A Dataset from Change History to Support Evaluation of Software Maintenance Tasks

Bogdan Dit, Andrew Holtzhauer, Denys Poshyvanyk
Computer Science Department
The College of William and Mary
Williamsburg, VA, USA
{bdit, asholtzh, denys}@cs.wm.edu

Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
Wichita, KS 67260-0083
kagdi@cs.wichita.edu

***Abstract*—Approaches that support software maintenance need to be evaluated and compared against existing ones, in order to demonstrate their usefulness in practice. However, oftentimes the lack of well-established sets of benchmarks leads to situations where these approaches are evaluated using different datasets, which results in biased comparisons. In this data paper we describe and make publicly available a set of benchmarks from six Java applications, which can be used in the evaluation of various software engineering (SE) tasks, such as feature location and impact analysis. These datasets consist of textual description of change requests, the locations in the source code where they were implemented, and execution traces. Four of the benchmarks were already used in several SE research papers, and two of them are new. In addition, we describe in detail the methodology used for generating these benchmarks and provide a suite of tools in order to encourage other researchers to validate our datasets and generate new benchmarks for other subject software systems. Our online appendix: http://www.cs.wm.edu/semeru/data/msr13/**

***Index Terms*—Generate Benchmarks, datasets, feature location, impact analysis**

I. Introduction

Techniques that support software maintenance tasks, are empirical by nature, thus demonstrating that a technique produces better results than existing techniques requires them to be evaluated and compared against one another. However, an unbiased evaluation and comparison is not always possible, due to the lack of well-established sets of benchmarks. This leads to situations where techniques are evaluated using different datasets, making their comparison problematic. For example, a recent survey on feature location techniques showed that out of the 60 papers that evaluated their proposed feature location techniques, only three of them (5%) used the same datasets in their evaluation [1]. In the other 95% of cases the techniques were compared using different datasets, which affects the fair comparison between techniques.

In this data paper, we want to address this problem and provide six datasets from four open-source Java applications, which can be used as benchmarks in the evaluation of various software maintenance tasks, such as feature location, impact analysis, developer recommendations, and traceability link recovery. Our datasets contain textual descriptions of change requests and locations in the source code where the change requests were addressed. These datasets are ideal for evaluating techniques based on Information Retrieval (IR). In addition, we provide execution traces that were collected based on the description of the change requests. These traces could be used in techniques that combine IR and dynamic information [2].

Among these six datasets that we make publicly available, four of them were already evaluated in a number of research papers related to feature location [3, 4, 5, 6, 7, 8], impact analysis [9], developer recommendations [10] and traceability link recovery [11]. The two remaining datasets (ArgoUML 0.24 and ArgoUML 0.26.2) are new.

In addition, we describe in detail the methodology used for generating these datasets from the historical data of the software systems (Section III), as well as any limitations associated with the process of generating this data (Section VI). We also provide a suite of Java tools that instantiate some steps of the methodology, which can be used to generate datasets for new software systems (Section IV).

By providing the methodology and tools, we offer other researches the possibility to verify our datasets and encourage them to generate new benchmarks for other software systems.

We refer the interested reader to visit our online appendix in order to get access to the datasets, the methodology, tools, and a detailed description of the data format.

II. Datasets

These datasets contain static, textual, and dynamic information about the software systems, which were generated by analyzing two primary sources of information: (i) issue tracking systems (ITSs) and (ii) source code repositories.

A. Glossary of Artifacts

Dataset (or benchmark): is a collection of artifacts derived from the ITS and source code repositories and is referred by the name and version of the system (e.g., jEdit 4.3).

Issue: is the generic term given to change requests, such as bug reports, feature requests, or any other type of tasks submitted to an ITS (e.g., Bugzilla, Trac, etc.)

IssueID: is the ID (i.e., numerical value, such as 123) of an issue, which is automatically assigned by the ITS.

GoldSet$_{\textbf{IssueID}}$: is the set of unique method names that were modified when the issue IssueID was implemented in the system. In other words, it contains the names of the methods

Table 1 Description of the datasets. The columns represent the dataset name (system and version number), the major releases corresponding to the interval for analyzing the SVN data, the number of issues, the type of execution traces (marked or full), the total number of gold set methods in the entire dataset, the number of lines of code, files and methods for the system used to build the corpus

Dataset	Period	Issues	Trace Type	# Gold Set Met.	KLOC	Files	Methods
ArgoUML 0.22	0.20-0.22	91	Full	701	149	1,439	11,000
ArgoUML 0.24	0.22-0.24	52	Full	357	155	1,480	11,464
ArgoUML 0.26.2	0.24-0.26.2	209	Full	1,560	186	1,752	14,597
JabRef 2.6	2.0-2.6	39	Full	280	74	579	4,607
jEdit 4.3	4.2-4.3	150	Marked	748	104	503	6,413
muCommander 0.8.5	0.8.0-0.8.5	92	Full	717	77	1,069	8,187

that were changed when a bug was fixed, or when a feature was added to the system. The method names in the gold set are fully qualified (i.e., they contain the package name, class name, method name and signature).

$\text{Trace}_{\text{IssueID}}$ (or Execution Trace for IssueID): represents an execution trace that was collected by exercising the scenario presented in the description of the issue IssueID. The execution trace is characterized by a list of methods that were executed when the user attempted to (i) reenact the steps that lead to the buggy behavior described in IssueID or (ii) exercise a feature described in IssueID.

Marked Trace: is a trace where the user has control over the beginning and the end of the trace recording process.

Full Trace: is an execution trace that records executed methods from the start of the application until the application is closed. Full traces usually capture more information than marked traces.

$\text{Query}_{\text{IssueID}}$: represents the textual description of the issue IssueID, and consists of the title and description of the IssueID.

Corpus: is a collection of textual documents (e.g., contents of files, classes or methods). For our datasets, we refer to a corpus as the collection of all the method contents for a particular version of the software system.

B. Description of the Datasets

The six datasets that we are making publicly available contain in total 633 issues, 633 execution traces and 4,363 gold set methods and are summarized in Table 1.

The first three datasets (ArgoUML0.22, 0.24 and 0.26.2) are generated from ArgoUML, a popular UML editor. The other three datasets (JabRef 2.6, jEdit 4.3 and muCommander 0.8.5) were generated from JabRef, a manager for BibTeX references, jEdit, a popular text-editor for programmers, and muCommander, a cross-platform file manager.

The columns from Table 1 are enumerated and described next, and exemplified on the first dataset. The first column represents the name and version of the dataset (e.g., ArgoUML 0.22), which was generated by analyzing the SVN commits of ArgoUML submitted between version 0.20 and 0.22 (see column 2). For this dataset, there were 91 issues identified (see column 3), which contain a total of 701 gold set methods (see column 5). The type of execution traces collected is *full* traces (see column 4). Version 0.22 of ArgoUML has 149 KLOC (lines of code) spreading across 1,439 files and 11,000 methods (see columns 6, 7 and 8 respectively).

III. Methodology for Generating the Datasets

This section describes the methodology used for generating the datasets. The steps are as follows:

A. Choose the Software System

The first step consists of choosing a Java software system (e.g., jEdit) with the following characteristics: (i) uses SVN as the source code repository, (ii) has an ITS that keeps track of the change requests, (iii) a subset of SVN log messages are referencing IssueIDs, and optionally, (iv) the system allows collecting execution traces (i.e., the system is not a library that would make it difficult for a user to interact with it in order to collect execution traces). The last requirement is optional, and is only needed for generating datasets that contain dynamic information in the form of execution traces.

Note that the choice of Java systems was restricted by the fact that our tools for (i) generating gold sets, (ii) generating the corpus, and (iii) collecting traces work only with Java systems.

B. Choosing the SVN Commits

Choose the period of time between two major releases for the system (e.g., jEdit v4.2 and jEdit v4.3). In the following, we will refer to the earlier version of the system as the *previous release* (e.g., jEdit version 4.2), and to the older version as the *current release* (e.g., jEdit version 4.3).

For each SVN commit submitted between the *previous* and *current release*, we analyzed its log message (see Section III.C) and its change set (see Section III.D).

C. Choosing the Issues

For each SVN Commit, its SVN log message was parsed in order to identify the IssueIDs. The subset of SVN commits that contained IssueIDs in their SVN log message (called *SVNCommitsMapped*) were mapped to the issue *IssueID* from the ITS. For example, if SVN commit #123 contained the log message "fix for bug #45678", the issue #45678 (from the ITS) was mapped to the SVN commit #123. We manually verified each mapping to ensure the correctness of the data and to discard SVN commits that contain numbers that do not represent IssueIDs (e.g., "Eliminated a small code duplication found in r10817", "[...] viewtopic.php?f=4&t=413"). In addition, we also included the cases where an IssueID was mapped to multiple SVN commits (i.e., the change request represented by the IssueID was implemented across multiple SVN commits).

D. Generating the Gold Sets

For each SVN commit from *SVNCommitsMapped* (e.g., #123), we analyzed its associated source code files. More specifically, the version of each modified file (e.g., #123) was compared against the previous version of the file (e.g., #122 or earlier) in order to identify the methods that were modified during the SVN commit. These methods are part of the gold set associated with the IssueID (e.g., #45678) the SVN commit is mapped to (i.e., #123). The details of the tool used for generating these gold sets are presented in Section IV.

E. Generating the Corpus

The corpus of the *current release* was generated using the *CorpusGenerator* tool (see Section IV), which parses all the Java files associated with that release and extracts as documents all the contents associated with a method (i.e., javadoc comments, modifies, type, name, signature and body).

F. Generating the Execution Traces

For each issue generated in Section III.C, we identified the candidates suitable for generating execution traces on the *current release*. We generated the execution traces by reproducing the scenario presented in the description of the issue. In some cases, the steps to reproduce the bug or feature are enumerated in a straightforward way, whereas in other cases these steps had to be inferred from the description (because they are not explicitly stated). Issues for which we could not collect an execution trace (i.e., the symptoms to reproduce the buggy behavior are not described or cannot be inferred) were discarded. The execution traces were collected using either the Java Platform Debugger Architecture (JPDA) or the Eclipse Test & Performance Tools Platform (TPTP). The traces collected with JPDA (e.g., for jEdit) did not contain any method signatures and they are marked traces. The traces collected using TPTP contained the method signatures and they are full traces. Section VI discusses the decision of choosing the *current release* for generating the execution traces.

G. Cleanup

Not all the issues and gold sets generated in the previous steps became part of the final dataset. Some of the artifacts that did not adhere to a set of standards were discarded. For example, we only kept issues for which their gold sets had at least one method in the corpus of methods, and at least one method in the execution trace.

Methods that appear in the gold set may not necessarily appear in the corpus, due to the inherent process of refactoring that a software system undergoes between two consecutive releases. For example, a method *foo.A.a()* that was modified in an SVN commit (e.g., #123), and appears in the gold set of issue #45678, may not necessarily appear in the corpus, if the system experienced refactorings, such as the method name was renamed, its signature was changed, the class name was renamed, the class was moved in other packages, or the method was deleted or merged with other methods. Our tools do not automatically keep track of all the changes to the fully qualified name of methods and this is left for future work.

In an initial attempt to address these limitations, we used a simple process, where we manually modified the fully qualified name from the gold set to reflect the name from the corpus. For example, if a large number of methods from the gold set (e.g., *foo.A.a()*, *foo.A.b()*, *foo.A.c()*, *foo.A.d()*, etc.) did not appear in the corpus because the class *foo.A* was renamed to *foo.ARenamed*, we manually renamed the methods in the gold set to *foo.ARenamed.a()*, *foo.ARenamed.b()*, and so on. This manual process was applied only on a handful of gold sets that were identified during quality control of ensuing that at least one gold set method appears in the corpus. We acknowledge that this anecdotal manual process should have been replaced with a more thorough automatic approach, one which keeps track of all the refactorings during two software releases, but this endeavor is left for future work.

IV. Tools

We provide the following suite of Java tools that could help researchers generate new datasets for other systems, by following the methodology described in Section III. In addition, we provide Matlab implementations for two IR techniques, namely VSM and LSI.

DownloadSVNCommits is a tool based on the SVNKit library, which extracts all the pertinent information related to the SVN commits between the specified *previous* and *current releases*: (i) the SVN log message (which will be parsed for issues) and (ii) the content of the files at SVN revision *N* and *N-1* (these files will be analyzed for extracting the gold set).

ConvertJPDATraces and **ConvertTPTPTraces** are two tools that extract the list of methods that were executed for each type of execution trace.

GoldSetGeneratorFromSVNCommits uses the Eclipse Abstract Syntax Tree (from Eclipse's Java Development Tools) to automatically generate a list of methods that were changed between two versions of a java file (i.e., the version associated with the current SVN commit and its previous version). The tool only takes into account semantic changes to the code, and does not add to the gold sets methods that experienced formatting changes (e.g., indentation, adding blank lines, formatting comments).

CorpusGenerator uses the same underlying technology as *GoldSetGeneratorFromSVNCommits* to generate a corpus consisting of all the methods of a software system. In addition, this tool can also generate corpora for software systems at class or file-level granularity.

CorpusPreprocessor preprocesses a corpus produced by *CorpusGenerator*, by eliminating non-literals, splitting identifiers, stop word removal and stemming.

CorpusConverter converts a preprocessed corpus generated by *CorpusPreprocessor* to a term by document matrix that can be used as input for IR techniques, such as VSM and LSI.

VSM and **LSI** are two Matlab scripts that use VSM and LSI to compute the similarities between a query and the methods of a system (i.e., the corpus).

V. Description of Schema

This section describes the format of the data. Each dataset contains the following files and folders:

GoldSets: a folder with files named *GoldSet[IssueID].txt*. Each file contains the gold set methods, one per line. A gold set method is the fully qualified name of a method (e.g., *foo.A.a(int)*).

Traces: a folder with files named *trace[IssueID].trcxml* (TPTP format) or *Trace[IssueID].log* (JPDA format). Each file represents an execution trace collected for issue *[IssueID]*. The online appendix contains more details about the trace format.

Queries: a folder where each issue *[IssueID]* has two files named *ShortDescription[IssueID].txt* (i.e., title) and *LongDescription[IssueID].txt* (i.e., the description).

listOf[IssueType]IssueIDs.txt: is a file containing the list of IssueIDs for the dataset, one per line. The [IssueType] represents the type (e.g., *bug*, *feature*, *patch*) that was assigned to the issue in the ITS. The IssueIDs correspond to the *[IssueIDs]* from file names from the *GoldSets*, *Traces* and *Queries* folders.

CorpusMethods-<dataset>.corpusRaw and **CorpusMethods-<dataset>-AfterSplitStopStem.txt**: are two files containing the un-preprocessed and preprocessed corpora respectively. Each line of these files is a document representing the content of a method.

CorpusMethods-< dataset >.mapping: is a file containing the fully qualified names of the methods that have a correspondence in the preprocessed corpus file (i.e., the method name from line *i* corresponds to the method on line *i* from the file *CorpusMethods-<dataset>-AfterSplitStopStem.txt*).

IssuesToSVNCommitMapping.txt: is a file containing the IssueID and the list of SVN commits that map to it.

VI. Limitations and Design Decisions

Some of the methods from the gold sets do not have a correspondence in the corpus. This is due to the methodology for generating the data and the refactoring process between two consecutive software releases (see Section III.G). In addition, the SVN commits that do not explicitly include in their log messages the IssueIDs they addressed (i.e., the log messages lack the link to the ITS), are not included in the dataset.

In our datasets, we do not exclude from the gold sets the methods that were modified at one point between two releases, but which due to subsequent refactorings did not appear in the these releases. We leave this information in the gold sets for researchers that might need it for some tasks that would not require a corpus for the evaluation. Moreover, the solution that requires minimum effort to bypass this discrepancy between the gold set methods and the methods corpus requires filtering the gold set methods from the results, as was done in all the approaches that used our datasets [3, 4, 5, 6, 7, 8, 9, 10, 11].

Due to the refactorings between two consecutive software releases, some methods may not appear in the *previous release* (e.g., if they were added or renamed) or the *current release* as well (e.g., if they were renamed). We chose the *current release* for generating the corpus and the execution traces because even though the methods that were changed in order to fix the bugs submitted between these releases have similar chances of being present in the *previous release* or *current release* (i.e., due to refactorings), the methods that were added in order to implement the features introduced in the *current release* have zero chance of being present in the *previous release* but have a very high chance of being present in the *current release*. Thus we used one release to capture both the added features and the locations of the methods responsible for the buggy behavior as described in the bug description. If other researchers would require the use of the *previous release* in their evaluation, they could generate the corpus for the *previous release* using the *CorpusGenerator* tool, and filter from the gold sets the methods that do not appear in that corpus.

The quality of the execution traces might have been impacted by the quality of the steps to reproduce. For some issues, the steps to reproduce the bug or feature are described in an unambiguous way, whereas in other cases the description is open to interpretation. Due to the stochastic nature of the process of manually collecting execution traces, other researchers could generate different traces.

Despite all these limitations that are inherent from the process of generating the data and from the quality of available sources of information, our datasets can be used to support various software maintenance tasks, such as feature location [3, 4, 5, 6, 7, 8], impact analysis [9], developer recommendations [10] and traceability link recovery [11].

Acknowledgment

This work is supported in part by the NSF CCF-0916260 and NSF CCF-1016868 awards. Any opinions, findings and conclusions expressed herein are the authors' and do not necessarily reflect those of the sponsors.

References

[1] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *JSEP,* pp. to appear, doi: 10.1002/smr.567, 2012.

[2] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich, "Feature Location via Information Retrieval based Filtering of a Single Scenario Execution Trace," in *ASE*, 2007, pp. 234-243.

[3] B. Dit, L. Guerrouj, D. Poshyvanyk, and G. Antoniol, "Can Better Identifier Splitting Techniques Help Feature Location?," in *ICPC*, 2011, pp. 11-20.

[4] B. Dit, M. Revelle, and D. Poshyvanyk, "Integrating Information Retrieval, Execution and Link Analysis Algorithms to Improve Feature Location in Software," *ESE,* vol. 18, pp. 277-309, 2013.

[5] L. R. Biggers, C. Bocovich, R. Capshaw, B. P. Eddy, L. H. Etzkorn, and N. A. Kraft, "Configuring Latent Dirichlet Allocation based Feature Location," *ESE,* pp. to appear, doi: 10.1007/s10664-012-9224-x, 2012.

[6] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to Effectively Use Topic Models for Software Engineering Tasks? An Approach based on Genetic Algorithms," in *ICSE,* 2013, pp. to appear

[7] S. Davies, M. Roper, and M. Wood, "Using Bug Report Similarity to Enhance Bug Localisation," in *WCRE*, 2012, pp. 125-134.

[8] L. R. Biggers and N. A. Kraft, "A Comparison of Stemming Algorithms for Text Retrieval Based Feature Location," http://software.eng.ua.edu/reports/SERG-2012-03, 2012.

[9] M. Gethers, B. Dit, H. Kagdi, and D. Poshyvanyk, "Integrated Impact Analysis for Managing Software Changes," in *ICSE*, 2012, pp. 430-440.

[10] M. Linares-Vasquez, H. Dang, K. Hossen, K. Kagdi, M. Gethers, and D. Poshyvanyk, "Triaging Incoming Change Requests: Bug or Commit History, or Code Authorship?," *ICSM*, 2012, pp. 451-460.

[11] N. Ali, Y.-G. Guéhéneuc, and G. Antoniol, "Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links," *TSE,* doi: 10.1109/TSE.2012.71, 2012.