_{avg} is Accuracy of match per class and N is Total number of class,1259 in this case.
The following graphs depict the result of these techniques applied to *JRefactory* as a whole:

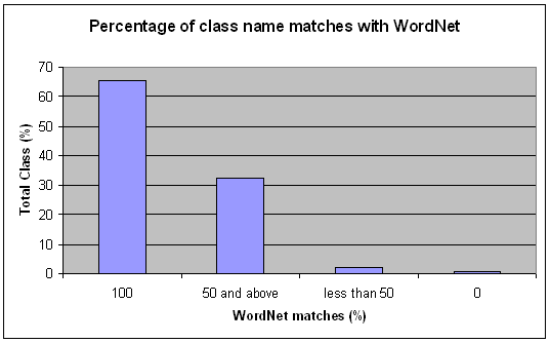


Figure 1. Class Name match with WordNet

The graph above displays the percentage of class names that correspond to WordNet lexical entries. Encouragingly, out of the total number of 1259 classes, an average match of 84.96% with WordNet entries was calculated. We group class name matches into four categories as shown in figure 1. 65.66%, of the classes had a 100% match with WordNet. This was significantly greater than the 0.6% of classes that had no match at all with WordNet.

Figure 2 shows the percentage of method names that correspond to WordNet entries. *JRefactory* contains 6966 methods, and of these, 74.33% match with WordNet entries. The vast majority (60.70%) achieves a 100% match

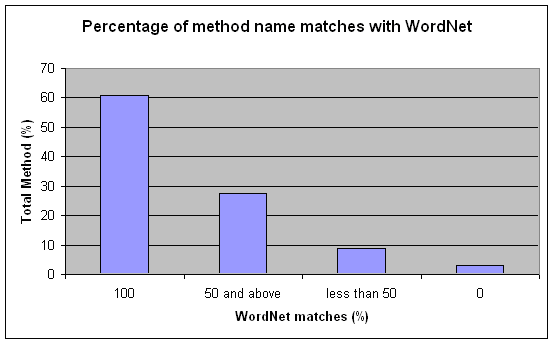


Figure 2. Method Name Match with WordNet

with WordNet; while very few (2.97%) has no discernable mapping to WordNet at all.

Although this is as yet only a pilot study, the results are very promising as regards the goals the LASER project. The vast majority of class names and their lexical components, and a strong majority of method name components, are amenable to conceptual annotation using linguistic techniques.

It is not enough that code-level names correspond to known words; these names must be used in ways consistent with their meaning so that the conceptual basis of the code can be discerned. Ideally, when subclass and its superclass can both be mapped to WordNet, we should expect the corresponding synsets to explicitly relate via an is-a relationship.

To test this expectation, we performed another experiment to consider the *hypernyms* of the lexical labels associated with class names. Hypernym is linguistic term for a word whose meaning includes the meanings of other words, as the meaning of `Transportation` includes the meaning of `Airplane`, `Train` and `Automobile`.

To conduct this experiment we parsed 1259 classes from the open source *JRefactory* [1] project. We completed a study to see if the subclass name and its superclass name have a *hypernym* relationship or not. We explore WordNet to conduct this experiment. Results are calculated as follows: Suppose subclass name is `MySchool` and superclass name is `Organization`. We developed a technique to decompose the names using the capitalization standard used in Java. We use WordNet to search for the *hypernym* relationship between linguistic-head of subclass name(`School` in this case) and those of superclass name(`Organization` in this case). If we find *hypernym* relationship then we associate it with 100% match otherwise with 0% match.In this case it is 100% match. Results are shown in Table 2.

	Total class	**Hypernym relationship** (%)	**No relationship** (%)
Classes	1259	84.51	15.49

Table 2: Hypernym Relationship between subclass and superclass

We focused only on subclasses and their associated superclass. We discovered 1259 superclasses in the *JRefactory* project; of these 84.51% had a *hypernym* relationship with their subclasses. Based on this impressive result, we can argue that a subclass and its superclass are lexically similar and if also, future experiment prove structural similarity, as we suspect, then this presents as with a real opportunity for future *parallel reuse*.

6. Conclusion and Future Work

Analogical Reasoning enables a software environment to provide automated support for software reuse. Lexically-driven analogy at the code level can be used for better understanding of the developer's conceptual goals. In this paper we presented ongoing work on the concept of lexically-driven analogy. We have shown that both *hierarchical* and *parallel reuse* can be enhanced using lexical similarities of class names.

Future work on LASER will be focused on the implementation of techniques for suggesting a super-class for the current class under development using lexically-driven analogy. In our next case study we will consider variables, calculating the *part-of* relationship between a variable name and its class name. LASER will also facilitate intelligent refactoring, based on lexico-conceptual understanding of the software being developed.

7. Acknowledgement

The authors would like to acknowledge the financial contribution of Faculty of Science, University College Dublin to this project.

References

[1] *JRefactory, An Open Source Refactoring Tool for Java*. http://jrefactory.sourceforge.net.

[2] P. Gomes, F. Pereira, P. Paiva, J. Ferreira, and C. Bento. Case retrieval of software designs using wordnet. *ECAI-2002, the Fifteenth European Conference on Artificial Intelligence*, 2002.

[3] P. Gomes, F. Pereira, P. Paiva, N. Seco, J. Ferreira, and C. Bento. Experiments on case-based retrieval of software designs. *ECCBR-2002, the 2002 European Conference on Case-Based Reasoning*, 2002.

[4] M. T. Harandi. The role of analogy in software reuse. *ACM symposium on applied computing states of the art and practice*, 1993.

[5] S. Henninger. An evolutionary approach to constructing effective software reuse repositories. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 1997.

[6] C. W. Krueger. Software reuse. *ACM Computing Surveys (CSUR), Volume 24 Issue 2*, 1992.

[7] H. E. Letha and G. D. Carl. Automatically identifying reusable oo legacy code, computer. 1997.

[8] G. A. Miller. *WordNet,Cognitive Science Laboratory, Princeton University*.

[9] R. Prieto-Diaz. Status report- software reusability. *IEEE Software, v10 n3*, May 1993.

[10] W. Tracz and S. Edwards. Implementation working group report. *Reuse In Practice Workshop, Software Engineering Institute, Pitt, Pa*, 1989.

A Case Study on Recommending Reusable Software Components using Collaborative Filtering

Frank McCarey, Mel Ó Cinnéide and Nicholas Kushmerick
Department of Computer Science,
University College Dublin,
Belfield, Dublin 4, Ireland.
{frank.mccarey, mel.ocinneide, nick} @ucd.ie

Abstract

The demand for quality, highly functional software reinforces the need for reusable software components. However, as repositories of reusable components increase in size and complexity, the challenge for developers to remain conversant with all components becomes greater. This paper proposes a software recommendation system based on collaborative filtering, which has been shown to be effective in other domains. Based on the usage patterns of existing classes and the class currently being developed, our system proposes a set of reuse candidates to the programmer. We present the results of our analysis of the usage of Swing classes in several open-source applications and find that the collaborative filtering technique is promising in providing recommendations in this context.

1. Introduction

It is of growing importance for enterprises to have effective reuse of software components as they invest in developing and maintaining large software systems [6]. Software reuse is an approach to developing systems where artifacts that already exist are used again. Artifacts vary from software components to analysis models; this paper concentrates exclusively on software components. A component is a well-defined unit of software that has a published interface and can be used in conjunction with other components to form larger units [9]. Using existing components can help develop better, faster and cheaper software systems in an industrial context, e.g. [8].

Developers are not always eager to use reusable components, even if these components may be useful and improve productivity. A Productivity Paradox, as identified in [5], exists. Although reusable components for solving this problem are available, most developers are not motivated to learn these reusable components. The reasons behind the lack of motivation are discussed in [13]. A developer will give preference to a suboptimal solution as they perceive the time and effort to locate and learn components to be too costly. Even if a developer is willing to reuse a component they may not be able to locate it. As the repositories of components increase, there exists a real challenge for developers to remain conversant with all components. To assist software developers in making full use of large component repositories, information access needs to be complemented by information delivery [14].

Software repositories contain a wealth of valuable information. Usage of software components can be automatically extracted from these repositories. This may then be used to infer links between components, examine usage patterns of a component or set of components and examine the relationship between the user task and components used.

This paper describes a recommendation system, based on a collaborative filtering approach, that allows developers discover reusable components for the purpose of supporting learning on demand, improving developer productivity/quality and promoting software reuse. Repositories of open-source Java code, available from *SourceForge* [4], are mined and usage histories of components are automatically collected. Based on the collaborative filtering approach, we use collected histories to recommend to the developer a set of candidate components that may be useful to this individual developer.

Learn on demand is an approach which allows users to learn new information or components as they are needed. Benefits of this style of learning are discussed by [13]. One such benefit is that the user can immediately see the convenience of this reusable component for an authentic problem situation, thereby increasing the motivation for learning. The system will proactively provide users with task-relevant and personalised recommendations by inferring the need for components based on a collaborative filtering tech-

nique.

A basic principle of most collaborative systems is that most users can be clustered into groups. Users in a group share preferences and dislikes for particular items and are likely to agree on future items. A recommendation for a user is based on the opinions or ratings of other like-minded users. This principle can be extended to software classes. A Java class can be considered a user and a software component as an item. A Java class may use zero or more components. If two Java classes share similar ratings for a component, or a set of components, then there is evidence that they will share the same ratings in future components. Thus a recommendation for a particular component to a user (Java class) is based on the rating of like-minded users on this component.

The rest of the paper is organized as follows. Related work is introduced in section 2 while section 3 formally describes the collaborative filtering process and details the algorithms used. Results are displayed and evaluated in section 4 followed by a discussion on future research directions in section 5. Finally Section 6 shows conclusions.

2. Related Work

Traditional methodologies for component search and retrieval can be classified into four different categories, namely *Keyword Search*, *Faceted Classification*, *Signature Matching* and *Behavioral Matching* [10]. Each of the retrieval schemes have a number of limitations that result in less than adequate retrievals. A common shortcoming of these schemes is the failure to take into account the user or relevant domain information when querying the component repository.

Semantic-Based Method Retrieval [12] improves on the above; the user specifies component requirements using natural languages. Domain information, user context and component relationships are all considered. This methodology relies on ontology as a knowledge base. Empirical results indicate this technique is superior to traditional schemes. However, in reality developers are not aware of all available components. If they believe a reusable component for a particular task does not exist then they are unlikely to query the component repository. Thus component retrieval must be complemented with component delivery.

CodeBroker [13] infers the needs for components and proactively recommends components, with examples, that match the inferred needs. The need for a component is inferred by developer comments and method signature. This solution greatly improves on previous approaches however the technique is not ideal. Firstly the reusable components in the repository must be sufficiently commented, to allow matching, this may exclude many components. Secondly the developer must actively and correctly comment his/her code.

Ohsugi et al. [11] propose a system, based on collaborative filtering, to recommended useful functions in application software such as MS-Word or MS-Excel. Software usage or function execution histories are automatically collected from many users in order to provide opportunities to commence user clustering. A set of candidate functions is then recommended to the individual, based on the opinions of like-minded users. Our work applies a similar approach to a different problem domain, namely reusable software components, recommending a set of candidate software components to the developer, based on the opinions of like-minded users.

3. Collaborative Filtering

The goal of collaborative filtering algorithms is to suggest new items or to predict the utility of a certain item for a particular user based on the user's previous likings and the opinions of other like minded users [1]. The first recommender systems based on collaborative filtering to automate predictions was *GroupLens* [3]. Collaborative filtering systems are founded on the belief that users can be clustered. Users in a cluster share preferences and dislikes for particular items and are likely to agree on future items. In the context of this paper, a user can be considered to be a Java class and an item refers to a software component.

Collaborative filtering algorithms can be divided into two classes, Memory-Based algorithms and Model-Based algorithms. Memory based algorithms operate over the entire user database to make predictions. In contrast, Model-Based algorithms use the user database to learn a model which is then used for recommendations. Memory-based methods are simpler, seem to work reasonably well in practice and new data can be added easily. For these reasons we decided to use a memory-based algorithm. A key aspect of collaborative filtering is the identification of similar users to the active user for whom the recommendation is being sought. This similarity is based on item usage history. Our prediction system has 3 components:

1. Usage History Collector.
2. User Similarity Analyser.
3. Recommender.

3.1. Usage History Collector

The usage history collector records all invocations of a particular method. The collector stores information about both the user (invoking class) and the item (method invoked). A distinction is made between overloaded methods by recording method signatures. The collector is implemented using the Byte Code Engineering Library [2]. This data is then transformed into a user-item matrix.

3.2. User Similarity Analyser

Each user is treated as a vector; the vector holds a count for all components/methods that the user can invoke. Count will hold a value of zero if the user has never used the particular method. The similarity between two users can be computed by determining the cosine of the angle formed by their vectors. This cosine will fall in the range [-1, 1]. A cosine of 1 indicates two users are identical. A cosine of -1 indicates two users share no similarities. This technique needs to be tailored to the problem domain. To illustrate this, table 1 displays the usage history for a small number of Swing method invocations taken from random GUI Java applications in *SourceForge* [4].

Item	User	User	User
	RemoteD	**HostList**	**CompileDlg**
JTextComponent: getText	1	1	8
JTextComponent : setText	2	2	4
JList: getSelectedValue	2	1	0
JList: getSelectedIndex	**1**	**2**	**0**

Table 1: Sample Usage History for Swing Methods

If compared on the *JList* method *getSelectedIndex*, user *RemoteD* and user *HostList* will be considered as similar as user *RemoteD* and user *CompileDlg*, as both counts differ by one. However it is more likely that user *RemoteD* is more similar to user *HostList* than user *CompileDlg*. This is because we know that both *RemoteD* and *Hostlist* have *JList* objects and that both need to get the index of the item in the *JList* which the user has selected, therefore we can see clear similarities between the two classes at both code and task level. Hence a check is needed before vector similarity is computed to ensure both users have used this item, if not then an arbitrary small similarity value of 0.2 is assigned.

3.3. Recommendations

Recommendations were based on the following algorithm. Let *U* and *C* denote the set of similar users and components respectively. The recommendation R_{ac} for user *a* on component *c* is:

$$R_{ac} = \sum_{i \in U} (v_{ic} \times sim_{ai}) \quad (1)$$

where v_{ic} is the count for user *i* on component *c* and sim_{ai} is the similarity between user *a* and user *i*, as calculated in subsection 3.2. A recommendation is made if R_{ac} is greater than or equal to the arbitrary threshold value, τ = 1. The results of this equation depend on the component count of other users who are similar to user *a*. A class that is dissimilar to user *a* will have minimal impact on the final recommendation.

The set of users *U* is based on the standard *k*-Nearest Neighbour algorithm. The optimal value for *k* was found to be 4. Hence for the following experiments we used *k* = 4 neighbours.

4. Recommendation Evaluation

4.1. Outline of Experiment

We have conducted experiments to investigate whether the algorithm described in Section 3.3 could accurately predict useful components to the developer. The component repository used in the experiment contained 1123 methods from the standard Java Swing library. Recommendations were made for a total of 343 Java classes, taken from over 40 GUI applications on SourceForge. In each class several sets of recommendations were made. For example, if a fully developed class used 10 Swing components, then we removed 1 random component from the class and a recommendation set was produced for the developer based on the remaining 9 components. Following this recommendation a further component was removed from the class and a new recommendation set was formed for this developer based on the remaining 8 components. This process was continued until all but 1 component remained.

4.2. Evaluation

Precision and *Recall* are the most popular metrics for evaluating information retrieval systems. *Precision* is defined as the ratio of relevant recommended items to the total number of items recommended, as shown in equation 2. This represents the probability that a selected item is relevant.

$$P = N_{rs}/N_s \quad (2)$$

where N_{rs} is the number of relevant items selected and N_s is the number of items selected. An item, or component, is deemed relevant if it is used by the developer. *Recall*, as shown in equation 3, is defined as the relevant items selected to the total number of relevant items. This represents the probability that a relevant item will be selected.

$$R = N_{rs}/N_r \quad (3)$$

where N_{rs} is the number of relevant items selected and N_r is the number of relevant items.

4.3. Experiment 1

Initial experiments were carried out on the most suitable classes for prediction. A suitable class has one or more close neighbours, i.e., similar classes. An unsuitable class

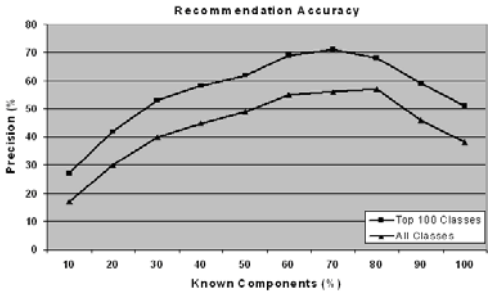


Figure 1. Average Accuracy of Recommendations

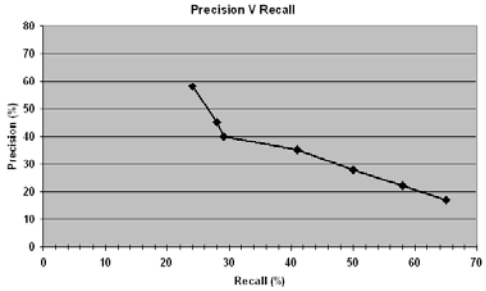


Figure 2. Precision versus Recall

has no close/similar neighbours and thus reduces prediction quality. The upper plotted line in figure 1 displays the accuracy of recommendations for the 100 most suitable classes for predictions:

Focusing on the recommendations for the top 100 classes (the upper plotted line); the first point displays the average accuracy of a recommendation when the developer has utilised between 0% and 10% of the total components they will use. When the developer has utilised only 10% to 20% of the components, recommendations are remarkable good with an average accuracy of over 40%. Recommendation accuracy steadily increased until over 70% of components that the developer will actually employ are used. The decline in accuracy after this point can be explained as follows: Consider a class that uses 10 components and the system is making a recommendation when 50% of the components are known, i.e., 5 components. If the system correctly recommends the remaining 5 components plus 2 incorrect components, then recommendation accuracy will be 71%. However, if the system is making a recommendation when 90% of the components are known and correctly recommends the 1 remaining component plus the 2 incorrect components then recommendation accuracy will be 33%. It is likely that the two incorrect components are used in a very similar class to this active class for whom the recommendation is being sought, thus explaining the recommendation in the first place. One possible way to overcome repeating incorrect recommendations is to allow the developer to explicitly reject a recommended component. As a result the component will not be suggested again. It is the author's belief that a number of the incorrect recommendations may not have been out of context, that is to say the incorrectly recommended component may indeed have been suitable for the active class.

The promising results for the top 100 classes illustrate the system potential as an effective recommender system. The lower plotted line displays the average recommendation accuracy for all 343 classes. When the developer has utilised between 10% and 20% of the total components that they will actually use, recommendations are 25% accurate. Recommendation accuracy peaked to 43% when between 50% and 60% of components were employed. The noticeable fall in recommendation accuracy is due to the fact that recommendations are for all classes, including classes that are unique or at least considerably different from all other classes under test.

To evaluate the performance of the recommendation algorithm, it is important that both *precision* and *recall* are considered together. Figure 2 displays the trade off between precision and recall. *Prediction* and *Recall* figures are an average value based on recommendations for all classes. As expected the lower the recall, the greater the precision. When threshold $\tau = 3$, as discussed in subsection 3.3, *recall* is relatively small however precision is almost 60%. When $\tau = 0.1$, *recall* improves greatly at the expense of *precision*.

4.4. Experiment 2

It has been suggested that 40-60% of code is functionally identical to previously written code [7]. Therefore if we increase the number of usage histories collected it is likely that the probability of finding a class similar to the active class, for whom the recommendation is being sought, should also increase. 100 usage histories were randomly removed from the usage history database leaving a total of 243. The fall in recommendation accuracy, as shown in figure 3 on the following page, confirms the relationship between the number of usage histories and recommendation accuracy.

The average recommendation accuracy for the top 100 classes is 43%. This compares poorly to an average of 56% when all 343 usage histories are collected. This would suggest that the greater the number of usage histories collected, the more reliable the recommendations. Clearly this trend will not continue indefinitely, at some point adding more usage histories will have little or no effect on recommendation accuracy.

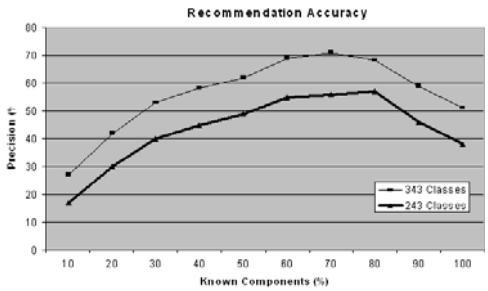


Figure 3. Average Accuracy for top 243 classes

5. Future Work

In our system, classes are considered similar if they use the same or a similar set of components. This similarity measure needs to be extended. Firstly, we will consider the sequence in which components are actually used and secondly, different granularities of similarity will be considered such as fields, methods and components used. Finally a run-time analysis may be useful in highlighting different run-time usage patterns for a particular component between two different classes.

With any recommendation system it is important to explain how the recommendation was derived. Explanations increase a programmer's confidence in recommendations and provide him/her with a mechanism for handling incorrect recommendations. *CodeBroker* [13] provides an interesting feature that allows developers to view example code of a particular component in use. Experiments carried out on *CodeBroker* discovered that developers prefer code examples as opposed to descriptive texts. We will create an intelligent IDE by developing a non-intrusive component recommender as an Eclipse plug-in. The recommender will support explanation by code example and Java documentation retrieval. Finally confidence measures will be added for recommended components, this measure will be based on the closeness of the active class with neighbouring classes.

6. Conclusion

A recommender approach to enable developers discover useful and relevant reusable software components has been presented. Our recommendation scheme addresses various shortcomings of previous solutions to the component retrieval problem. Recommendations consider the developer and problem domain whilst avoiding placing any additional requirements on the developers

Recommender systems are a powerful technology that can extract additional knowledge for a software company from its code databases and then exploit this in future developments. From several experiments, we have demonstrated that this approach offers real promise for allowing developers discover reusable components with minimal effort.

7. Acknowledgments

Funding for this research was provided by the Irish Research Council for Science, Engineering and Technology under grant RS/2003/127

References

[1] B.Sarwar et al. Item-based collaborative filtering recommendation algorithms. In *Proceedings of the tenth international conference on World Wide Web*, Hong Kong, 2001.

[2] *Apache Jakarta Project*. Bytecode Engineering Library. http://jakarta.apache.org/bcel/index.html.

[3] P. Resnick et al. GroupLens: An Open Architecture for Collaborative Filtering of Netnews. In *Proceedings of CSCW '94*, Chapel Hill, NC, 1994.

[4] *VA Corporation*. SourceForge. http://sourceforge.net.

[5] J. Carrol and M. Rosson. *The paradox of the active user. In J.M. Carroll (Ed.), Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. MIT Press, Cambridge, 1987.

[6] K. Daudjee and A. Toptsis. A technique for automatically organising software libraries for software reuse. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, Canada, 1994. IBM Press.

[7] W. Frakes and P. Gandel. Representation methods for software reuse. In *Proceedings of the conference on Tri-Ada '89: Ada technology in context: application, development, and deployment*, pages 302–314, Pennsylvania, USA, 1989.

[8] M. Griss. Software reuse at hewlett-packard. In *Proceedings of the 1st International Workshop on Software Reusability*, Dortmund, Germany, 1991.

[9] J. Hopkins. Component primer. *Communications of the ACM*, Vol. 43:pages 27–30, 2000.

[10] A. Mili, R. Mili, and R. Mittermeir. A survey of software reuse libraries. *Annals of Software Engineering*, Vol. 5:pages 349–414, 1998.

[11] N. Ohsugi, A. Monden, and K. Matsumoto. Recommendation system for software function discovery. In *Proceedings of the 9th Asia-Pacific Software Engineering Conference (APSEC2002)*, Dec. 2002.

[12] V. Sugurmaran and V. Storey. A semantic-based approach to component retrieval. *ACM SIGMIS Database*, Vol. 34:pages 8–24, 2003.

[13] Y. Yunwen and G. Fischer. Information delivery in support of learning reusable software components on demand. In *Proceedings of the 7th international conference on Intelligent user interfaces*, California, USA, 2002. ACM Press.

[14] Y. Yunwen and G. Fischer. Personalizing delivered information in a software reuse environment. In *Proceedings of 8th International Conference on User Modelling (UM2001)*, Sonthofen, Germany, 2002.

TEMPLATE MINING IN SOURCE-CODE DIGITAL LIBRARIES

Yuhanis Yusof and Omer F. Rana
School of Computer Science, Cardiff University, PO Box 916, Cardiff CF24 3XF, UK
{y.yusof, o.f.rana}@cs.cardiff.ac.uk

Abstract

As a greater number of software developers make their source code available, there is a need to store such open-source applications into a repository, and facilitate search over the repository. The objective of this research is to build a digital library of Java source code, to enable search and selection of source code. We believe that such a digital library will enable better sharing of experience amongst developers, and facilitate reuse of code segments. Information retrieval is often considered to be essential for the success of digital libraries, so they can achieve high level of effectiveness while at the same time affording ease of use to a diverse community of users. Four different matching mechanism: exact, generalization, reduction and nameOnly is used in retrieving Java programs based from information extracted through template mining.

1. Introduction

A Digital Library (DL) may be regarded as a managed collection of information (in digital format) with associated services. These services enable access to this information over a network, and can range from support for selecting and browsing material in the collections, organising and archiving it, and making it available in different visual formats. Often DLs contain a diverse collection of information for use by many different users, and the size of information contained within it can vary. Examples of documents kept in DLs include journal papers, chapters of electronic books and magazines, and product documentation. Various techniques can be used in understanding the content of the DLs – such as classification, association discovery, clustering, visualization of data, information extraction, etc.

Information extraction is the process of capturing structured information from a particular document. Natural language processing techniques can be used to extract data directly from text if either the data, or the text surrounding the data, form recognizable patterns [2]. A key motivation for our work is to facilitate software reuse through information extraction, whereby a software engineer or programmer could make use of existing software packages to create new programs. Software reuse has been shown through empirical studies to improve both the quality and productivity of software development. Our thesis is that software reuse should not just be restricted to reusing software libraries in their entirety, but also enable software developers to understand the process associated with solving a problem encoded in the software library. A software developer may be interested in understanding how a particular feature has been coded in a particular language – rather than perhaps make full use of code that has been implemented by someone else.

Recent efforts in object oriented programming (such as the Common Object Request Broker Architecture (CORBA), and recently Web Services) indicate the significance of writing software as independent services to enable reuse. As software reuse is becoming more important, there is a need to store open-source applications in the format of a searchable repository. Such a repository will store all documents related to an application, which includes the source code, comments in versioning systems (such as CVS), documentation provided with the source code (such as a Javadoc document with Java source code), and details about the structure of the library (such as a class hierarchy), etc.

2. Related Work

There is limited literature in the area of applying template mining for extracting information: Cowie and Lehnert [3] have successfully extracted proper names from documents. Extraction of facts from press releases within a company's financial information system has also been undertaken in a few systems, such as ARTANS [9], JASPER [10] and FIES [1]. It has also been proven that template mining is able to build an abstract of scientific papers [8] and also the extraction of citation from digital documents [4]. Four different template are utilised, one for extracting information about articles, while the others are used for extracting information from citations.

Even though there has been much research done into extracting information from text documents, a significant effort has not been put into extracting information from software packages – especially program source code. Most of the research done in the area of understanding source code is mainly focused in categorizing the programming language used into a particular software component or source code achieve [11]. Ugurel et al. [13] classified source code into appropriate application domains and also programming languages using three components, namely the feature extractor, vectorizer and Support Vector Machine classifier. Paul and Prakash [12] have produced a framework which uses pattern languages to specify interesting code features. The pattern languages were derived by extending the source programming language with pattern-matching symbols. They transformed the source code into specific symbols by including a set of symbols that can be used to substitute syntactic entities in the programming language. In this paper, we discuss the usage of template mining in retrieving relevant Java programs stored in software packages, within a Digital Library of source code.

A related area that has been investigated by others is the submission of queries to retrieve particular source code by name. One example of this is the retrieval of particular numerical algorithms via email. Dongarra and Grosse [5] demonstrate this with reference to their Netlib Digital Library. Our approach differs from this in that we are interested in a variety of search techniques – and not just an exact match. Furthermore, many such approaches are restricted to particular types of applications (numerical algorithms in this case), and therefore are restricted in their scope. We also see limited use of existing search engines for this particular problem, as search engines such as `Google.com` or `Altavista.com` do not provide any support for formulating a query based on program structure. Therefore template search mechanisms provide users, especially programmers, with specific search capabilities without neglecting the use of keyword search as offered by Unix utilities like `grep`.

3. Java Template Miner

In developing our Template Mining approach, we are making the following assumptions:

- Users have some indication of the types of source code they are interested in. This could be in terms of the keywords they assume to be present within such source code, or the likely method names that such source code could contain. Although not likely to be valid in a general case, we have found this assumption to hold true based on the existing source code archives such as `Sourceforge.net`. Perhaps one reason for this is that developers who offer their source code for use by others often also attempt to describe their data structures or method names with comments that could be relevant for others.
- Users are familiar with the likely structure of the source code they are trying to find. This may be particularly true for numerical approaches (where nested loops are often used over arrays or similar data structures). Often many programming languages targeted towards the scientific computing community provide specialist support for such data structures (examples include OpenMP and High Performance Fortran).

A prototype system is currently under development – and primarily focused at Java developers. The general architecture of the prototype is described in Figure 1.

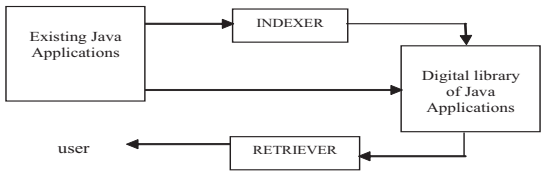


Figure 1. General architecture of source-code digital libraries

Two main components are included in the architecture, the INDEXER and the RETRIEVER. Existing Java applications are fed into the INDEXER, which will finally produce two text files: *index.txt* and *indexMethod.txt* file. Each source code file is analyzed individually, and is used to build a set of keywords. The index file now contains relevant keywords which represent each of the source files contained in the archive - and divided into the application from which the source file has been obtained. The index file thus establishes the rules to generate the internal representation of both the queries and content of software packages.

An index I is defined as a set of terms from each software package or user query: $I =< S_1, S_2, ...S_n >$, where S_i is a set of terms obtained from a software package, and P_j is the number of software packages in the repository. Therefore, $S_i \subseteq P_j$, such that $S_i = \{t_{i1}, t_{i2}, ..., t_{ik}\}$, where $t_{ij} \in P_j$, $(1 \leq j \leq k)$, $(k \leq \mid P_j \mid)$.

On the other hand, RETRIEVER extracts relevant Java source code from the DL that fully or partially matches the requested queries posed by a user. In general, it is defined as follows: let *C* be the set of all possible portions of Java source code for a given query. Let *A* be the set of all available portions of Java source code in the repository. Let $f : C \rightarrow PA$ where *PA* is the power set of *A*. Given a target query: $I_t =< S_1, S_2, ...S_n >$ the function(*f*) returns a set of Java programs from *A* that are the same or similar to I_t. We may specify the function as: $f(I_t) = A'$, where

$A' \subseteq A$, and $A' = \{I_{x1}, I_{x2}, ..., I_{xy}\}$, where $I_{xi} \in A$, and $(0 \leq y \leq \mid A \mid)$.

Users are given two choices, either to use the keyword/phrase query or the program template query. Here we will only be focusing on retrieving relevant Java files based on a program template. Several different types of information are automatically generated from the template query which will be mapped against the index files. In-

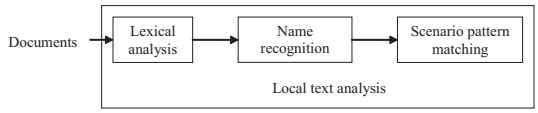


Figure 2. Structure of an information extraction system

formation extraction in this context involves two stages: extracting individual facts from the text, and integrating these facts to create a coherent understanding of the program. In doing so, we follow some elements contained in the structure of an information extraction system [7]. From Figure 2, lexical analysis is done by dividing the plain text into tokens which will then be used to treat each word separately. Each tokens is then referenced to a Java dictionary to determining its usage. For example token of `import, public, static, class` etc are recognized and accepted, while tokens containing `the, method, processing` etc are not. Following the analysis is the name recognition process where class name and method signature are analysed. Finally, pattern matching is implemented to integrate both facts captured through lexical analysis and name analysis. An example of a program template used in our experiment is provided below:

```
public class Matrix {
public Matrix getMatrix (int a, int b, int[]c) {
;
// body of the method //
}
public Matrix transpose () { }
public Matrix plus (int B) { }
public Matrix minus (double B) { }
}
```

Mining from the above template, several information can be extracted and mapped against the *index.txt* and *indexMethod.txt* files. These two files were initially created based on the software packages contained in the source code repository. They contain keywords captured from all text documents and source code programs of the repository. These include the class and method signatures which can then be used to retrieve similar Java programs from the repository. Class name is recognized by the Java keyword `class` while method signatures are recognized by three identifier: name, return type and the first parameter (if it exists). If there are no parameters provided in a particular method, the system will automatically replace it with the string 'none'. Currently, this system only recognizes seven main data types: `String, char, Boolean, int, double, float, long, byte` and `short`. If any other data types were found, the system will then automatically replace it with 'none'.

Based on these, the search process is undertaken using four different methods (described in more detail below): (1) exact matching, (2) generalization, (3) reduction, and nameOnly. In the exact matching procedure, class name and method signature are compared. If the system fails to find an exact match (equivalent to a Boolean AND) where all requirements (class and method signature) are successfully matched, it will then use the generalization method. In this method, the system will change some of the parameters in the method signature. For example, consider a Program Template:

```
public Matrix minus(double B) -> public Matrix
minus (float B)
```

where the data type double for argument B will be changed to data type float. Java files will be retrieved only if it matches the method name and either the return type or the first parameter. Alternatively, the search mechanism will retrieve Java programs based on the reduction technique (based on a Boolean OR) during the matching process. If either the return type or the first parameter is matched then the program is considered relevant to the query. Compared to the previous matching mechanism, reduction matching does not replace any data types before processing. Finally, the search mechanism will retrieve all Java programs which have the same method name while ignoring its signature. The final outcome of the template mining in source-code digital libraries is a list of Java programs, and this list is returned in a text file – *resultTemplate.txt*. These four approaches can be used as a first step in selecting suitable programs from an archive – with increasing degree of closeness to a template.

4. Result Analysis

Currently a Java repository of 114 MB has been built by storing 30 Java packages in it. Two index file are generated initially before template mining is done: *index.txt* and *indexMethod.txt*. `Table 1` shows two factors (time and file size) being observed during the process of generating index files and undertaking template mining in the Java source-code repository.

Table 1. Time and file size of generated index files

FILE (name)	TIME (ms)	FILE SIZE (bytes)
Index.txt	3140144	110590248
IndexMethod.txt	89591	2365555
resultTemplate.txt	7703	573

The processing time and file size of *index.txt* is longer and bigger as it contains all relevant words captured from the Java repositories. On the other hand, *indexmethod.txt*, only contains relevant keyword for method signatures. On completing the template mining and search process, the result is written into file *resultTemplate.txt*. Based on the program template used in the experiment (Section 3), the processing time taken to find and retrieve relevant Java programs is 7703 millisecond.

A precision and recall analysis is undertaken for comparing the effectiveness of source-code retrieval (`Table 2`). According to Frakes and Baeza-Yates [6], "Recall" is defined as the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in a repository. Similarly, "Precision" is defined as the ratio of the number of documents retrieved that were found to be relevant (as identified by a human expert) over the total number of documents retrieved from the repository via a given query. We use the same definition here for tabulating results of our experiment based on the program template mentioned in Section 3. Where the above definition says documents and database, the word Java programs and repository respectively can be substituted. We similarly adapt some of the other concepts identified when defining Recall and Precision for text documents. When evaluating Recall, we therefore investigate that out of all the relevant Java files in the repository, how many did our search actually retrieve? For evaluating Precision, we determine out of all the Java files that were extracted during a search, how many were relevant. According to Frakes [6], "both Recall and Precision take on values between 0 and 1" (or between 0 and 100 when expressed as percentages). `Table 2` presents the recall and precision figures for each search performed normalized to lie between 0 and 100.

Each of the extracted information (class name and method) is analyzed separately for an exact string and a sub-string match. Exact string matching is used to retrieve programs with the exact class name or method name, while sub-string matching is used to retrieve programs with the target string being a part of one string or if there exist any string as a substring of the target string. For example, if a user is searching for `Matrix.Java`, the file name `createMatrix.java` will also be retrieved through sub-string matching. Let M_i be the method name in the index file, and M_q be the target (query) method name. If the length of target string is less than the index term, we will accept Java programs which produces $(0 \leq (M_i.length - M_q.length) \leq x)$, otherwise the accepted string matching must fulfill $(0 \leq (M_q.length - M_i.length) \leq y)$. The values of x and y are chosen to be 10 and 5 respectively – these are heuristics based on our current data set.

Table 2. Precision and recall analysis of sub-string matching

Class or Method	SUBSTRING	
	Precision	Recall
Matrix	100%	80%
Matrix getMatrix(int)		
Exact matching	100%	100%
Generalization	-	-
Reduction	0%	-
nameOnly	100%	100%
Matrix transpose()		
Exact matching	100%	100%
Generalization	-	-
Reduction	0%	0%
nameOnly	100%	80%
Matrix plus(int)		
Exact matching	-	-
Generalization	-	-
Reduction	100%	50%
nameOnly	100%	50%
Matrix minus(Matrix)		
Exact matching	0%	0%
Generalization	-	-
Reduction	100%	33.33%
nameOnly	100%	33.33%

From the result, retrieved documents based on string matching produced the result of 100% for both precision and recall analysis. However, this does not happen in sub-string matching where recall analysis produces an average of 64.92%. The issue here is whether we would like to consider sub-strings within a sub-string to also be relevant during the search process. Although we see benefits of undertaking such an analysis in particular instances, the general benefit of this approach is not obvious to us. In this case, recall analysis would be 0% as the system is not able to produce any result. The current system is not able to extract sub-strings contained within a string automatically, and search for these as query terms. As long as the Java program (source string) contains the searched string (ignoring the sub-string length and location of each element), currently we still consider it as a precise retrieval. This leads us to obtain 100% precision as all the retrieved documents are found to be relevant to the query. Meanwhile, as for recall analysis, the system fails to retrieve some relevant java programs which then produced an inconsistent recall percentage. (`Table 2`) also contains elements of '-' as the answer set for the query is 0 and therefore, we could not calculate the percentage. From four retrieval techniques mentioned

in Section 3, retrieved Java files are categorized as follows:

1. matching of class name and all methods signature
2. matching of class name and all methods signature (generalization)
3. matching of class name and all methods signature (reduction)
4. matching of class name and all methods signature (nameOnly)
5. matching of class name only
6. matching of method signature only

Each retrieved document is being given different points if it is classified under the above categories. The values are then summed over all the categories to obtain a ranking for the retrieved files (in terms of their relevance to the initial query). Documents with the highest value of points will be ranked as the most relevant program to the program template defined by a user.

Comparing this process to a software repository such as `Tucows.com` where users are presented with software to be downloaded (in different categories), this system presents Java source code to help users (especially programmers and system analysts) in reusing existing software libraries. Similarly, in `SourceForge.net`, open-source applications are catalogued based on their particular categories, such as Programming Language and Operating System, etc. The search process utilised in `SourgeForge` makes use of keywords, and is based on general descriptions given to each of the stored packages. In other hand, both `SourceForge` and `Tucows` do not allow any source code retrieval based on users query. We see our approach as an obvious extension of the search process supported by such public domain software repositories – and our current test set is based on a subset of source code obtained from `SourceForge`.

5. Conclusion and Future Work

With the emerging interest in making source code available, and the significant emphasis being placed on this by many software architects, DLs that support the searching for source code have become necessary. The importance of the reuse of software artifacts has been discussed as a key motivator for the implementation of such source-code digital libraries. A system for supporting this is identified, to support users in locating Java source code. Relevant information is extracted from a program template provided by the user and this is compared against index files generated from software packages stored in the repository.

Currently, as a continuity to the research, we are extracting other information from the repository which includes class hierarchies and associations between different classes within a software. This is being achieved by automatically developing a Unified Modelling Language (UML) diagram (using reverse engineering tools), and by automatic extraction of design patterns from the source code. With this, users will have more choices in creating their query, and specifying their search criteria. Relevant programs will then be presented to the user ranked according to their relevance of both exact/inexact keyword search, and relevance based on program structure. We also intend to make better use of software categories in existing source code archives, and include these with the approaches mentioned above.

References

[1] W. Chong and A. Goh. Fies:financial information extraction system. *Information Services and Use*, 17(4):215–223, 1997.

[2] G. Chowdury, N. Kemp, M. Lawson, and M. F. Lynch. Automatic extraction of citations from the text of english-language patents - an example of template mining. *Information Science*, 22(6):423–436, 1996.

[3] J. Cowie and W. Lehnert. Information extraction. *Commun. ACM*, 39(1):80–91, 1996.

[4] Y. Ding, G. Chowdhury, and S. Foo. Template mining for the extraction of citation from digital documents. *Library Trends*, 48(1):181–207, 1999.

[5] J. J. Dongarra and E. Grosse. Distribution of mathematical software via electronic mail. *Communications of the ACM*, 30(5):403–407, 1987.

[6] W. B. Frakes and R. Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc., 1992.

[7] R. Grishman. Information extraction: techniques and challenges. In M. T. Pazienza, editor, *Information Extrcation*, Springer-Verlag Lecture Notes in AI, pages 10–27, Rome, 1997.

[8] P. Jones and D. Chris. A 'select and generate' approach to automatic abstracting. In T.McEnery and C.D.Paice, editors, *Proceedings of the BCS 14th Information Retrieval Colloqium*. Springer-Verlag, 1992.

[9] S. Lytinen and A. Gershman. Atrans:automatic processing of money transfer messages. In *Proceedings of Fifth National Conference on Artificial Intelligence*, pages 1089–1093, Los Altos, CA, 1986. Morgan Kaufmann.

[10] M. A. Peggy, J. H. Philip, K. H. Alison, M. S. Linda, B. N. Irene, and P. W. Steven. Automatic extraction of facts from press releases to generate news stories. In *Proceedings of the 3rd Conference on Applied Natural Language Processing*, pages 170–177, Trento, Italy, 1992.

[11] P. Ruben and F. Peter. Classifying software reuse. *IEEE Software*, 4(1):616, 1987.

[12] P. Santanul and P. Atul. A framework for source code search using program patterns. *IEEE Transaction on Software Engineering*, 20(6):463–475, 1994.

[13] S. Ugurel, R. Krovetz, and C. L. Giles. What's the code?: automatic classification of source code archives. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 632–638. ACM Press, 2002.

Multi-Project Software Engineering: An Example

Pankaj K Garg
garg@zeesource.net
Zee Source
1684 Nightingale Avenue, Suite 201, Sunnyvale, CA 94087, USA

Thomas Gschwind
tom@infosys.tuwien.ac.at
Technische Universität Wien
Argentinierstraße 8/E1841, A-1040 Wien, Austria

Katsuro Inoue
inoue@ist.osaka-u.ac.jp
Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

Abstract

In this paper we present an approach for developers to benefit from multi-project software knowledge. As we show in this paper, this can be achieved by gathering information about how numerous software projects are being built, and about the interrelation of the modules within the projects. Compared to approaches that only monitor a single project, the contribution of our approach is that it not only supports the reuse of isolated software modules or libraries but also the knowledge surrounding the code and individual projects. For instance, if a component is replaced with another probably better implementation within a project, this knowledge can be shared with **all** *relevant projects. In this paper, we show how the collection of such data allows developers to learn about such decisions from other projects, and hence how to benefit from such "multi-project" knowledge.*

1. Introduction

Recent advances in computer and networking hardware have enabled the collection and analysis of huge amounts of information. In this paper we present the idea that such advances can be leveraged for *Multi-Project Software Engineering*, i.e., engineering of thousands of software projects simultaneously. In the past, software reuse has focused on sharing code among projects, whether through black-box or white-box reuse. With Multi-Project Software Engineering, not only does code get reused across projects, but *knowledge* surrounding the code and the project gets extensively reused among the projects.

For example, if there's a break-down in a reusable component in one project, then information about the problem can be instantaneously broadcast to the thousands of users of that component. Correspondingly, each of the other projects does not have to independently discover the problem, and waste time through redundant problem resolution processes. Similar mechanisms can be put in place to avoid extra wasted effort when temporary or permanent fixes are discovered for the component's problem.

Enabling such Multi-Project Software Engineering requires a networked infrastructure that manages process and product information for multiple projects effectively. Design choices of what information gets stored, and how, will have substantial impact on the functionality and power of the resulting engineering processes. To that effect, we borrow heavily from the lessons learned from decades of working of the Open Source communities [14]. As such, we describe an architecture of storing and utilizing Multi-Project Software Engineering data, leveraging some of the key technologies developed by the Open Source community for supporting their own software development processes.

We are actively researching different methods of analyzing the vast amounts of multi-project software engineering data. For example, one promising area of work is automatic categorization of software systems, based on the source code of the systems, or keywords and comments associated with the source code [9, 10, 11]. In this paper, we

describe another significant analysis opportunity: utilizing multi-project data to improve the effectiveness of the reuse processes for component-based reuse. In particular, we describe the opportunity for real-time and continuous adaptation of the "best possible" component for a multitude of projects.

Suppose a component C is used by a multitude of software systems S_1 through S_n. Further, suppose that there are m implementations of C, C_1 through C_m, and each implementation has a slight variation in the interface provided, operating system supported, performance characteristic, and so forth. Hence, each of the n projects have to make a choice about which implementation of the component to use. Moreover, as time progresses, and the implementations of the components change, the projects have to keep making these choices. Current technologies do not provide much support for one project's choices influencing other projects, except through out-of-band communication or coordination among the projects.

Through Multi-Project Software Engineering, we can provide automation for several aspects of this reuse process: (1) we cluster related components together, e.g., by their usability or interface provided, (2) we rank the components by their popularity among S_1 through S_n, and (3) we can provide automatic substitution of components if and when a problem is discovered in one of the implementations. In the rest of this paper we give an overview of technologies used to provide such automation.

2. Storing and Utilizing Data

An essential component of Multi-Project Software Engineering is the ability to systematically collect and organize large amounts of data, from tens of thousands of software projects. This requires: (1) mechanisms for defining the data to be collected from each project, (2) systematic organization of the collected data, and (3) mechanisms for easily obtaining the data from each project.

For each of these questions, we learn from the experiences of the Open Source and Free Software communities that have demonstrated environments for collecting and organizing vast amounts of multi-project data, through the pioneering efforts such as the Open Source Development Network (OSDN) [15] and the Gnu software tools. Hence, similar to the OSDN, for each project we capture complete versioned source code trees, email discussion archives, bug report and their workflow, and documents associated with the project including web pages. We use a combination of the hierarchical file system and relational database to organize the large amounts of data.

Rather than collect such data *a posteriori*, we collect and organize such data *in situ*. A critical aspect of this is to collect data as a *side-effect* rather than as an *after-thought*. This implies the existence of a Multi-Project Software Engineering Environment (MSEE) that can easily accommodate the development effort of tens of thousands of projects. In the following, we briefly describe the architecture of one such MSEE, SourceShare [1, 2], with which we are most familiar. Other MSEE's (e.g., see [7]) have similar architecture.

SourceShare is a web-based service. Through the web interface, SourceShare provides capabilities to:

- Add a new software project to the collection
- Browse through existing projects, using various sorting orders like categories, software name, contact name, or date of submission.
- Search through the software projects, either through the source code, software descriptions, mailing list archives, or issues and bug reports.

When a user adds a new software project, SourceShare requires the user to input a set of information about the software, e.g., who were the authors of the software, some keywords, a brief software description and title, etc. SourceShare stores this information in an XML file associated with the project. It also instantiates a version control repository, a mailing list, and a bug tracking system for that software project. Henceforth, users of SourceShare can start working on the project using the version control repository for their source code management. As in the case of Open Source software, SourceShare requires that all decision making and discussions about the software project be carried out using the email discussion list associated with the project, thereby maintaining a history of project decision making.

General users of SourceShare are free to browse through the source code and mailing list discussion forums to get a better understanding of the software. If they find any problems or issues with any software, they can input such issues in the bug tracking system associated with that software.

Hence, an MSEE provides some important features enabled by the rapid advances in network, CPU, and disk capacities:

- maintain and make visible tens of thousands of software projects at the same time,
- systematically collect and organize fine-grained data on each project for source code versions, problem reports and their resolution, and project discussions,
- provide a uniform web-based interface to all information, and
- collect data as side-effect of normal project activities.

3. Multi-Project Analysis

In a multi-project environment, larger projects typically rely on components implemented as part of other, probably smaller projects. A potential problem in such a work environment, especially in an Open Source setting, is that projects may die and are no longer maintained by their developers. In the subsequent discussion, we use project D to denote such a project. If such a situation arises, projects that rely on a component implemented as part of project D need to find other components providing the same or a similar functionality.

Typically, if such a situation arises, the maintainers of all the projects relying on a component that was provided by project D are seeking for alternatives. Thus, the process of locating projects that provide a viable alternative is duplicated several times. Using multi-project analysis, it is possible to base ones decision on the decision made by other projects that have used the same project D in the past. That is, one can issue queries such as "*which other projects have been using components provided by project D, and which other projects did they use in order to substitute the components originally provided from D?*"

One challenge is that for different projects, different substitute projects may be adequate, hence we must rank the components according to their popularity or similarity to the original project D. In order to solve this problem, we can use the component rank system that we have presented in [8].

Another challenge is that the substitute component typically provides a different interface than that provided by the original component D. In this case, it is necessary to adapt the substitute component. After the first project, however, has switched already to that given substitute component, we can learn from this other project the steps that have been taken in order to adapt the substitute component and reuse this kind of adaptation code. In order to achieve this functionality, a technique such as provided by type-based adaptation [6] can be used.

3.1 Component Rank

There are two kinds of technology elements needed to find alternative components.

1. Find a set S_C of components which have similar functionality to the replaced component C in D.
2. Find the most reliable component in S_C.

To resolve these issues, we have developed a model called component rank [8]. In this model, the search space for the components is represented as a directed graph. Each node in the graph represents a component. Each edge shows a use relation from a component to another in the sense of a method invocation, instance variable access, and so on. The component rank for a component is then the sorted order of the component by its eigenvalue of the adjacent matrix of the graph. This ranking intuitively shows significance and reliability of the components in the search space, i.e., a component with many incoming use edges from higher ranked components has a higher rank. A component used by many other components inside the project or other projects have many incoming edges, and it will generally have a high rank.

There are many cases that a single component developed in a project is repeatedly duplicated and reused in later projects with slight modifications. In order to identify the duplication, we define syntactical similarity among components. Various mechanisms have been used to identify syntactical similarity among the source codes of components, such as code-clone detection, distance computation by *diff*, and various metrics-value computation (e.g., LOC or complexity). Very similar components are merged into a single node in the graph so that the effect of multiple duplication will be removed.

A component search system has been developed using this model. In this system, a Java class is a component, and the system has many features, such as keyword search, use relation trace among classes, various software metrics computation, code-clone detection, and so on.

To find the alternative components using the component rank system, the following process will be applied.

1. Find S_C in the search space.
   (a) Locate C in the search space. There are two possible ways to do this. We may browse the package hierarchy for the target C, or give keywords which will uniquely extract C.
   (b) Search all similar components S_C. The system has already collected and merged all syntactically similar components into a single node in the graph. So the collected components for C are the member of S_C. Also, functionally similar components are collected by the keyword search mechanism of the component search system. Unique names in library-call statements or in comments will be used as the keyword.
2. Compute the component ranks of all components in S_C, and pick up a high-valued component as reliable one. The system lists up the components in the sorted order of the component ranks, and the developer checks each component from the top until a satisfactory component is found.

3.2 Component Substitution

In the previous section, we have seen how a substitute component can be located on the basis of the knowledge of other projects. Once a substitute component has been located that provides the same functionality as the original component, it needs to be adapted. This is because it is unlikely that another implementation C_{subst} of the component C provides the same interface as the original component C_{orig}.

One option for the developer is to modify the implementation of C_{subst} to make it fit his needs. This approach, however, defeats the purpose of component-based development as it no longer allows the component C_{subst} to be maintained separately. Otherwise, the implementation of C_{subst} would have to modified whenever a new version made available. In order to avoid this problem, developers typically implement small wrappers that adapt the component in a way such that it provides the interface required by their application (i.e., that of C_{orig}).

Once, a project has already located a substitute component and the developers have already implemented the necessary wrappers in order to provide the interface of the original component, it would be more efficient if other projects could simply reuse these wrappers. This can be achieved by putting them into a shared repository where they can be queried for as information about the component interfaces they wrap, respectively provide. By doing so, this repository can be queried by other projects. Hence, adapters become first order objects and can be reused in a way similar to component implementations.

This kind of infrastructure is provided by type-based adaptation which we have presented in [6]. In addition to storing wrappers in a repository, type-based adaptation can only determine when adapters can be combined in order to provide more powerful adaptations. In fact, adapters can be combined when the interface provided by one adapter is the subtype of the interface required by another adapter. In certain situations, this relationship may be relaxed as we have shown in [5].

Type-based adaptation only requires the ability to identify the interfaces required by a project C_{orig} and that provided by a component C_{subst}. Both can be identified on the basis of project data available in the CVS repository as well as the project's inter-dependencies. By using this information and the adapter's stored in the repository, type-based adaptation can automate the adaptation process by deciding when a adapters are needed and how they are to be applied.

More importantly, type-based adaptation can determine when it is necessary to chain several existing wrappers to effect an adaptation that is more powerful than any one existing wrapper can do by itself. This ability to chain wrappers together greatly increases the power of the process and requires many fewer wrappers to be written by the programmers. As we have mentioned before, we only have to define rules on when two wrappers may be combined. In the simplest case, this is the case one wrapper provides the interface that can be used by another wrapper and hence, they may be combined.

4. Related Work

Mockus, Fielding and Herbsleb present a case-study about Open Source Software projects in [12]. They used email archives of source code change history and Bugzilla problem reports to analyze the overall community and development process such as the code contribution, problem reporting, code ownership, and code quality including defect density in the final programs and problem resolution capacity. However, they only used this data to compare the open source development methods by looking at the Apache and Mozilla projects with the traditional commercial one. Our approach, however, focuses on the knowledge surrounding multiple projects and to try to learn from these projects.

Another approach that takes modification and problem reports into account is presented by Fischer, Pinzger and Gall in [3]. They use this data to analyze the evolution of a given software project and to track and to identify hidden relations between different features of the software system. Unlike our approach, however, they only take a single software project into account and do not try to identify relationships between multiple different software projects.

Component rank was first inspired by the famous document search engine Google [4, 13]. Google collects various documents from the Internet, analyzes their link structures among the documents, and computes the significance of each document using similar algorithm as ours. For queries of keywords from the users, Google returns the lists of documents containing the keywords in decrease order of their significance.

Google mainly targets general documents such as HTML and PDF, but it also contains source code of software. So it could be used as the software component search. However Google generally returns less precise lists than the lists made by the component rank. This is because Google does not have any mechanism for the source code analysis such as use-relation analysis and similarity check done in our component rank computation.

5. Summary

In this paper, we have presented a novel idea that analyzes the modification and bug report data of multiple different projects. By analyzing this data, it is possible

to deduct knowledge used by the different projects that if shared among the project can lead to improved software engineering practices. A benefit that we have shown in this paper, is that projects can benefit from other projects that make use of the same or similar components.

By identifying projects that use the same component implementation C_i of a component C, it is possible to share knowledge surrounding the component across multiple projects. For instance, if one of the projects using C_i uses an updated version of this component or replaces it with another component providing the same kind of functionality, this knowledge can be shared across projects. Hence, it allows a component C_i to be replaced with a superior component C_j more quickly within a large number of projects. Such kind of knowledge is especially of importance when the component implementation C_i no longer fulfills the requirements of the project. In this case, it is necessary to locate a substitute component c' that is able to meet the projects demands. As we have seen, using our component rank system it is possible to determine the usability and flexibility of different software components and based on that knowledge to infer which component implementation C_j should be used to replace the original component C.

Another challenge that is typically encountered is that a substitute component may not provide exactly the same component as the original component used for a given project. In such a case it is necessary for a given component C_j to be adapted in order to meet the requirements of a given project (i.e., to provide the same interface as the original component C_i). Such adaptation, however, can be easily accomplished using type-based adaptation. As we have explained, type-based adaptation allows the reuse of code that has been implemented in order to adapt a substitute component C_j in order to meet the requirements of the originally used component C_i.

References

[1] J. Dinkelacker and P. Garg. Corporate Source: Applying open source concepts to a corporate environment. In *1st ICSE International Workshop on Open Source Software Engineering*, 2001.

[2] J. Dinkelacker, P. Garg, D. Nelson, and R. Miller. Progressive Open Source. In *Proceedings of the International Conference on Software Engineering*, Orlando, Florida, 2002.

[3] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, pages 90–99. IEEE, Nov. 2003.

[4] The Google website. `http://www.google.com`.

[5] T. Gschwind. Automated adaptation of component interfaces with type based adaptation. Technical Report TUV-1841-2003-14, Technische Universität Wien, Apr. 2003.

[6] T. Gschwind. Type Based Adaptation: An adaptation approach for dynamic distributed systems. In *Proceedings of the 3rd International Workshop on Software Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*, pages 130–143. Springer-Verlag, 2003.

[7] T. J. Halloran, W. L. Scherlis, and J. R. Erenkrantz. Beyond Code: Content management adn the open source development portal. In *3rd ICSE International Workshop on Open Source Software Engineering*, 2003.

[8] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component Rank: Relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering*, pages 14–24, 2003.

[9] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic Categorization for Evolvable Software Archive. In *International Workshop on Principles of Software Evolution*, pages 195–200, In conjunction with ESEC/FSE 2003, Helsinki, Finland, 2003.

[10] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. Automatic Categorization Tool for Open Software Repositories. In *Workshop on Open-Source in an Industrial Context*, In conjunction with OOPSLA 2003, Anaheim, CA, 2003.

[11] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue. On Automatic Categorization of Open Source Software. In *3rd Workshop on Open Source Software Engineering*, pages 79–83, In conjunction with ICSE 2003, Portland, OR, 2003.

[12] 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.

[13] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford Digital Library Technologies, Jan. 1998. `http://dbpubs.stanford.edu/pub/1999-66`.

[14] E. S. Raymond. *The Cathedral and the Bazaar*. O'Reilly, 1999.

[15] SourceForge.net. `http://sourceforge.net`.

Author Index

International Workshop on Mining Software Repositories

MSR 2004