2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)

A Dataset for Dynamic Discovery of Semantic Changes in Version Controlled Software Histories

Chenguang Zhu
University of Toronto
Toronto, Canada
czhu@cs.toronto.edu

Yi Li
University of Toronto
Toronto, Canada
liyi@cs.toronto.edu

Julia Rubin
University of British Columbia
Vancouver, Canada
mjulia@ece.ubc.ca

Marsha Chechik
University of Toronto
Toronto, Canada
chechik@cs.toronto.edu

***Abstract*—Over the last few years, researchers proposed several semantic history slicing approaches that identify the set of semantically-related commits implementing a particular software functionality. However, there is no comprehensive benchmark for evaluating these approaches, making it difficult to assess their capabilities.**

This paper presents a dataset of 81 semantic change data collected from 8 real-world projects. The dataset is created for benchmarking semantic history slicing techniques. We provide details on the data collection process and the storage format. We also discuss usage and possible extensions of the dataset.

***Keywords*-Semantic history slicing; change history; benchmark**

I. Introduction

Over the last few years, researchers proposed several automated approaches to support software developers in identifying code changes related to a particular high-level functionality [1], [2], [3], [4]. These approaches are collectively referred to as *semantic history slicing*. Semantic history slicing techniques have been applied to different software evolution tasks such as assisting developers in transferring functionalities between branches of a configuration management system [2], performing transformations on change histories [1], and producing focused and easy-to-merge pull requests [4].

Existing semantic history slicing tool implementations such as CSlicer [2] and Definer [3], accept as input the project source code, a selected range of its version history and a test suite acting as the slicing criterion. The test suite is assumed to be deterministic and exercising behaviors of a software functionality (or feature in most cases). Semantic history slicing aims at finding a sub-history that preserves the target functionality, i.e., producing a syntactically correct, compilable program that passes the original test suite. It is called a *semantic history slice* [4].

In the absence of a unified benchmark for semantic history slicing, researchers evaluate their techniques on data that they themselves collect. Such data is manually extracted from open source software repositories. For example, Li et al. [4] evaluated their tool, CSlicer, on data taken from CSlicer's own source repository and three other open source project repositories, namely, Hadoop [5], Elasticsearch [6], and Maven [7]. Definer, another state-of-the-art semantic history slicing tool [3], was evaluated on data collected from the Apache Commons project repositories [8].

The absence of a comprehensive dataset for semantic change data is partly due to the difficulty in obtaining well-documented functionalities that come with test cases. For example, to collect semantic change data, researchers need to manually browse a large number of software projects and thoroughly inspect their version-controlled histories to find well-organized projects with informative documentation. In addition, since all semantic history slicing techniques also require a test suite as the history slicing criterion, further efforts are needed to identify those functionalities that are accompanied by corresponding test suites.

Obtaining ground truth for such a dataset is also difficult. Ultimately, a semantic history slicing technique should produce the smallest possible slice that would still pass the tests. Manually determining the smallest semantic history slice that does not contain unnecessary commits is time-consuming and error-prone [4].

To unify evaluation targets for semantic history slicing techniques and provide a common ground for researchers to benchmark their tool implementations, we report on a dataset for evaluating semantic history slicing techniques in this paper. The dataset consists of 81 items of semantic change data collected from the repositories of 8 open source Java projects on GitHub.

The rest of this paper is organized as follows. In Sect. II, we describe the methodology used for collecting data. In Sect. III, we explain the storage mechanism of the dataset, and illustrate the data schema using an example. In Sect. IV, we give instructions on how to use the dataset for semantic history slicing. We suggest ways for extending the dataset in Sect. V and conclude in Sect. VI.

II. Dataset Creation

In this section, we describe the creation process of our dataset. Central to our dataset is the concept of a *semantic history slice* [4]. A semantic history slice is a sub-sequence of a change history that preserves the functionality of interest, which is defined by a set of test cases.

For each functionality, the dataset features an *1-minimal* semantic history slice as its ground truth. The concept of 1-minimality was used in the literature [3], [4], [9] as a standard

978-1-5386-1544-7/17 $31.00 © 2017 IEEE
DOI 10.1109/MSR.2017.49

for precise history slices the computation of which is also tractable in practice. It essentially means that no single commit in the computed history slice can be removed without breaking the functionality of interest.

Project Selection. We manually inspected repositories of various open source Java projects, focusing on a set of Apache projects which use JIRA [10] as their issue tracker. JIRA holds complete change log and release notes. For each release version, a number of functionalities are exhibited on the release notes page, categorized by developers as new features, improvements, bug fixes, tasks, etc.

Each functionality is assigned a unique ID, referred to as "issue key", which is associated with an issue report recording detailed information about the functionality. A JIRA issue key is a string with the format "ABC-123", where "ABC" stands for the name of the project containing this functionality, and "123" is a number for uniquely identifying a particular functionality. A JIRA issue key of a functionality is usually embedded in the log messages of the commits implementing it, making it easy to locate it in the history.

From a collection of Apache software projects, we selected those that are well-managed and accompanied by complete documentation and change logs on JIRA. The first and second authors independently evaluated 26 projects, inspected their issue trackers, and filtered out those that either had no change logs or had very few features documented in the release notes. Afterwards, they cross-checked the results and ended up with eight projects that both authors considered to be good.

Functionality Selection. For each of the selected projects, we determined the functionalities to be included in the dataset. All known semantic history slicing techniques need test cases as slicing criteria to define functionalities of interests. Therefore, when we created the dataset, we only considered the functionalities accompanied by a test suite, where the code of functionalities and the test cases are committed together. From the eight projects selected in the previous step, we inspected features for 33 release versions. There were 478 features in total, and we filtered out those that had no test cases associated with them, resulting in 81 features.

History Range Selection. The purpose of semantic history slicing is to locate the commits relevant to a functionality within an appropriate history segment. We selected input history segments directly from the original project histories to closely simulate the practical application scenarios. The range of the selected history should also cover all the relevant commits for the whole life cycle of a functionality. Thus, for each functionality selected in the previous step, we would like to determine a range of history that subsumes the entire development activities of the functionality. To do that, we manually inspected the change logs of the projects and searched for the JIRA issue keys of the functionalities in the project version histories. For each selected functionality, we determined a sequence of commits that have the corresponding key in their commit messages.

For a functionality F, we identified a sequence of commits

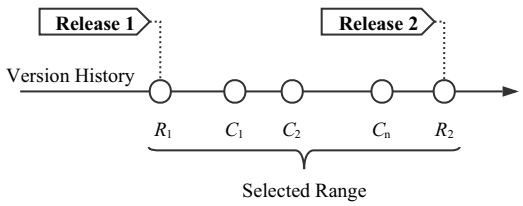

Fig. 1. History range selection between releases.

$C_1, C_2, \ldots, C_n$ which all contain the same JIRA issue key corresponding to F (Fig. 1). In Fig. 1, R_1 and R_2 are commits that are labeled as version releases either explicitly using release tags or implicitly using commit messages. R_1 is the closest release commit before the first appearance of the JIRA issue key of F, and R_2 is the closest release commit after the last appearance of the issue key. The history segment from R_1 (inclusive) to R_2 (exclusive) is selected as the input history for semantic slicing with respect to functionality F.

The rationale for selecting contiguous history segments between release commits is that every functionality is always finalized and published at some release version. Despite being conservative and potentially including irrelevant commits before and after the true functionality development period, we ensure that all potentially relevant commits are included in the selected range.

Obtaining 1-Minimal Semantic History Slices. We developed a data collection tool that applies a *delta debugging*-style partitioning algorithm [9] to produce 1-minimal semantic history slices (ground truth for semantic history slicing). Delta debugging repeatedly partitions the input space and opportunistically reduces the search space when the desired test behaviors are preserved in one of the partitions, until a minimal partition is reached. Zeller et al. [9] formally proved that the delta debugging algorithm is guaranteed to yield a 1-minimal partition that preserves the properties of interest.

Our tool is similar to delta debugging but is adapted to our application scenario to find an 1-minimal history slice that pass the functionality-exercising test cases. For each selected functionality in the dataset, we use its associated test suite and the selected history range as the input of the tool, running it to obtain a 1-minimal semantic history slice.

We carried out all the data processing jobs on a desktop computer running Linux with an Intel i7 3.4GHz processor and 16GB of RAM. The average time spent computing ground truth for a given functionality is approximately 1.5 hours. We divided the entire experimental set into 5 groups and ran them in parallel. We set a two-hours time limit on the tool execution for each functionality. Afterwards, we manually inspected the result, filtered out the functionalities that cause timeout, and extracted the meta-data from the remaining functionalities.

For each functionality in the dataset, we further verified that the semantic history slice returned by our tool is indeed 1-minimal in case there is flaw in the tool implementation. We did this by removing each commit individually from our computed semantic history slice, running the test suite and

expecting the tests to fail. The confirmed result of test suite demonstrated that the semantic history slice in our dataset is 1-minimal, as desired.

Dataset. Using the approach described above, we extracted 81 semantic change data from 8 open source Java projects found on GitHub. The final dataset covers a broad range of projects (previous collections contained four [2] and three [3]) and a significant number of data items (all previous collections [2], [3], [4] had less than 20). Our dataset is summarized in Table I.

The first six columns list aggregated statistics of the selected functionalities as well as the corresponding history segments and test suites. Columns "#F" and "#R" represent the number of selected functionalities and the number of release versions from which we gathered these functionalities, respectively. Column "Avg. Commits" lists the average length of history of developing a functionality. Columns "Avg. Files" and "Avg. LOC" show the average number of files changed and the average number of lines of code edited, during the development of the functionality, respectively. Column "Avg. Tests" shows the average number of tests in the associated test suite of the functionality. For example, we selected 17 functionalities from the `commons-lang` project, analyzed change histories for 4 release versions. The input history segments span over 365 commits, with about 211 files and 21.1 KLOC changed. The average number of test cases for a target functionality in `commons-lang` is about 4.14.

The last two columns contain metrics about the ground truth. Column "Avg. Slice" stands for the average size of the 1-minimal history slice of each functionality, expressed in terms of the number of commits. Column "Avg. Reduce" shows the average reduction rate in percentage. For each functionality, the *reduction rate* is the percentage of the unnecessary commits within the original input histories. The average reduction rate over all projects is about 87.9% which means that the 1-minimal history slices is on average 12.1% of the input histories.

III. Data Representation

We store the dataset in a GitHub repository (see https://github.com/Chenguang-Zhu/DoSC). It consists of the forked versions of the selected project repositories as persistent copies in case the original project histories are modified in the future. For each semantic change data in the dataset, we provide a file with its meta-data written in the YAML [11] format. Fig. 2 shows an example of such a file for the functionality CALCITE-1168[1]:

- *id* is the issue key of the functionality on JIRA – a unique identifier originally assigned by developers in the issue tracking system.
- *description* is the developers' description of the functionality, found on the JIRA release notes page.
- *project* designates the project in which the functionality belongs. The project's *name* and *url* are provided.

[1]https://issues.apache.org/jira/browse/CALCITE-1168

```
1  id: CALCITE-1168
2  description: Add DESCRIBE SCHEMA/DATABASE/TABLE/query
3  project:
4    name: Calcite
5    url: "https://github.com/apache/calcite"
6  issue url: "https://issues.apache.org/jira/browse/
       CALCITE-1168"
7  history start: "8eebfc6d"
8  history end: "aeb6bf14"
9  test suite:
10   - "SqlParserTest.testDescribeSchema"
11   - "SqlParserTest.testDescribeTable"
12   - "SqlParserTest.testDescribeStatement"
13 history slice:
14   - "a065200a"
15   - "da875a67"
```

Fig. 2. Meta-data of the functionality CALCITE-1168.

- *issue url* designates the link of the issue report of the functionality on JIRA. The issue report page contains detailed information and activity log of the functionality.
- *history start* specifies the starting point of the history segment where the functionality was developed. It is the SHA-1 ID of a release commit, which is the closest release version before the functionality was developed.
- *history end* specifies the ending point of the history segment where the functionality was developed. It is the SHA-1 ID of the closest release version after the functionality was developed.
- *test suite* designates the associated test suite of the functionality. The test suite exercises the behaviors of the functionality. All the test methods of the test suite are listed in this field.
- *history slice* designates the 1-minimal semantic history slice with respect to the functionality, i.e., the ground truth for semantic history slicing.

IV. Using the Dataset

In this section, we discuss the usage of our dataset.

Semantic History Slicing. The main motivation for creating the dataset is to provide evaluation targets for semantic history slicing techniques. Our dataset allows researchers to assess the capabilities of their history slicing tools and easily compare with other techniques.

To use the dataset as a benchmark for semantic history slicing, users need to follow the following steps:

1) Pick a functionality that they would like to analyze from the dataset. View its meta-data file.
2) Access the repository of the project via the link provided in the *project url* field of the meta-data.
3) Use the `git clone` command to clone the project repository to the user's local file system.
4) Extract the names of all test methods listed in the *test suite* field of the meta-data.
5) Use the extracted test cases and the history segment specified by the starting point (field *history start*) and the ending point (field *history end*) as the input on which to run the history slicing tool.
6) Compare the resulting semantic history slice with the 1-minimal ground truth we provide (field *history slice*).

TABLE I
OVERVIEW OF SOFTWARE PROJECTS IN THE DATASET

Project	#F	#R	Avg. Commits	Avg. Files	Avg. LOC	Avg. Tests	Avg. Slice	Avg. Reduce (%)
commons-lang	17	4	365.21	211.36	21112.64	4.14	44.86	88.97
calcite	19	7	89.83	332.67	31150.77	3.39	6.61	90.65
maven	11	6	82.09	183.09	7153.27	2.27	8.18	89.24
commons-compress	14	6	155	156.33	7172.67	5	17.33	89.01
flume	9	3	104.11	299.33	21355.56	4	20.22	79.82
pdfbox	5	3	203	188.4	10184	6.2	2	98.7
commons-configuration	3	2	117.33	254	54576	6	20.67	65.61
commons-net	3	2	205	188.33	7202.33	6.67	29	87.05
Overall	81	33	165.93	240.71	19833.14	4.08	18.53	87.92

Note: complete statistics on the dataset can be found at https://github.com/Chenguang-Zhu/DoSC

7) Repeat the steps 1-6 until the evaluation is sufficient.

Other Applications. In addition to semantic history slicing, our dataset can also be used for other software analysis tasks such as *dynamic feature location* [12], requiring only simple modifications to the above method.

Feature location aims to identify software components that implement a specific program functionality. The dynamic location techniques [13] monitor executions of some target feature and analyze runtime traces collected during the executions, to identify the set of related program entities for the feature. In our dataset, each target functionality is accompanied by a test suite capturing its behaviors and a set of essential commits implementing the functionality. The test suites provided in our dataset can be used to activate and gather execution traces for the corresponding features, while the set of essential commits can be mapped to relevant code entities to evaluate the effectiveness of the feature location techniques.

V. EXTENDING THE DATASET

An obvious way of extending the dataset is to analyze more repositories. In our current setting, this would require open source Java projects which use issue tracking systems for recording the functionalities and organizing the change logs. The essential requirement for this inclusion is that the projects have well-organized version controlled histories (so that there is a clear way of identifying a particular functionality) and have corresponding test cases.

The dataset can also be extended to include project histories containing known bugs and failed test cases that manifest the buggy behaviors. This extension would allow evaluation of *fault localization* techniques [14], [15] which determine a set of code edits in the change histories that may lead to program faults. Dynamic fault localization techniques analyze execution traces of passed and failed test runs, identify the most probably failure-inducing code changes, and sometimes also produce ranking of the potential culprits.

We provide a meta-data schema template as part of the dataset, and encourage other researchers to contribute to the dataset directly on GitHub. To obtain 1-minimal history slice, we implemented a data collection tool based on the delta debugging-style of history partition and made the tool publicly available. The tool is located in the same repository as the dataset itself.

VI. CONCLUSIONS

In this paper, we presented a dataset to benchmark semantic history slicing techniques. The dataset consists of meta-data of 81 functionalities covering a diverse range of software version histories from 8 software projects.

To the best of our knowledge, our dataset is the first to focus on supporting semantic history slicing research. It provides comprehensive and well-documented data for semantically-related changes in the context of version controlled histories. We believe that the dataset will provide insights on capabilities of tools and help advance the state-of-the-art in semantic history slicing research.

REFERENCES

[1] K. Muşlu, L. Swart, Y. Brun, and M. D. Ernst, "Development History Granularity Transformations," in *Proc. of ASE'15*, Lincoln, NE, USA, November 2015, pp. 697–702.

[2] Y. Li, J. Rubin, and M. Chechik, "Semantic Slicing of Software Version Histories," in *Proc. of ASE'15*, Lincoln, NE, USA, November 2015, pp. 686–696.

[3] Y. Li, C. Zhu, J. Rubin, and M. Chechik, "Precise Semantic History Slicing through Dynamic Delta Refinement," in *Proc. of ASE'16*, Singapore, Singapore, September 2016, pp. 495–506.

[4] ——, "Semantic Slicing of Software Version Histories," *IEEE Transactions on Software Engineering*, February 2017.

[5] How to Contribute to Hadoop Common. [Online]. Available: https://wiki.apache.org/hadoop/HowToContribute

[6] Elasticsearch: Distributed, Open Source Search and Analytics Engine. [Online]. Available: https://www.elastic.co/products/elasticsearch

[7] Apache Maven Project. [Online]. Available: https://maven.apache.org

[8] Apache Commons Project. [Online]. Available: http://commons.apache.org

[9] A. Zeller and R. Hildebrandt, "Simplifying and Isolating Failure-inducing Input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[10] JIRA Software. [Online]. Available: https://www.atlassian.com/software/jira

[11] YAML Ain't Markup Language. [Online]. Available: http://www.yaml.org/

[12] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering: Product Lines, Languages, and Conceptual Models*. Springer Berlin Heidelberg, 2013, pp. 29–58.

[13] R. Koschke and J. Quante, "On Dynamic Feature Location," in *Proc. of ASE'05*. ACM, 2005, pp. 86–95.

[14] L. Zhang, M. Kim, and S. Khurshid, "Localizing Failure-inducing Program Edits Based on Spectrum Information," in *Proc. of ICSM'11*, 2011, pp. 23–32.

[15] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of Test Information to Assist Fault Localization," in *Proc. of ICSE'02*. ACM, 2002, pp. 467–477.