Connecting Source Code Changes with Reasons

Namita Dave, Renan Peixoto da Silva, David Drobesh, Pragya Upreti, William Erdly, Hazeline U. Asuncion
School of Science, Technology, Engineering, & Mathematics
University of Washington, Bothell
Bothell, WA, USA 98011
e-mail: {namitad, rpeixoto, ddrobesh, pupreti, erdlyww, hazeline}@u.washington.edu

***Abstract*—Understanding the reasons behind software changes is a challenging task, as explanations are not always apparent or accessible. In addition, when third party consumers of software try to understand a change, it becomes even more difficult since they are not closely working with the code. To address these challenges, we propose a technique for explicitly connecting code changes with their reasons, referred to as Flexible Artifact Change and Traceability Support (FACTS). FACTS presents a holistic view of changes by (1) generating traceability links for code changes at different levels of abstraction and (2) tracing code changes to heterogeneously represented reasons. Our user experiment indicates that FACTS is useful in understanding code changes.**

Keywords-software evolution tools; software traceability; software maintenance.

I. INTRODUCTION

Changeability is one of the essential difficulties with software [1]. Developers know the reasons for changes they make in the source code. These reasons may be recorded, but they are not always explicitly connected to code changes. This situation is more difficult for Third Party Consumers of Software (TPCS) who wish to understand the reason for a specific code change but they lack access to developers who performed the change. We aim to cater to these TPCS.

There are different approaches to address the challenge of determining *what* source code changed between two versions (e.g., [2], [3]), but these generally provide only one level of granularity of changes (e.g., line, method only). This limited view of changes may be addressed by software maintenance techniques to connect code with other code [4]–[6]. More importantly, there are limited techniques in addressing *why* a change occurred. There are techniques that connect code to documentation [7][8], to assist with impact analysis [9], but these do not provide a holistic view of *past* changes. Our approach is also distinctive from other software traceability approaches [7][8][10] in that we focus on tracing code *changes* (i.e., not source code, but the change between two versions of software) to their reasons. Most closely related work connects code changes with reasons for change by using a document summarization technique [11], but this technique falls short in determining which documents to summarize.

We address these gaps with our technique, Flexible Artifact Change Traceability Support (FACTS). FACTS assists with understanding past changes by (1) explicitly connecting code changes (at different levels of abstraction) to heterogeneously represented reasons via a traceability link, (2) visualizing the generated connections in an understandable manner, and (3) providing a novel set of tools to support the technique.

We define **reasons** as insights into a code change, as these may not be complete explanations, but only clues that can be connected with other clues. Thus, we do not include descriptions of code changes. For example, move method or the addition of a Bridge pattern is not a reason for change, i.e., they do not attempt to answer ***why*** these changes were made. These are succinct summaries of ***what*** changed. Reasons may be extracted from a sprint task list, user stories, bug reports, commit descriptions, news, or other artifacts.

The contributions of this paper are as follows. First, FACTS generates traceability links based on any available artifacts present during software development. Second, it flexibly works with any software life cycle methods, including contexts that use minimal methods [12]. The only assumption of this technique is that a version control system is used [13][14] and that reasons for change are available. Finally, our user experiment indicates that our tracing technique is useful to TPCS. Our prior work involved collecting metrics for project management [15]. This paper elaborates on tracing code changes with reasons.

This paper is organized as follows. Section II provides a motivation behind our work and Section III briefly covers existing techniques. Section IV presents the FACTS approach. Section V covers the user experiment. We close the paper with avenues for future work.

II. MOTIVATION

TPCS who have two snapshots of source code generally ask the questions: Q1) What is the difference between these two versions? Q2) Why was this change made?

While there are numerous techniques that answer Q1 for software developers (see Section III), our approach caters to TPCS. For example, TPCS who see that a method moved from one source code file to another does not know *why* the move occurred, unlike developers who may have tacit knowledge about the change. TPCS include project managers (PMs), software engineers who build on top of another product, or Principal Investigators (PIs) of scientific software. The importance of Q1 & Q2 is described below.

Scenario 1: Scientific software development. In this context, software is used to solve a scientific problem and researchers generally have limited background in Computer Science (CS) or Software Engineering (SE)[16] A PI may

 ISBN: 978-1-61208-620-0

rely on Research Assistants (RAs) (e.g., graduate students or post-doctoral researchers) to write software [12] These RAs may also have limited background in CS or SE [12]. Before the RAs leave, the PI has at least two versions of the software: the original version given to the RAs and the modified version. At this point, the PI examines the source code to determine the source code changes (Q1) and if the changes satisfied the tasks given to the RAs (Q2).

Scenario 2: Distributed software development. Software development occurs in multiple locations, which may have different time zones. Even though a development team follows software engineering practices, a PM may still encounter difficulties understanding all the changes that occurred from the last official release to the current version. This task is more difficult in organizations required to conform to government regulations, since code quality is paramount [15] Here, PMs must quickly locate the changes (Q1) (i.e., view coarse-grained changes first and fine-grained details as needed) [15]. They also need to determine if they satisfy the deliverables for a release (Q2).

III. RELATED WORK

We discuss existing evolution techniques that FACTS leverages and compare FACTS to work in other areas.

A. Software Evolution

Determining code changes between two versions has been well studied, with techniques that compare changes at the line [3]class or package [2], or at the behavioral level [17]. Additional information can be provided by AST Diffs, such as location of the change or the type of change (e.g., add, move) [18]. FACTS leverages these types of tools and connects their output to provide coarse-grained (e.g., packages changed) and fine-grained (e.g., methods changed) changes (see "Connect CCD to CCD" in Section IV.D.3).

More recently, there are techniques that provide coarse-grained changes in the form of summary. ChangeScribe applies this summary as an automatically generated commit message [19], Another technique focuses on identifying structural code changes [20]. Again, these are focused on *what* changed, and may be folded into our framework.

B. Reasons for Code Change

A closely related work provides a reason for a code change by using natural language processing techniques to summarize documents such as bug reports and source code [11], [21]. This approach first creates a chain of documents, obtains summaries from those documents, and displays them with the code. This work falls short in determining *which* software artifacts to connect to source code changes, which we do with our technique. Another approach depends on embedded unique identifiers within commits, such as a bug or issue ID, to connect reasons to method changes [22], which we do not require. Another technique focuses on connecting code changes to requirements [23], while our approach takes a broader set of artifacts than requirements. Finally, classification of code changes based on categories of maintenance activities [24]. We aim to provide a richer set of reasons than categories of changes.

C. Software Traceability

Software traceability research aims to identify relationships between various software artifacts. We focus on traceability links between 1) code change to code change and 2) code change to documentation. A representative set of traced artifacts is in Table I along with their techniques and intended users. None of these techniques connect different granularities of code changes with heterogeneous artifacts, i.e., provide a holistic view of changes.

"Code changes to code changes" Traceability Links. There are various techniques for connecting related source code, including traditional information retrieval methods [4]–[6] and static analysis tools [24]. Our approach is not focused on locating related source code, but connecting code changes to other code changes.

"**Code Changes to Documentation**" **Traceability Links**. Techniques that connect source code to documentation [8], [10], [25] do not specify *why* a change occurred. There are also several traceability techniques that assist with impact analysis (*potential* view of change) (e.g., [9]). Our technique, meanwhile, focuses on the changes that occurred in the past and understanding them (*actual* view of changes). There are also techniques that create traceability links between code and other artifacts [26], [27] or between code, artifacts (in software repositories), and people [28]. However, these techniques do not connect code changes (at different levels of granularity) with heterogeneous artifacts.

D. Background on Random Forest

Random Forest classification involves a collection of several weak classifiers to create strong classifier [29]. A decision tree-learning algorithm on a subset of training data trains each of the weaker classifier. The random forest uses these multiple random trees classifications to vote on an overall classification for given input data set. We used Weka implementation of Random Forest with cosine similarities as features (see Section IV.D).

IV. APPROACH

FACTS is a systematic approach to tracing code changes with reasons, reasons with other reasons, and code changes to other code changes. Figure 1 shows the framework of our approach, with each layer performing the core steps in the approach. The blue dashed line in the middle distinguishes between code and reasons, and these steps can occur in parallel. We discuss from the bottom layer to the top, as each layer uses the output of the layer below it.

TABLE I. SAMPLE OF EXISITNG TRACEABILITY TECHNIQUES

Traced Artifacts	*Techniques*	*User*
issue reports & commit [30]	ChangeScribe with undersampling Random Forest	Developer
Issue reports & commit [13]	Diff comparison, user specification	Developer, PM
Code, document, and people [28]	Directed graph & regular language reachability	Developer
test code & source code [31]	Slicing & Coupling tool (SCOTCH), Latent Semantic Indexing	Developer

A. Extractor Layer

The extractor layer is responsible for extracting software change information. There are two types of extractors in this layer: extractors that extract reasons from various artifacts and extractors that obtain specific code versions. These extractors are built on top of other third party tools.

1) Extracting Reasons

One can find reasons for changes in a repository, database, or embedded within artifacts [15]. Explicitly stored reasons, such as bug databases, is straightforward to extract. One simply uses public Application Programming Interfaces (APIs) to perform such extraction [32]. If APIs are not available, but reasons are accessible via a website, then reasons can be scraped off the project site [33].

To extract embedded reasons, we used keywords such as "history" or "change notes". We also leverage the APIs of various tools (e.g.[34][35]). The process of extraction is tool-specific and can be artifact-specific (e.g., the process of extracting an issue and changes notes may be different even though both may be extracted from web pages). We built an extractor for each tool and artifact type.

2) Extracting Code Versions

The code extractors allow us to obtain snapshots of the source code for two versions to obtain changes in the software. We obtain these versions via an API provided by a code repository (e.g., JGit [32]).

B. Detector Layer

The detector layer identifies the changes within two versions of source code. This layer leverages various differencing techniques, such as line diff, class diff, and package diff. This layer provides insight into changes at different levels of granularity, providing a holistic view of changes. Moreover, one could also complement these structural diffs with behavioral diffs, such as SymDiff [17].

In addition to these techniques, we also use a technique for mining refactoring patterns [36]. Refactoring patterns provide yet another view into the code changes that may be more succinct than can be provided by existing diffs.

C. Transformer Layer

Once extraction finishes, the transformer layer transforms reasons and code changes into their uniform representations.

1) Transforming Reasons

Reasons, which are extracted from various artifacts, are represented as an **Artifact Change Description (ACD).** Here is a sample ACD:

```
id: acd1
timestamp: 5/1/10 3:20pm
sourceType: Release Notes
author: John Doe
path: https://github.com/apache/cassandra/
      blob/trunk/CHANGES.txt</path>
description: Remove pre-startup check for
open JMX port (CASSANDRA-12074)
```

Figure 2. Example of Artifact Change Description (ACD).

The extracted ACDs (issue database, commit records, requirements specifications, design documents), contain the following elements: id, path, description, sourceType, author, and timestamp. ID is an auto-generated identifier. Path specifies the location of change description on the local machine, on the local network, or on the Internet. The path may also point to a specific location within the artifact to support accessibility at different levels of granularity [15]. Description is a free-form text that describes the change. This may be a commit comment or a user story. SourceType specifies the type of artifact. If the extractor is a web scraper of a bug database, source type is "bug" or the specific name of the bug database. The author is an optional element, since this may not always be available. Timestamp, also optional, indicates the time when a change description was created.

2) Transforming Code Changes

Code Change Description (CCD) is a representation of source code or other implementation change. Figure 3 shows an example of a CCD entry.

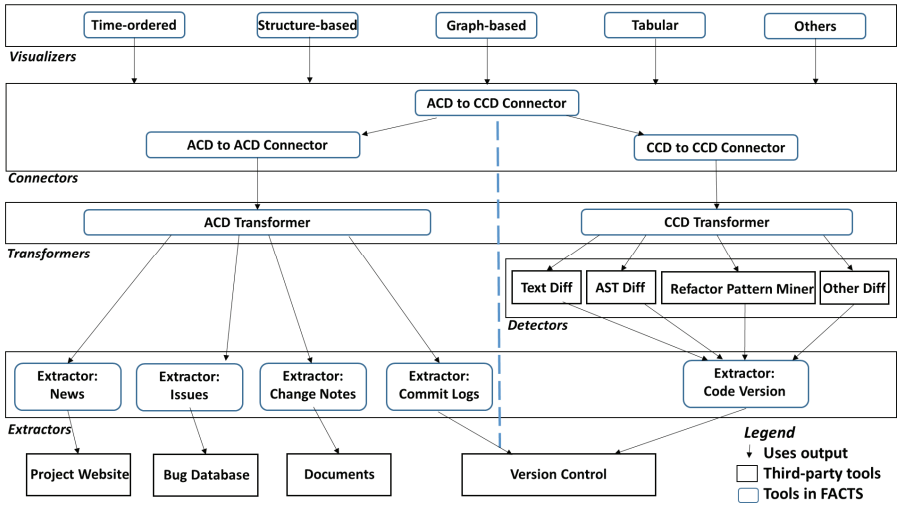

Figure 1. FACTS Framework.

```
id: ccd1
timestamp: 6/1/16 4:20pm
sourceType: package diff
sourcePath: acme/analysis/authDiff.txt
changeType: add
description: package auth
base-snapshot: 134a434f342347
compare-snapshot: 184a434f342959
```

Figure 3. Example of Code Change Description (CCD).

The extracted Code Change Description have the following elements: timestamp, the type of change (add, delete, edit), and the description of the change. The base- and compare-snapshot contain unique identifiers for the versions of the code that are compared, which may be commit identifiers, revision numbers, or release numbers. Description provides details about the changeType. The sourceType specifies the type of code differencing tool used. For example, the output of line diff has a sourceType "line diff". The output of the diff of a reverse engineered package diagrams has a sourceType "package diff".

D. Connector Layer

The connector layer is responsible for linking the extracted information. It connects information at the same level of abstraction (CCD to CCD and ACD to ACD) and across different level of abstraction (ACD to CCD).

Figure 4 shows an overview of the traceability links generated in FACTS. The blue dashed line in the middle separates the reasons for change, i.e., ACDs, from the code and other implementation change, i.e., CCDs. We use the Commit ACD as the focus of our traceability links to leverage the commit link already provided by a version control system. This allows us to bridge an abstraction gap between code and the explanations for change.

1) Pre-process

Before generating traceability links, we pre-process the data as follows. First, we remove all special characters. Then we split camel-cased words, words with underscore or hyphen, into individual words. Next, we transform all the words to lowercase. All these steps are performed prior to generating the traceability links.

For ACD-ACD traceability links, additional steps are performed. We lemmatize, remove stop words, and expand abbreviations. We expanded the abbreviations in order to normalize the vocabulary. We use the library JWKTL to lookup terms within the Wiktionary dump [37]. This utility translates abbreviations into a canonical, expanded and non-abbreviated word form.

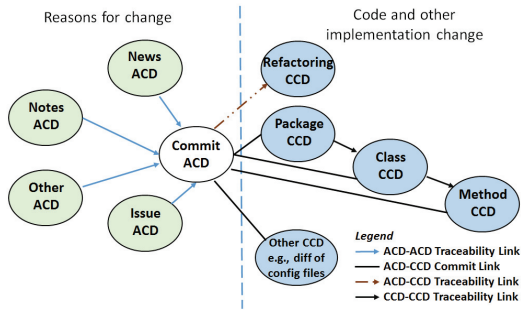

Figure 4: Traceability links in FACTS.

2) Same level of abstraction: Connect ACD to ACD

As we mentioned, reasons do not provide complete explanation for a change. Therefore, we connect different ACDs to better understand the reasons for change.

We can connect different ACDs using text similarity as description tag is free form text. We use a bag of words representation of documents and Random Forest implementation in Weka to create a prediction model capable of identifying traceability links.

Each classifier is built using the features described in Table II. Features F1-F4 are straight text comparisons. These features are insufficient, since they ignore important concepts that should be weighted more heavily. Thus, we identified the following groups of concepts that should receive more weights: features, architecture elements, tool operations (see F5-F10).

Sometimes, it is necessary to give more weight to terms that occur together in a document. This can be achieved by using N-grams with tf-idf (Term Frequency-Inverse Document Frequency) weighting (see F8-F10). N-grams can be calculated as follows: (1) the average N-gram distance between each term in the list of concepts and commit ACD; (2) the average distance between each term and other ACDs; (3) the N-gram distance between commit ACD and other ACDs. F8-F10 averages these three results.

Features F11-F13 are specific to connecting commit ACDs and issue ACDs. F11 checks if the commit was created between the issue report date and the update date of the same issue. F12 and F13 represent the number of days between the date of a commit and the issue report date, and the difference between the issue update date and the date that a commit was created, respectively.

3) Same level of abstraction: Connect CCD to CCD

Code changes are often provided at one level of granularity (e.g., line, class). Thus, understanding code change is limited at that level. To provide a holistic view of changes, CCDs at different levels of granularity can be connected. At the package level we trace changes from package CCDs to line diffs within packages, including changes to miscellaneous files (e.g., non-programming language programs, configuration files, scripts). For object-oriented languages, we trace package CCD to class CCD and to method CCD (see Figure 4). A traceability link is created by matching the fully qualified package names with the paths in the other diffs.

4) Different levels of abstraction: Connect CCD to ACD

Connecting code changes to reasons for changes is challenging since this requires bridging different levels of abstraction. The representation of concepts at the code level may be different from the representation of concepts at the reasons level. For example, the feature "elastic scalability" may have no matching terms in the source code.

To achieve this mapping, we leverage the developer action of committing code changes to the repository. When code changes are committed, we have a snapshot of the changes that occurred. In addition, developers generally enter a description for their changes.

 ISBN: 978-1-61208-620-0

TABLE II. ACD-ACD FEATURES LIST

Feature	Description
F1	number of terms in common
F2	cosine similarity
F3	maximum of cosine
F4	average of cosine
F5	weighted cosine for feature terms
F6	weighted cosine for architecture element terms
F7	weighted cosine for operation terms
F8	n-grams for feature terms
F9	n-grams for architecture terms
F10	n-grams for operation terms
F11	issueReportDate < commitDate < issueUpdate
F12	commitDate - issueReportDate
F13	issueUpdateDate - commitDate

Coarse-grained mapping: We leverage a commit ACD as a bridge between code and explanation. The connection between the commit ACD and the other ACDs are also provided by the technique we mentioned in the previous section. Thus, all the explanations for all the code changes within a given commit can be provided (see Figure 4). This connection is coarse-grained, at the level of a commit.

Finer-grained mapping: We can also achieve a more fine-grained connection between code changes and reasons by connecting refactoring pattern CCDs with commit ACD. Our approach to connecting mined refactorings with reasons for their occurrence consists of two main phases: *Training Phase* and *Test Phase*, and they share the following steps:

Step 1: Commit-Refactoring Pairing: Similar to creating ACD-ACD traceability links, we use Random Forest to build a model for connecting each commit ACD to refactoring CCD (output from Ref-Finder). This traceability link contains the following attributes: link (true or false), refactoring, commitID, commitDate, commitMessage, and calculated similarity features (discussed next).

Step 2: Similarity Analysis: After generating commit-refactoring links, we use similarity features (F1-F5) between refactorings and commit logs. Table III describes the features used, which were defined based on the information contained in refactoring changes and commit logs, including differences introduced in each commit, that could possibly enhance the process for growing decision trees within the classifier method. F1 checks if the refactoring activity is found in a commit log, by matching the full name of a package, class, method or parameter. F2 corresponds to the number of terms in common between a refactoring CCD and a commit ACD. F3 measures the cosine similarity between a refactoring CCD and a commit ACD. F4 and F5 are the maximum and the average of cosine similarity between commit ACD and refactoring CCD.

E. Visualizers Layer

This layer provides various visualizations, but we focus on two visualizations here: structure-based and graph-based visualizations.

Structure-based visualization allows users to view code changes to reasons. One can view coarse-grained changes (package view), then zoom-in to detailed changes (class and method view). Here users can see reasons for change.

The graph-based visualization allows users to view reasons with code changes. Users may select which concepts to view: features, architecture, operations. This displays the list of commits related to that particular concept. The user can view all code changes in a graph by selecting a commit. In addition, all the related reasons are displayed at the bottom, in this case news and issues ACDs.

F. Limitations of the approach

The limitations of the approach are as follows: dependence on a version control system, development of artifact/tool-specific extractors, and offline processing of artifacts. Since we focus our tracing technique on commit CCDs, we assume the usage of a version control system by a development team. This is not an unreasonable requirement as these tools are in widespread use in the software industry.

In order to obtain high quality reasons (i.e., minimal noise), the ACD extractors must be tool- and even artifact-specific. While this may require additional overhead in building extractors, this is a one-time overhead.

TABLE III. CCD-ACD FEATURES LIST

Feature	Description
F1	String of interest (yes/no)
F2	Number of terms in common
F3	Cosine similarity
F4	Maximum of cosine
F5	Average of cosine

Due to the sheer volume of change information obtained, we currently assume offline processing. This is also not unreasonable, as users can simply run the tool overnight and view the results the following day, similar to running a nightly build. It may be possible to parallelize the processing of all ACDs and CCDs, but this is currently beyond the scope of this paper.

Finally, while FACTS provides links to possible reasons, users have to read the linked documents (as automated techniques are not able to reach 100% accuracy). However, in the future, we plan to incorporate user feedback so that our classifier can learn from incorrectly classified links. In addition, while our approach is limited by the availability of artifacts, we believe it provides any available reasons to help uncover *why* a change was made.

V. EVALUATION

We conducted an experiment to assess (1) whether the FACTS approach is useful in understanding past code changes and (2) whether exposure to the FACTS tool support encourages developers to use it in their work. Answers to these Research Questions (RQ) were obtained using open-ended and multiple choice survey questions.

1) Method

Participants. In our study, we had a total of 36 participants, 22 of whom are Computer Science & Software Engineering graduate students and senior/junior level

 ISBN: 978-1-61208-620-0

undergraduate students and 14 industry users. Half of the participants were assigned to the control group and the other half to the treatment group. The software industry users (experience range from 5 months to 18 years) included technical leads, developers, managers, analysts, and one patent lawyer. Limited exposure or background with traceability tools was most common between both groups, despite their having had on average 5.75 years of industry or comparable experience using Eclipse IDE.

Dataset. The experiment was performed using the Apache Cassandra project, which presented a realistic challenge: very large codebase and a high velocity of code change. Specifically, we used v2.1.0 (200K LOC) and v3.0.1 (350K LOC). Cassandra also has multiple, concurrent branches, and frequent tagging and merges. Thus, our codebase selection has many realistic aspects as a sample for a moderate- or large-sized software product.

Environmental setup. All experiments were conducted on two virtual machines on which the users remotely connected. One virtual machine contains the Eclipse Diff Tools while the other machine contains the FACTS tool as an Eclipse Plugin. In both environments, both versions of the Cassandra project were available to enable users to examine the differences between the two versions.

Procedure. Subjects were asked to do the following: (1) fill out a pre-experiment online survey, (2) perform a task, and (3) fill out a post-experiment online survey. The pre-experiment online survey gathered demographic information about the participants (e.g., roles, length of experience) with their interest in traceability and software maintenance tools.

With regards to their task, they were given the hypothetical scenario of ACME Corp upgrading from v2.1.0 to v3.0.1 of Cassandra. The subject was informed that this was their first day working for ACME, and then instructed to learn two things: (A) Cassandra's high level design and (B) recent code changes. The subjects were given a maximum of two hours to explore and compare codebases, and directed to be ready to receive their initial code modification tasks assignment at the end of that brief self-orientation period. This instruction fits TPCS type of users. The "Treatment" group used FACTS tool support to understand code changes, while the "Control" group used traditional Eclipse IDE tools.

The post-experiment survey consisted of two parts: a quiz and a set of questions regarding their task. To test their understanding, they were given a timed "quiz" regarding code changes. Similar to an industry work environment, the questions in the quiz are very difficult.

To minimize bias, several study controls were used. First, random group assignment was used. Subjects did not know if they were in the control or treatment group. Second, we used anonymous user-codes to shield subject identity. Additional care was taken that our researcher and subjects were "double-blinded" of each other's identity. Finally, the questions in the quiz were generated by an undergraduate student using Eclipse Compare/Diff tools and were checked by researchers in our group. This shows the questions are fair, and subjects in the Control have an equal chance to answer the questions as the subjects in the Treatment group.

2) Results

With regards to RQ1, we found evidences that the FACTS tool support is useful in understanding past code changes. First, on average the subjects in the treatment group answered more correct questions than the control group (Control=1.5 vs. Treatment=2.28 correct answers). One explanation to why there is not a much wider gap between the two groups is that, due to the visualizations being in their prototype stages, there were some tool errors that came up during the study. Also, the earlier versions of the visualizations, which are the structure-based and tabular visualizations, were the only ones presented to the users.

Second indication that the approach is useful is that 71% of the treatment group (a statistically significant result) indicated that the tool, i.e., the approach, helped with their understanding of the codebases, citing ease of tracking changes along with time savings as benefits. One of these subjects, an industry user who is responsible for overseeing distributed development teams (up to 25 people in four time zones), said, "*It is difficult to keep track of what changes in the project. This tool would save time, and make the process more useful as well*."

With regards to RQ2, we also found that the subjects responded positively to using an improved version of the FACTS tool support in their work. When users in the treatment group were asked their level of interest in using the FACTS tool support to improve their current work process, 46% stated that they were "most interested" or "very interested", and 38% stated that they have "some interest".

3) Discussion

The sub-populations of industry subjects compared with student subjects showed some marked distinctions that are significant for the target population of TPCS. The industry developers showed a generally higher engagement and motivation rate, relative to the sub-population of student subjects, as measured by all of the following: (1) willingness to participate in future studies of FACTS traceability tools (64% industry vs. 50% students); (2) time you would invest in optimized versions of FACTS-IDE tool (13.7 hours industry vs. 10.96 hours student, when asked to base on 40 hour/week); (3) response to the question "Does the software tool help you link artifacts of change with actual code changes?" (Yes replies were 71% industry vs. 73% students). These results, when coupled with the participation rate (94% for industry subjects vs. 69% for student subjects) indicate a generally positive applicability trend of the FACTS tool to the actual population of developers.

VI. CONCLUSION

In this paper, we presented a novel framework for connecting code changes with reasons for change, with minimal requirements on the types of documents present or the software processes used. We systematically trace all the extracted changes and reasons within the same level of abstraction and across different levels of abstractions. Finally, the change information is visualized to assist with understanding reasons behind a code change. Our evaluation

 ISBN: 978-1-61208-620-0

indicates that FACTS is useful for understanding code changes, especially for industry subjects.

In the future, we plan to assess the accuracy of our tracing techniques. We also plan to further improve the user interface, improve the scalability of our tool, and conduct additional experiments and case studies with industry users.

ACKNOWLEDGEMENTS

We thank Karen Potts, Nathan Duncan, Wenbo Guo, Andrew Byland, Jonathan Featherston, Haihong Luo for developing visualizations and other tools in the FACTS framework, Steve Kay and Hoa Vo for assisting with user studies, and Delmar Davis for his input on existing techniques. R.P. da Silva is sponsored by CAPES and the Science Without Borders program. This work is based in part by the US National Science Foundation under Grant No. CCF 1218266 and ACI 1350724.

References

[1] F. P. Brooks Jr., "No Silver Bullet Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, pp. 10–19, Apr. 1987.

[2] L. Bendix and P. Emanuelsson, "Diff and Merge Support for Model Based Development," *Proc Int'l Workshop on Comparison and Versioning of Software Models*, 2008, pp. 31–34.

[3] G. Canfora, L. Cerulo, and M. Di Penta, "Ldiff: An Enhanced Line Differencing Tool," *Proc Int'l Conf on Software Engineering (ICSE)*, 2009, pp. 595–598.

[4] N. Dave, K. Potts, V. Dinh, and H. U. Asuncion, "Combining Association Mining with Topic Modeling to Discover More File Relationships," *Intl J. Adv. Softw.*, vol. 7, no. 3 & 4, pp. 539–550, 2014.

[5] R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia, "On the Equivalence of Information Retrieval Methods for Automated Traceability Link Recovery," *Proc Int'l Conf on Program Comprehension*, 2010, pp. 68–71.

[6] S. P. Reiss, "Semantics-based Code Search," *Proc ICSE*, 2009, pp. 243–253.

[7] G. Antoniol, G. Canfora, G. Casazza, and A. De Lucia, "Maintaining Traceability Links During Object-oriented Software Evolution," *Softw. Pract. Exper*, vol. 31, no. 4, pp. 331–355, 2001.

[8] K. M. Anderson, S. A. Sherba, and W. V. Lepthien, "Towards Large-scale Information Integration," *Proc ICSE*, 2002, pp. 524–534.

[9] S. Lehnert, Q. u a Farooq, and M. Riebisch, "Rule-Based Impact Analysis for Heterogeneous Software Artifacts," *Proc Conf Software Maintenance and Reengineering (CSMR)*, 2013.

[10] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *Trans. Softw. Eng. (TSE)*, vol. 28, no. 10, pp. 970–983, Oct. 2002.

[11] S. Rastkar, "Summarizing software artifacts," PhD Thesis, University of British Columbia, 2013.

[12] S. Ahalt *et al.*, "Water Science Software Institute: Agile and Open Source Scientific Software Development," *Comput. Sci. Eng.*, vol. 16, no. 3, pp. 18–26, 2014.

[13] "Git version control." [Online]. Available: https://git-scm.com/. [Accessed: 05-Feb-2018].

[14] "Subversion." [Online]. Available: https://subversion.apache.org/. [Accessed: 05-Feb-2018].

[15] H. U.Asuncion, M. Shonle, R. Porter, K. Potts, N. Duncan, and W. J. M. Jr, "Using Change Entries to Collect Software Project Information," *Proc Int'l Conf on Software Engineering & Knowledge Engineering (SEKE)*, 2013.

[16] B. Yasutake *et al.*, "Supporting Provenance in Climate Science Research," *Proc Int'l Conf on Info, Process, & Knowledge Mgmt (eKNOW)*, 2015.

[17] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebelo, "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs," in *Computer Aided Verification*, 2012, pp. 712–717.

[18] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine, "Dex: a semantic-graph differencing tool for studying changes in large code bases," *Proc Int'l Conf on Software Maintenance (ICSM)*, 2004, pp. 188–197.

[19] L. F. Cortes-Coy, M. Linares-Vasquez, J. Aponte, and D. Poshyvanyk, "On Automatically Generating Commit Messages via Summarization of Source Code Changes," *Int'l Working Conf on Source Code Analysis and Manipulation*, 2014, pp. 275–284.

[20] J. I. Maletic and M. L. Collard, "Supporting source code difference analysis," *Proc ICSM*, 2004, pp. 210–219.

[21] S. Rastkar and G. C. Murphy, "Why Did This Code Change?," *Proc ICSE*, 2013, pp. 1193–1196.

[22] A. S. Ami and S. Islam, "A Content Assist based Approach for Providing Rationale of Method Change for Object Oriented Programming," *Intl J. Info Eng. Electron. Bus.*, vol. 7, p. 49, 2015.

[23] E. B. Charrada, A. Koziolek, and M. Glinz, "Identifying outdated requirements based on source code changes," *Proc Int'l Requirements Engineering Conf*, 2012, pp. 61–70.

[24] M. Sharp and A. Rountev, "Static Analysis of Object References in RMI-Based Java Software," *TSE*, vol. 32, no. 9, pp. 664–681, 2006.

[25] X. Chen and J. Grundy, "Improving Automated Documentation to Code Traceability by Combining Retrieval Techniques," *Proc Int'l Conf on Automated Software Engineering*, 2011, pp. 223–232.

[26] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans Softw Eng*, vol. 31, no. 6, pp. 446–465, 2005.

[27] M. Kersten and G. C. Murphy, "Mylar: A Degree-of-interest Model for IDEs," *Proc Int'l Conf on Aspect-oriented Software Development*, 2005, pp. 159–168.

[28] A. Begel, Y. P. Khoo, and T. Zimmermann, "Codebook: Discovering and Exploiting Relationships in Software Repositories," *Proc ICSE*, 2010, pp. 125–134.

[29] L. Breiman, "Random Forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, Oct. 2001.

[30] T. D. B. Le, M. Linares-Vasquez, D. Lo, and D. Poshyvanyk, "RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information," *Proc Int'l Conf on Program Comprehension (ICPC)*, 2015.

[31] A. Qusef, G. Bavota, R. Oliveto, A. D. Lucia, and D. Binkley, "SCOTCH: Test-to-code traceability using slicing and conceptual coupling," *Proc ICSM*, 2011, pp. 63–72.

[32] "JGit." [Online]. Available: https://eclipse.org/jgit/. [Accessed: 05-Feb-2018].

[33] "jsoup: Java HTML Parser." [Online]. Available: https://jsoup.org/. [Accessed: 05-Feb-2018].

[34] "Excel Services REST API." [Online]. Available: https://docs.microsoft.com/en-us/sharepoint/dev/general-development/excel-services-rest-api. [Accessed: 05-Feb-2018].

[35] A-PDF, "A-PDF Text Extractor," Aug-2016. [Online]. Available: http://www.a-pdf.com/text/. [Accessed: 05-Feb-2018].

[36] M. Kim, M. Gee, A. Loh, and N. Rachatasumrit, "Ref-Finder: A Refactoring Reconstruction Tool Based on Logic Query Templates," *Proc Int'l Symposium on Foundations of Software Engineering*, 2010, pp. 371–372.

[37] "Java Wiktionary Library." [Online]. Available: https://dkpro.github.io/dkpro-jwktl/. [Accessed: 05-Feb-2018].

 ISBN: 978-1-61208-620-0