Investigating completeness and consistency of links between issues and commits

by

Casey Albert Thompson

B.S. Computer Science, University of California Irvine, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES

(Computer Science)

The University of British Columbia

(Vancouver)

August 2017

© Casey Albert Thompson, 2017
Abstract

Software developers use commits to track source code changes made to a project, and to allow multiple developers to make changes simultaneously. To ensure that the commits can be traced to the issues that describe the work to be performed, developers typically add the identifier of the issue to the commit message to link commits to issues. However, developers are not infallible and not all desirable links are captured manually. To help find and improve links that have been manually specified, several techniques have been created. Although many software engineering tools, like defect predictors, depend on the links between commits and issues, there is currently no way to assess the quality of existing links. To provide a means of assessing the quality of links, I propose two quality attributes: completeness and consistency. Completeness measures whether all appropriate commits link to an issue, and consistency measures whether commits are linked to the most specific issue. I applied these quality attributes to assess a number of existing link techniques and found that existing techniques to link commits to issues lack both completeness and consistency in the links that they created. To enable researchers to better assess their techniques, I built a dataset that improves the link data for two open source projects. In addition, I provide an analysis of information in issue repositories in the form of relationships between issues that might help improve existing link augmentation techniques.
Lay Summary

Software developers use commits to keep track of changes they make to a software project. To ensure that the commits can be traced to the issues that describe the work to be performed, developers manually create links between the commits and the issues. However, developers sometimes forget to manually create the links. To help find and improve links, several techniques have been created. Although many software engineering assistance tools depend on the links, there is currently no way to assess the quality of created links. To evaluate techniques that create links, I identify a way to assess the quality of the created links. To enable future researchers to better assess their techniques, I built a dataset that improves the link data. I finish with an analysis of issues-to-issue relationships that might help improve existing linking techniques.
Preface

All of the work presented in this thesis was conducted in the Software Practices Lab at the University of British Columbia, Point Grey campus.

The development of the dataset in Chapter 4 was done by A. Marques, and myself. A. Marques donated days as the second researcher to double-check the dataset that was dependably developed.

Chapter 5 of the research presented in this thesis has been previously published in the article:

# Table of Contents

Abstract ........................................... ii

Lay Summary ........................................ iii

Preface ................................................ iv

Table of Contents ....................................... v

List of Tables ......................................... viii

List of Figures ......................................... xi

Acknowledgements ..................................... xii

1 Introduction ......................................... 1
   1.1 Contributions ..................................... 3
   1.2 Roadmap ......................................... 4

2 Background and Related Work ....................... 5
   2.1 Background of Software Process Data ............. 5
      2.1.1 Version Control Systems ..................... 5
      2.1.2 Issue Repositories ............................ 6
      2.1.3 Issue Relationships .......................... 9
   2.2 Existing Techniques to Link Commits to Issues .. 10
      2.2.1 Manual Linking by Developers ............... 10
      2.2.2 Traditional Heuristics ...................... 11
2.2.3 State-of-the-Art Techniques ............................................. 12

2.3 Research in Understanding Issue Repositories ..................... 13
  2.3.1 Research that Studies Issue Tracking Systems ................... 14
  2.3.2 Approaches that Target Issue Tracking Systems ................. 14
  2.3.3 Software Process Work .................................................. 15

3 Assessing the Quality of State-of-the-Art Techniques ............... 16
  3.1 Completeness and Consistency ............................................ 17
    3.1.1 Completeness .......................................................... 17
    3.1.2 Consistency ............................................................ 19
  3.2 Performance of State-of-the-Art ....................................... 20
    3.2.1 Identifying Projects .................................................. 21
    3.2.2 Selecting Projects .................................................... 21
    3.2.3 Technique Modifications ............................................ 22
  3.3 Comparison of State-of-the-Art Techniques ........................... 24
    3.3.1 Completeness .......................................................... 24
    3.3.2 Consistency ............................................................ 25
    3.3.3 Summary of Quality Assessment .................................... 29

4 Creating a Complete and Consistent Dataset .......................... 30
  4.1 Existing Datasets ............................................................ 30
  4.2 Dataset Creation Process ................................................ 31
    4.2.1 Projects Selected ..................................................... 32
  4.3 Link Identification ........................................................ 34
    4.3.1 Preprocessing Dataset ............................................... 34
    4.3.2 Identification of Links .............................................. 35
    4.3.3 Dataset Validity ...................................................... 37
  4.4 Dataset ....................................................................... 38
    4.4.1 Summary ............................................................... 39

5 Semantic Relationships between Issues ................................. 41
  5.1 Qualitative Study .......................................................... 42
    5.1.1 Coding Process ........................................................ 42
    5.1.2 Codes ................................................................. 43
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1.3</td>
<td>Threats to Validity</td>
<td>46</td>
</tr>
<tr>
<td>5.1.4</td>
<td>Results</td>
<td>46</td>
</tr>
<tr>
<td>5.2</td>
<td>Summary</td>
<td>47</td>
</tr>
<tr>
<td>6</td>
<td>Discussion and Future Work</td>
<td>48</td>
</tr>
<tr>
<td>6.1</td>
<td>Improving Links of Commits to Issues</td>
<td>48</td>
</tr>
<tr>
<td>6.2</td>
<td>Techniques to Categorize Issue Relationships</td>
<td>49</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Algorithm</td>
<td>49</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>54</td>
</tr>
<tr>
<td></td>
<td>Bibliography</td>
<td>56</td>
</tr>
<tr>
<td>A</td>
<td>Links examined for consistency</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Tables

Table 2.1 Relationships instances ........................................ 10
Table 3.1 Statistics of the projects selected .............................. 22
Table 3.2 The number of commits and issue in 2016, and % completeness of manually stated links by developers .................. 23
Table 3.3 The % completeness for existing work and % improvement (Improv.) of linking relationship completeness of each state-of-the-art technique on selected projects. ......................... 25
Table 3.4 Relationships instances in each project that have an issue linked to a commit. Bold numbers are relationships that represent work breakdown in the issue repository ...................................... 26
Table 3.5 Results of manually examining consistency of up to five links from each project. ................................................. 28
Table 4.1 Commit and issue information for each project. ............... 32
Table 4.2 Comparison of relationship types for the window selected compared to the issue repository as a whole for both Connect and Sonarqube ............................................................... 33
Table 4.3 Results of linking process in identifying links commits to issues .......................................................... 38
Table 5.1 Codes occurring in each repository .............................. 46
Table 6.1 Example of transformation using a restricted language. ....... 50
Table A.1 Links examined for consistency evaluation in Chapter 3. No changes to the links were made by Phantom or Loaner.
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>The dotted line is a recovered link between issue and commit. The solid line is an existing relationship.</td>
<td>2</td>
</tr>
<tr>
<td>2.1</td>
<td>An example of a commit with the link CONN-1190 (highlighted in yellow) manually inserted by developers.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Gateway-3941 an issue from Jira issue repository for the connect open source project</td>
<td>8</td>
</tr>
<tr>
<td>3.1</td>
<td>Examples of complete and consistent linking relationships for two projects. Circles are commits in a version control system. Squares are issues in an issue repository. Dotted lines are relationships between issues. Solid lines are links between issues and commits.</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>Codes developed through open coding.</td>
<td>43</td>
</tr>
<tr>
<td>6.1</td>
<td>Example of categorizing with a supervised machine learning algorithm that learned to label issue pairs as check validity if the child issue the contains the word <code>test</code>.</td>
<td>52</td>
</tr>
</tbody>
</table>
Acknowledgements

In the institutional inauguration, Susan E. Sim and Rosalva Gallardo Valencia were instrumental in instructing an immature yet inquisitive initiate in investigations. Invariably, I’m infinitely indebted.

Cherished and charming champion called Ashely V. Port constantly contributed caring criticism. Ashley is a companion who cheerfully contributed to Casey’s commendable caliber and constantly comforted when consolation commanded.

An appreciative Albert acknowledges an accommodating advisor, Gail C. Murphy. Always assiduous at all advisory aspects, and an apodictic advocate at assisting in analytical areas.

Time to thank the Thompsons; through the turbulent times their therapeutic talk took tolerance and thriftiness. They taught that training takes toughness and technique to terminate tasks.

Steadfast people! Language sometimes packs lashes. Smiling pleased, let me script pleasant letters for Software Practice Labs splendid pals, lifelong. So passionate, leaving a smart powerful lab. Some premium last words, Reid Holmes showered positive learning, and simply put laudable Elisa Baniassad shared practical lessons.
At all who affectionately assisted my achievements. Alas advancement!
Chapter 1

Introduction

To assist developers in creating complex software, a number of software development tools have been created, including traceability [21], defect prediction [27] and time to completion [15]. These tools rely on underlying software process data, including issues in an issue repository, source code changes (i.e., commits) in a version control system, and links between commits and issues. A link from a commit to an issue indicates which artifacts—source code, documentation, and others—were changed in order to complete the work or defect described in an issue. Sometimes, the work required to complete an issue is split over multiple commits. It is possible that a commit can link to multiple issues, and an issue can link to multiple commits.

The state-of-practice in linking consists of developers manually stating the id of the issue being addressed in commit messages of a commit. Developers also use tool support such as Mylyn\(^1\) or Jira add-ons\(^2\) to link commits to issues. The state-of-practice does not link all commits and issues, Bird and colleagues analyzed seven different projects and found on average 63\% of fixed the issues in the projects they examined are not linked to a commit [8]. State-of-the-art techniques in linking commits to issues try to improve state-of-practice using the existing linking relationship to identify and link commits and issues that are not linked.

Links between commits and issues are used in a number of tools that do de-

\(^1\)www.eclipse.org/mylyn verified 13 July 2017
\(^2\)Git Integration for JIRA, Commit Policy for JIRA, Jenkins Plugin
Existing technique to recovering link

The correct link for the commit

Figure 1.1: The dotted line is a recovered link between issue and commit. The solid line is an existing relationship.

(a) Existing technique to recovering link

(b) The correct link for the commit

The goal of this thesis is to improve the quality of software process data in the form of links between commits and issues. In order to improve the quality, there needs to be a means to assess quality. However, there does not exist a standardized approach to such assessment in the community. To provide a means of assessment, I introduce the quality attributes of completeness and consistency. I define completeness as the extent to which all commits should link to an issue in the issue repository. I define consistency as the extent to which commits should link to the most relevant issues. I show that existing techniques that aim to improve links between commits and issues do not fully address the attributes of completeness and consistency (see Chapter 3).

Given that existing data, drawn from open source projects, does not provide high levels of completeness or consistency, I create a dataset that improves the underlying software process data for two open source projects (see Chapter 4).
This dataset is created manually by identifying all the issues that should be linked to a commit. The dataset provides a means for researchers to test techniques for improving links between commits and issues against a well-formed dataset.

Given that existing state-of-the-art techniques do not create fully complete and consistent links, there is room to improve these techniques. One kind of data that has not yet been considered in these techniques is the structure that exists between issues in issue repositories in the form of relationships between issues. Developers use relationships between issues to hold information about how functionality in a system is related or how work is broken down. Using relationships in issue repositories can be challenging because different kinds of relationships are used for different projects, even when the same underlying issue repository technology is used. For instance, an Apache developer related when discussing issue relationships:

\[\text{it’s reasonable to use is related to. Or probably we can make it is required by, but I’m not so sure}^{3}\]

Ambiguity among the meanings of relationship types can lead developers to incorrectly label relationships. To explore the kinds of relationships developers use, I preformed a study of the issues in an issue repository to reveal underlying meanings of issue relationships by categorizing relationship types in three issue repositories. The study focused on work breakdown relationships, as those were the most prevalent among all three issue repositories. The study found there are indeed underlying categories of meaning for work breakdown relationships (see Chapter 5). In future work, researches might build on theses categories to help improve the completeness and consistency of the links between commits and issues.

1.1 Contributions

This thesis makes three contributions:

1. Existing techniques to link commits to issues are shown to be incomplete and inconsistent. Although a number of techniques have been proposed to improve links between commits and issues in a project, these techniques

\[^{3}\text{From personal correspondence. Emphasis added.}\]
may still not produce as accurate a result as needed for software engineering
techniques, like defect prediction, that build upon the information.

2. To help support the development and evaluation of new techniques to im-
prove link data between commits and issues, I created a dataset consisting of
a portion of two open source projects. The creation of the dataset followed a
process to make the dataset as complete and consistent links as possible.

3. The kinds of relationships found in issue repositories is investigated, with an
identification of the meaning of work breakdown relationships and a method
to semantically categorize work breakdown relationships.

1.2 Roadmap

I begin by describing background about how developers use version control sys-
tems and issue repositories, followed by related work on techniques to link com-
mits to issues and earlier efforts in characterizing issue repositories (Chapter 2). I
then present a definition of two quality attributes completeness and consistency and
then assess existing techniques with respect to those attributes (Chapter 3). Next, I
describe the process necessary to create a dataset, followed by the creation of the
dataset (Chapter 4). I then describe a qualitative study to understand and categorize
the relationships that exist between issues (Chapter 5). I also discuss how a better
understanding of relationships can improve approaches to link commits to issues
(Chapter 6) before concluding (Chapter 7).
Chapter 2

Background and Related Work

This chapter provides background on the software process data considered in this thesis and reviews related work about the structure and contents of issue repositories. The latter is relevant to the use of structure in issue repositories that is proposed as a means of improving software process data.

2.1 Background of Software Process Data

The software process data examined in this thesis comes from two sources, version control systems and issue repositories. A version control system contains changes made to a project by developers. An issue repository contains issues used to track the many parts being worked on in a project. In addition to issue repositories, I examine the issue relationships and how they are used by developers.

2.1.1 Version Control Systems

Developers use version control systems to track source code changes made to a project and to allow multiple developers to make changes at the same time. Many different types of version control systems (VCS) exist, such as git\(^1\), svn\(^2\), and cvs\(^3\). Principles, such as whether there is a master repository\(^{[53]}\) or whether the VCS is distributed as in git, despite being based on different principles, all have the central

---

\(^1\)git-scm.com verified 13 July 2017  
\(^2\)subversion.apache.org verified 13 July 2017  
\(^3\)savannah.nongnu.org/projects/cvs verified 13 July 2017
goal of maintaining a history of all changes to source code. Each individual change is called a commit. In addition each comment has meta-data, such as the author of the change, timestamp and a textual message about the commit. In this thesis, I use GitHub\(^4\) as the forge to study to find open source projects, as GitHub offers simple search features to identify open source projects that meet desired specifications and all projects hosted on GitHub use git.

Figure 2.1 is an example of a commit in git for the open source project Connect\(^5\). The top of Figure 2.1 (1) lists all of the commits in git for the project. The bottom half provides details for the specific commit that is highlighted in the list of all commits. The subject of the commit (2) contains a short description of the reason for the commit. Underneath the subject (3) is the person who is considered the author of the commit and the date the commit was made. The author can manually, or sometimes through tool support add information about which issue the commit addresses. In this commit, the commit message identifies that the issue “CONN-1190” is addressed. On the right side (4) is the SHA that is the result of using a hash algorithm on the changes in the commit. The SHA is a unique identifier used by the VCS to verify that the data is not corrupt and by developers to uniquely identify the commit. The full description of the commit includes the short description and other text that the developer wants to include; this description is called the commit message (5). The files that are changed in the commit are listed in (6): these are any files that have modifications, renaming, additions, or deletions. The bottom shows changes that occur in a file (7): the changes are represented by the lines added and/or deleted and are referred to as a diff.

### 2.1.2 Issue Repositories

Developers use issue repositories to keep track of work and communicate with others \([6]\). Issue repositories contain multiple types of issues; each type typically contains different information. This information can include what a system should do, who works on different parts of a system, what bugs are being fixed, and more \([5, 24, 33]\).

\(^4\)github.com, verified 13 July 2017  
\(^5\)connectopensource.com, Connect is a project supporting health information exchange, verified 13 July 2017
Figure 2.1: An example of a commit with the link CONN-1190 (highlighted in yellow) manually inserted by developers.

There are many different implementations of issue repositories, some focused on identifying bugs and some focused on tasks to build a system. An installation of an issue repository typically allows a project to set various configuration parameters to help fit the repository to the needs of the project. Despite differences between issue repositories, many commonalities exist. Figure 2.2 is an example of an issue (GATEWAY-3941) in the issue repository for the open source project Connect. I describe the relevant parts of an issue that are common amongst most issue repositories and that are used in this thesis to identify links between issues and commits. The top bar (1) shows the issue id with the title of the issue. On the right hand side (2), are people involved with the issue and important dates for the issue. Underneath the title (3) are details that a developer uses to understand the scope of the issue with respect to the system and to find the status of the issue. The description of the issue informs the assigned developer details necessary to complete work on the issue (4). Issue relationships provide links to issues that have some association with the issue (5) and (6). The bottom contains communications with other developers, and other activities to help keep track of changes to the issue (7).

6connectopensource.atlassian.net/browse/GATEWAY-3941 verified 13 July 2017
Figure 2.2: Gateway-3941 an issue from Jira issue repository for the connect open source project
2.1.3 Issue Relationships

Developers use relationships between issues to hold information about how functionality in a system is related or how work is broken down. The ways in which developers use relationships in issue repositories can reveal the degree to which a project follows a particular process or to which a project tracks dependencies between system functionality, amongst others. Figure 2.2 shows the issue relationships, (5) and (6), for the issue (GATEWAY-3941). For example in this issue, one of the relationships developers use is the ‘depends on’ relationship to describe the implementation of the issue (GATEWAY-3941) depends on research of the issue (GATEWAY-3895).

Developers have a limited amount of time to perform many different activities, such as planning, developing, and testing. Even with limited time, developers choose to spend manual effort creating relationships between issues. Across three major open source projects, Mylyn, Connect, and HBase, a large percentage of issues have relationships, an average of 35% per repository, even though they use different issue repositories. Connect has the largest percentage at 75% which means that many of the issues in this project have a relationship to another issue. The large percentage of issue relationships means that developers use the relationships to capture information on which they rely.

Developers use a variety of different types of relationships in issue repositories. In Bugzilla the default issue relationships that are used are depends-on and duplicates. In Jira, there are five default relationships, which are relates to, duplicates, blocks, clones, and sub-task. When JIRA is used for a project following an agile project management style, it is also common to see such relationships as supported-by and part-of-epic.

Table 2.1 shows the many types and instances of relationships for three open source systems: Mylyn, Connect and HBase. There is common meaning...
Table 2.1: Relationships instances

<table>
<thead>
<tr>
<th></th>
<th>Mylyn</th>
<th>Connect</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocked</td>
<td>299</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Breaks</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Clone</td>
<td>32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Contains</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depends-on</td>
<td>2174</td>
<td>205</td>
<td>286</td>
</tr>
<tr>
<td>Duplicates</td>
<td>1520</td>
<td>43</td>
<td>159</td>
</tr>
<tr>
<td>Incorporates</td>
<td></td>
<td></td>
<td>208</td>
</tr>
<tr>
<td>Part-of-epic</td>
<td></td>
<td></td>
<td>559</td>
</tr>
<tr>
<td>Requires</td>
<td></td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>Relates-to</td>
<td></td>
<td></td>
<td>1901</td>
</tr>
<tr>
<td>Subtask</td>
<td>1643</td>
<td></td>
<td>1368</td>
</tr>
<tr>
<td>Supercedes</td>
<td></td>
<td></td>
<td>31</td>
</tr>
<tr>
<td>Supported-by</td>
<td></td>
<td></td>
<td>1361</td>
</tr>
<tr>
<td>Total relationships</td>
<td>3694</td>
<td>3811</td>
<td>4532</td>
</tr>
<tr>
<td>Total issues</td>
<td>12959</td>
<td>5154</td>
<td>13334</td>
</tr>
</tbody>
</table>

among the different relationship types across different repositories. As an example of common meaning in relationships, Bugzilla’s depends-on has similar meaning to Jira’s sub-task.

### 2.2 Existing Techniques to Link Commits to Issues

In general, most linking consists of developers manually stating links as part of their development process. A number of automated tools have also been created to help automate this step and to improve linking data from existing repository information. I describe the how developers manually state issues in commits, and two approaches, traditional heuristics, and state-of-the-art techniques.

#### 2.2.1 Manual Linking by Developers

For many projects, developers follow company or project development practices, by manually inserting issue ids in commit messages and commit SHAs into issues. These practices are either explicitly written in a contributor’s guide, such as found...
in the Connect project contributors guide\textsuperscript{12}, or are implicit practices found by looking at commit messages. The system of manually stating the issue id is reliant upon developers never forgetting to add the issue id and exactly matching the issue name and id. Bird and colleagues show that developers do forget to state issue ids in commit messages, resulting in incomplete liking data \textsuperscript{4}.

As an example, for the Connect project, a developer stated issue id in the commit message is expected to reference the issue in a fixed format in the commit message. An example of a manual link is in Figure 2.1 above, this shows the author (developer) (2) wrote the commit message (5) “CONN-1190: Rebase conflict resolved for glassfish-web.xml.” The author manually included the issue id “CONN-1190” in the commit message. These issue ids follow a pattern that is unique to each issue repository, for example in Connect the pattern is “CONN-” followed by a number.

Other tool support, such as Mylyn\textsuperscript{13}, exists to automatically create commit messages with the issue id. This tool is a plugin to the open source Eclipse IDE\textsuperscript{14} and allows developers to track automatically source code that is relevant to the issue being addressed. The tool requires a developer to manually indicate what issue is being worked on. When a developer is finished working on an issue and makes a commit, the tool inserts the issue id and title of the issue as the commit message. Use of such tooling increases then number of commit messages with an issue id, making later retrieval and identification easier.

\subsection{Traditional Heuristics}

Early work attempts to coalesce both an issue repository and version control system by creating links between issues and commits. The work relies on developers manually stating the issue id in the commit message. I refer to this as traditional heuristics as this is the first method used to link issues and commits. Traditional heuristics uses patterns to link commits to issues, looking in the commit message for key words like “fix” or “bug” in combination with an issue id or finding issue

\textsuperscript{12}connectopensource.atlassian.net/wiki/display/CONNECTWIKI/How+to+Contribute+Code verified 13 July 2017
\textsuperscript{13}eclipse.org/mylyn verified 13 July 2017
\textsuperscript{14}eclipse.org verified 13 July 2017

11
ids in a fixed format like “CONN-12”. These techniques minimize false positives by removing links from commits to non-existing issues in the issue repository, and remove links where the commit is made more than 7 days before or after an issue is solved. Multiple papers use traditional heuristics to implement tools such as defect prediction, identify changes that introduce bugs, or predicting when an issue will be reopened [18, 21, 23, 27, 33, 49, 57].

There exists tool support to automatically create links from issues to commits, such as add-ons for Jira. These add-ons scan commit messages in a version control system and extracts issue ids that match the issue ids found in the issue repository. Any deviation from an exact issue match would cause the tool to miss a potential link. Examples of Connect commit messages that would not match are “CON-20 Updated PDP”, and “CONNECT-1368: More updates.” Although these commit messages reference an issue, the tool would not create a link because the issue ids do not match the any issue in the issues repository.

2.2.3 State-of-the-Art Techniques

Recent research in linking commits to issues primarily uses information retrieval (IR) techniques that are not dependent on the developer stated issue id in commit messages [10, 32, 35, 45, 56]. There are three approaches state-of-the-art techniques used to link commits to issues: the first is to use similarity of the text in the commit and issues, the second to use only heuristics based on authorship and time, and the last to use machine learning to create the links.

Wu and colleagues were the first to use an IR based linking technique (Relink); using similarity of the text in a commit and issue rather than being based off identifying issue ids in commit messages [56]. Relink uses three criteria for linking a commit to an issue: similarity of the commit message to the text in the issue title, description, and comments; the time between an issue comment and a commit; the author of the commit must be assigned to the issue. Relink uses the links identified with heuristics to set a minimum threshold for similarity of text between a commit and issue, and time between a issue comment and a commit necessary to link commits to issues.

Nguyen and colleagues build on Relink to create a technique called Mlink, the
technique uses code elements to identify links between issues and commits [35]. Mlink identifies code elements in issues and compares them to the code changes in a commit. In addition to using Relinks techniques, Mlink uses both commit message and comments added in the code and compares them to the text in the issue title, description and comments. Mlink differs from Relink as it does not use authorship to determine if a commit links to an issue. Mlink optimizes thresholds using the same process as Relink.

Bissyande and colleagues technique differs from both the Relink and the Mlink techniques by using only similarity of the text in commits and issues to link commits to issues [10]. They use three techniques to identify text in issues and commits: vector space modeling (VSM), latent semantic analysis (LSA), and latent Dirichlet allocation (LDA). In the end Bissyande and colleagues find that only using text similarity performs worse both Relink and Mlink.

Loaner and Phantom [45] use heuristics with time and authorship to determine if a link should be made. Unlike Relink, these techniques do not use text similarity. The Phantom technique identifies issues that have existing linked commits and finds similar commits based on files changed to link to the issue. The Loaner technique links commits to issues that have no existing link, and differs from Relink as it does not require text similarity between issues and commits. Loaner and Phantom optimizes thresholds using the same process as Relink.

RCLink [32] uses machine learning to link commits to issues. Le and colleagues, the authors of RCLink, use machine learning to create links between commits and issues. The technique creates features from information found in issues and commits for their machine learning technique. Le and colleagues report that RCLink performs better than Relink [56], Bissyande and colleagues technique [10], and MLink [35].

2.3 Research in Understanding Issue Repositories

I propose in this thesis the idea of using structure in issue repositories, as found in relationships between issues, to improve software process data. I provide in this section an overview of related research in issue repositories.
2.3.1 Research that Studies Issue Tracking Systems

A number of researchers have investigated and described characteristics of issues and issue repositories. Mockus and colleagues characterize aspects of problem reports (i.e., issues), such as time to resolve a problem, as part of a characterization of open source development in the Mozilla\footnote{mozilla.org, verified 13 July 2017} and Apache\footnote{apache.org, verified 13 July 2017} projects \cite{mockus}. Anvik and colleagues characterize the Eclipse\footnote{eclipse.org, verified 13 July 2017} and Firefox\footnote{mozilla.org, verified 13 July 2017} repositories describing bug triage and duplicate bug problems that happen in open source repositories \cite{anvik}. Ko and colleagues analyzed titles of individual issues to determine such aspects as the degree to which issues refer to particular parts of a system and how much regularity there is between issue titles \cite{ko}. Bettenburg and colleagues consider what additional information should be included in a issue to assist a developer \cite{bettenburg}. Banerjee and colleagues examined Mozilla and Eclipse repositories, finding that the maturity of a reporter reduces how often insignificant, poor quality, and duplicate bugs are detected \cite{banerjee}. Jankovic and colleagues find issues and commits can be used to reconstruct software processes; they define issues as parallel or sequential with the existence or nonexistence of a “block” relationship link between issues \cite{jankovic}.

2.3.2 Approaches that Target Issue Tracking Systems

Other research efforts use the information in an issue repository to improve software development. Sandusky and colleagues examined and grouped reports in an issue repository to see if groupings could improve the management of problems \cite{sandusky}. Cubranic and Murphy \cite{cubranic} and Anvik and colleagues \cite{anvik} applied machine learning to categorize issues automatically and to automate such bug triage activities as assigning the issue to an individual for attention. Runeson and colleagues \cite{runeson} applied natural language processing methods to automatically recommend duplicate issues. Wang and colleagues use a combination of natural language processing execution information to improve recall detection of duplicate bug reports \cite{wang}. Rocha and colleagues recommend the next bug to work using cosine similarity to find similar bug \cite{rocha}. Two works of which I am aware make use
of relationships between issues. The first by Rastkar and Murphy describes how extractive summarization can be applied to produce summaries of linked issues automatically [39]. The second by Choetkiertikul and colleagues uses explicit and implicit relationships between issues to predict delays in a software project and shows that relationship information can help to more accurately predict delays in software projects [15]. By characterizing relationships between issues, I aim to identify more opportunities for using relationship information to improve software development practices.

2.3.3 Software Process Work

Initial research in software process attempted to prescribe a software process model to developers, which is a formal representation of the process to be used in development [11]. Others have considered how to interpret software data from VCS or issue repository information. Cook and Wolf introduced automatic identification of formal software process models though event logs generated from data collected from version control systems and issue repositories. Cook and Wolf describe three techniques using event log data to construct a UML like process model, using statistics, Markov chain, or neural networks [16]. Kindler and colleagues use the events in a version control system to generate event logs, in combination with a developer annotating the purpose of each commit, to generate UML like process models from event logs [28]. Hindle and colleagues use version control systems, issue repositories, and mailing lists to generate activity timelines showing software process [22]. Duan and colleagues create dynamic project specific UML like software process models by making assumptions on the purpose of files in a version control system to automatically generate event logs [19]. Research specific to event log creation has focused on how data can be extracted from multiple repositories in order to enable process model generation reuse. Poncin and colleagues introduce Fraser a tool to automate event log creation from any number of version control systems and issue repositories. They show that by separating event log extraction and analysis steps into distinct phases allows reuse of analysis tools [36]. These approaches look at high level aspects of software process in version control systems and issue repositories to create process models.
Chapter 3

Assessing the Quality of State-of-the-Art Techniques

A growing number of software engineering techniques rely on software process data. As described in the introduction, one particular form of data that is relied upon is the association (or linking relationship) of commits to issues. Despite the widespread use of this data, there does not exist a standardized means to assess the quality of the data. In this chapter, I introduce two quality attributes to assess the linking relationship: completeness and consistency. A linking relationship is complete when all commits link to an issue, and is consistent when commits are linked to the most specific issue.

If software process data lacks completeness or consistency, then the techniques, which rely on that data, may produce mis-leading results. From an evaluation perspective, a lack of completeness or consistency can be evaluated with potential bias, i.e., the recall (due to completeness) and the precision (due to consistency) of these techniques may be incorrect. To assess the degree to which existing techniques aimed at improving the linking relationship achieve completeness and consistency, I assessed four state-of-the-art techniques. My method of assessment involved selecting 10 open source projects, and running the four state-of-the-art techniques on the 10 projects to identify links for commits not linked to an issue. Using the results of state-of-the-art techniques I show that each technique does not create fully complete and consistent linking relationship. In addition, the existing linking rela-
tionship in each project is neither complete or consistent which is needed in order to compare techniques and create a new technique to link commits to issues.

3.1 Completeness and Consistency

Software techniques, such as defect prediction, traceability, and prediction for time to completion of issues rely on the link relationship between issues and commits [15, 21, 27]. The linking relationship is defined by set $L$, where $L$ is a set of pairs (links) $(i, c)$, $i$ is an element (an issue) in the issue repository $I$, and $c$ is an element (a commit) in the VCS $C$ (see equation 3.1.)

$$L \subseteq (i, c) \in I \times C$$ (3.1)

Each link expresses which code and other artifacts were changed as part of an issue (i.e. a link from an issue to a commit or commits.) If many links in the set $L$ are missing or are not accurate, then the software techniques can produce incorrect results which are discussed later in this section. $L$ is a many-to-many relationship: commits can link to none or multiple issues, and an issue can link to none or multiple commits. To help assess the quality of $L$ for a given project, I introduce the concepts of completeness and consistency.

3.1.1 Completeness

I consider $L$ to be complete when all commits are linked to one or more issues. Formally defined $L$ is complete when $\text{Complete}(L)$ is true (see equation 3.2.)

$$\text{Complete}(L) \equiv \forall c \in C. \exists (i, c) \in L$$ (3.2)

Meaning that for every commit $c$ in the VCS $C$ there exists a link $(i, c)$ in $L$. Figure 3.1a shows an example of a completely linked project. The project has all commits linked to one or more issues in an issue repository. The completeness for a project $\text{Complete}(L)$ is assessed quantitatively using the percentage of commits linked to issues.

A stronger definition of completeness would place a constraint that all issues appear in $L$. However, in most issue repositories, there exist issues that describe
work or intended aspects of a project that do not result in changes to artifacts; as such, this constraint could never be fully met. Some reasons that issues can be created which do not address source code changes are to provide potential directions that a project might take, or to provide customer support for a part of a project [1]. Due to the many potential reasons to create an issue, linking all issues to a commit is not likely to occur, and is not investigated for this thesis.

As an example of the importance of completeness of $L$ consider defect prediction, which relies on the identification of files that often have bugs. The message accompanying a commit—known as a commit message—and the documentation in changed artifacts, may not provide enough detail to automatically determine the reason behind the commit (i.e., bug fix, feature implementation, etc.) Defect prediction uses $L$ to identify why a change happened and to improve the prediction algorithm. An incomplete $L$, where all commits are not linked to an issue, can cause the defect predictor to give less accurate results [8].
3.1.2 Consistency

Multiple issues might be used to describe the source code changes in a commit. These issues may have relationships between themselves, which are manually created by developers as explained in Chapter 2.1.3. L is consistent when commits link to the most specific issues, for those issues involved in a relationship. By specific, I mean the issue’s text describe the reasoning for the source code changes in those commits. An issue relationship is defined by \( R \) which is a set of pairs \( i_i, i_j \) where both \( i_i \) and \( i_j \) are in the set of issues \( I \), and there is a relationship between \( i_i \) and \( i_j \) (see equation 3.3.)

\[
R \subseteq (i_i, i_j) \in I \times I \tag{3.3}
\]

A subset of links \( L' \) involve issues that have relationships to other issues, and is used to determine the consistency of \( L \). \( L' \) is a subset of the links in \( L \) containing pairs \( (i', c) \) where \( i' \) is in either side of a relationship in the set \( R \).

\[
L' = \{(i', c) \in L | (i', i) \in R \text{ or } (i, i') \in R \text{ for some } i' \in I \} \tag{3.4}
\]

I focus on set \( L' \), because the issues in these links are part of a relationship and developers may manually state a link, as part of their development process, to a commit to a less specific issue that has a relationship. As an example of linking to a less specific issue consider Figure 3.1b where issues B and C have a relationship to issue A, and the intent of source code changes in commit 1 are described in issue A. If a developer links commit 1 to issue C, the link is to a less specific issue and the link should be changed to issue A, which is the most specific issue.

To determine if a commit is linked to the most specific issue in a relationship, I define a function \( f \) that takes a link, and the set \( R \) and returns true if a commit \( c \) is linked to the most specific issue \( i' \) (see Equation 3.5.) \( L \) is considered to be consistent when the function \( \text{Consistent}(L) \) is true. This means \( L \) is consistent if the function \( f \) determines that all the commits in \( L' \) are linked to the most specific
issues (see Equation 3.6.)

\[ f((i', c), R) = \begin{cases} 
    \text{true} & \text{if } c \text{ is linked to the most specific } i' \\
    \text{false} & \text{otherwise}
\end{cases} \]  \hspace{1cm} (3.5)

Consistent \((L) \iff \forall (i', c) \in L'. f((i', c), R)\)  \hspace{1cm} (3.6)

Figure 3.1b shows an example of a consistently linked project. Each commit is linked to the most specific issue in the project. Function \(f\) is done manually to check each link and identify if the commit is linked to the most specific issue, if function \(f\) returns false \(L\) is considered to be inconsistent.

Linking commits to the most specific issue is important for software tools that predict delay in software projects [15]. These tools extract features from links, and use those features to train machine learning techniques to predict delay. Choetkertikul and colleagues predict delays using the time difference between when an issue is opened and closed and the files changed, among other features. If a commit is linked to a less specific issue the time between open and closed may be different and cause the estimation to be inaccurate. Having commits linked to the most specific issue may improve time to completion estimations.

### 3.2 Performance of State-of-the-Art

To demonstrate some of the problems with software process data, I evaluated the completeness and consistency of four state-of-the-art techniques across 10 open source projects that have an issue repository with issue relationships. The four techniques examined represent three approaches techniques used to do linking: Re-link [56] which uses similarity of the text in commits and issues, Loaner, and Phantom [45] which uses heuristics based on the authorship and time, and RCLink [32] which uses machine learning. Completeness for each technique is evaluated quantitatively by reporting on the percentage of commits linked to an issue. Consistency is evaluated qualitatively, by manually examining the results of each technique to determine based on the semantics of each commit whether the commit is linked to the most specific issue with a relationship.
3.2.1 Identifying Projects

I focused on open source projects because the VCS for these projects are easily accessible. I used Github\(^1\) as a way to identify open source projects. Github is a widely used collaborative online platform for developers and companies to store, and share a project’s version control system. As all of the state-of-the-art techniques were built for Java projects, I selected Java projects to be evaluated. On Github, projects can be ordered by the number of stars that users of Github have given the project. I follow the lead of other researchers \([13, 59]\), in using star ordering to avoid projects of low quality \([26, 40]\).

To assess consistency, I required projects that capture work within issues and use relationships between issues. I chose projects that use a Jira issue repository because Jira issue repositories by default allow developers to make relationships between issues. To identify if a project uses Jira I search the “readme” file of the Github project for the word “Jira.”

3.2.2 Selecting Projects

To be suitable as a target for evaluations, I required projects to have manually stated links by developers, and for the projects’ issue repository to use issue relationships. Manually stated links are needed because all of the state-of-the-art linking techniques being evaluated need projects that have existing links. To assess consistency, I need projects that have relationships between issues.

To ensure projects used relationships, 20 issues were randomly sampled from the Jira issue repository of each project. A project was selected if 25% or five of the 20 sampled issues participated in an issue relationship. I selected 25% as a threshold because work by Jankovic and colleagues showed an average of 24% of issues had an issue relationship across 3 open source projects that used Jira \([24]\). I did further data collection and downloaded all Apache projects that used JIRA and found 24% of over 4 million issues had an issue relationship. The projects that have existing links and over 25% of the issues have a relationship are expected to be typical open source projects.

I had to consider the first 17 projects to find 10 suitable projects. The selected

\(^{1}\text{github.com, verified 13 July 2017}\)
Table 3.1: Statistics of the projects selected

<table>
<thead>
<tr>
<th></th>
<th>Commits</th>
<th>Issues</th>
<th>Issues with Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>cordova-android</td>
<td>2903</td>
<td>11957</td>
<td>4232</td>
</tr>
<tr>
<td>Flink</td>
<td>9887</td>
<td>5935</td>
<td>1402</td>
</tr>
<tr>
<td>H2o 3</td>
<td>16102</td>
<td>3954</td>
<td>1005</td>
</tr>
<tr>
<td>Maven</td>
<td>10205</td>
<td>4954</td>
<td>2044</td>
</tr>
<tr>
<td>Mongo Java</td>
<td>5561</td>
<td>2240</td>
<td>725</td>
</tr>
<tr>
<td>Mongo Hadoop</td>
<td>1085</td>
<td>281</td>
<td>58</td>
</tr>
<tr>
<td>Nutch</td>
<td>2212</td>
<td>2351</td>
<td>654</td>
</tr>
<tr>
<td>Pentaho</td>
<td>16049</td>
<td>14282</td>
<td>5503</td>
</tr>
<tr>
<td>Sonarqube</td>
<td>20905</td>
<td>7658</td>
<td>3623</td>
</tr>
<tr>
<td>Spring Framework</td>
<td>13603</td>
<td>15118</td>
<td>4723</td>
</tr>
</tbody>
</table>

projects are shown in Table 3.1 with the number of commits, issues, and the number issues with a relationship.

3.2.3 Technique Modifications

For all the state-of-the-art techniques assessed, I needed to identify the links stated manually by developers. I wrote a script that finds issue ids in commit messages using pattern recognition. I selected commits in 2016 and issues that are created, commented or resolved in 2016 for each project. Table 3.2 shows the total number of commits and issues for 2016 identified and the percentage of commits with links identified using the pattern recognition.

Each state-of-the-art technique needs specific changes in order to map the project data to the input each technique expects. I describe the changes made to each technique in turn.

Relink

To use Relink, I needed to adjust the issues and commits for each project. Relink expect issues, and commits to be numbered, I mapped issues ids to numbers 1 to n, and commit SHA to numbers 1 to n. Relink also expects the mapped issue number to be displayed in the commit message if there is a link, I created a mapping to

---

2I used the following pattern, (?Ni) <projectname> [/w] * /d+, for each project <projectname>is replaced with the name used in the issue id.
Table 3.2: The number of commits and issue in 2016, and % completeness of manually stated links by developers

<table>
<thead>
<tr>
<th>Project</th>
<th>Total Commits</th>
<th>Total Issues</th>
<th>Existing Completeness %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cb</td>
<td>178</td>
<td>3121</td>
<td>47</td>
</tr>
<tr>
<td>Flink</td>
<td>2015</td>
<td>2710</td>
<td>77</td>
</tr>
<tr>
<td>H2o 3</td>
<td>3705</td>
<td>1950</td>
<td>21</td>
</tr>
<tr>
<td>Maven</td>
<td>50</td>
<td>521</td>
<td>63</td>
</tr>
<tr>
<td>Mongo Hadoop</td>
<td>53</td>
<td>278</td>
<td>43</td>
</tr>
<tr>
<td>Mongo Java</td>
<td>245</td>
<td>2205</td>
<td>66</td>
</tr>
<tr>
<td>Nutch</td>
<td>112</td>
<td>341</td>
<td>60</td>
</tr>
<tr>
<td>Pentaho</td>
<td>928</td>
<td>9944</td>
<td>29</td>
</tr>
<tr>
<td>Sonarqube</td>
<td>3214</td>
<td>7658</td>
<td>65</td>
</tr>
<tr>
<td>Spring Framework</td>
<td>2163</td>
<td>1938</td>
<td>49</td>
</tr>
</tbody>
</table>

identify issue ids in the commit message and map it to the correct number from the previous step. For example when CONN-1190 is found in a commit message it is replaced by 120.

**Phantom and Loaner**

The issues for both Loaners and Phantoms needed to be adjusted for each project. Both techniques expect to link commits to issues that are in a “resolved” state. Projects have different development processes than the ones evaluated by the Phantom and Loaner techniques. I mapped “closed” issue status to “resolved” to match expected input for these techniques. Additionally, both techniques expect the issues to be classified by one of the three categories: bug, feature, other based on issue type. Using the definition of each category in their paper [45], I create the same classifications for issue types not mentioned\(^3\).

\(^3\)The issue types fault, pruning and refactoring were categorized as bug type. Dependency upgrade, engineering story, new feature, request, spike, story, sub-task, technical task, and test all were categorized as feature type. The rest were categorized as other type.
RCLink

RCLink expects a training set that has 10 times more commit issue pairs that are unlinked then then manually liked commits. To achieve this ratio, I randomly selected from the commits with a link, and issues not linked to that commit. The last change I did was refactorings to their algorithms in order to speed up their technique. Their algorithms had redundancy in the calculations, and I created a map to identify if an calculation was already made in order to avoid repetition. The refactorings were validated on their test data.

3.3 Comparison of State-of-the-Art Techniques

This section discusses how state-of-the-art techniques perform with regards to completeness and consistency in the projects identified in the previous section. Completeness is evaluated by looking at the percentage of commits linked to an issue. Consistency is measured manually by sampling five links from each project and determining if the commit is linked to the most specific issue. Five are chosen because it is a time consuming manual process.

3.3.1 Completeness

Table 3.3 reports the percentage completeness of the linking relationship after applying the state-of-the-art technique and the percentage of improvement. The results show that RCLink has a higher percentage of completeness than the other three algorithms for most of the 10 projects. In some cases, for example H2O, Pentaho, it more than doubles the number of commits linked to an issue. Though in some situations, like with CB, Mongo Java, and Nutch, RCLink does not have the most completeness, but is close to the better performing technique, with only one or two percentage points different. RCLink performance may be due to the technique being able to identify links in more situations. Phantom performed the worst of the four algorithms only increasing completeness by 1% on 2 projects. The poor performance may be because the technique works only in instances where projects have multiple commits that should link to a single issue.
Table 3.3: The % completeness for existing work and % improvement (Improv.) of linking relationship completeness of each state-of-the-art technique on selected projects.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Cb</td>
<td>47</td>
<td>22</td>
<td>2</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>Flink</td>
<td>77</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>h2o 3</td>
<td>21</td>
<td>5</td>
<td>13</td>
<td>1</td>
<td>142</td>
</tr>
<tr>
<td>Maven</td>
<td>63</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Mongo Hadoop</td>
<td>43</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>13</td>
</tr>
<tr>
<td>Mongo Java</td>
<td>66</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Nutch</td>
<td>60</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>8</td>
</tr>
<tr>
<td>Pentaho</td>
<td>29</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>123</td>
</tr>
<tr>
<td>Sonarqube</td>
<td>65</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>14</td>
</tr>
<tr>
<td>Spring Framework</td>
<td>49</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>26</td>
</tr>
</tbody>
</table>

3.3.2 Consistency

To evaluate consistency, I manually examine links to determine if the commit is linked to the most specific issue. I then report on whether a project has a consistent or inconsistent linking relationship, and if state-of-the-art techniques make changes to any of the links examined. I refer to the issue that is the source of a relationship as the parent and the issue that is the target of a relationship as the child.

For the 10 projects examined there are relationship types defined. Table 3.4 shows all relationship types and the number of instances of each relationship type that contain a issue with a link to a commit. I focus on issues with relationships that break work down, where the parent issue is split into child issues, which allow various parts of an issue to be worked on separately. This breakdown of work may cause developers to inconsistently link commits to issues, (i.e. linking a commit to a parent rather than a child.) I examined Jira documentation and identified that the “sub-task” relationship is used to break an issue into child issues (the documentation refers to these as smaller pieces of a larger task\(^4\)).

As can be seen in Table 3.4 not all projects use the “sub-task” relationship. I manually examined at all relationship types for the seven projects with less than five

\(^4\)confluence.atlassian.com/adminjiracloud/issue-types-844500742.html verified 13 July 2017
Table 3.4: Relationships instances in each project that have an issue linked to a commit. Bold numbers are relationships that represent work breakdown in the issue repository.

<table>
<thead>
<tr>
<th></th>
<th>Cb</th>
<th>Flink</th>
<th>H2o 3</th>
<th>Maven</th>
<th>Hadoop</th>
<th>Java</th>
<th>Nutch</th>
<th>Pentaho</th>
<th>Sonarqube</th>
<th>Spring Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>blocks</td>
<td>0</td>
<td>30</td>
<td>11</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>breaks</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>causes</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>clones</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>contains</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>contributes to</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>depends on</td>
<td>1</td>
<td>27</td>
<td>0</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>0</td>
<td>112</td>
<td>38</td>
</tr>
<tr>
<td>deprecates</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>duplicates</td>
<td>10</td>
<td>46</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>13</td>
<td>3</td>
<td>25</td>
<td>35</td>
<td>39</td>
</tr>
<tr>
<td>implements</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>incorporates</td>
<td>0</td>
<td>11</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>part of Epic</td>
<td>0</td>
<td>0</td>
<td>54</td>
<td>0</td>
<td>3</td>
<td>21</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>relates to</td>
<td>4</td>
<td>70</td>
<td>37</td>
<td>20</td>
<td>3</td>
<td>13</td>
<td>10</td>
<td>130</td>
<td>107</td>
<td>662</td>
</tr>
<tr>
<td>replaces</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>requires</td>
<td>0</td>
<td>22</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Sub-task</td>
<td>3</td>
<td>197</td>
<td>30</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>84</td>
<td>4</td>
</tr>
<tr>
<td>supersedes</td>
<td>0</td>
<td>21</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>13</td>
</tr>
<tr>
<td>total relationships</td>
<td>25</td>
<td>429</td>
<td>139</td>
<td>42</td>
<td>7</td>
<td>48</td>
<td>27</td>
<td>163</td>
<td>359</td>
<td>756</td>
</tr>
</tbody>
</table>
instances of a “sub-task” relationship; I discovered that “depends on” is also used in these projects to break work down. Although “relates to” is used frequently across all projects, it is used to describe work breakdown and features that address the same part of a project, due to its inconsistent use I do not use it in this evaluation. No work breakdown relationships were identified in the Mongo Hadoop after looking at all of the relationships.

To select issues for the manual examination, I randomly sample up to five links from each project. The issues in the links must meet the following criteria:

1. part of a work breakdown relationship
2. work that needs to be a parent in the relationship
3. marked as resolved or closed

The first criteria selects only issues that are in a work breakdown relationship, (i.e. those issues that have a “sub-task” or “depends on” relationship.) The second criteria is used to select the issues that have their work to be broken down (i.e. I select the parent in the “sub-task” or “depends on” relationship), as developers may link to the parent issues rather than the broken down parts. The last criteria is to select issues that are resolved or closed to remain consistent with the method each state-of-the-art technique uses to link commits to issues. There were a total of 29 links selected for the 10 projects, the number of links for each project is shown in Table 3.5. None links in the Mongo Java were to issues that needed work to be broken down.

The next step is to manually look at the commits selected and determine if each commit is linked to the most specific issue. To do the manual examination of a link I look at the commit, the issue and all of related issues (i.e. those issues that have a relationship to the issue in the link.) The process to identify if a commit should be linked to a different issue is done by comparing the issue title, description and comments, with the commit message, files changed and changes to code, to determine if the commit address the issue. This process is repeated for all related issues, and if another issue should be linked I mark the link as inconsistent.

The results of manually examining the 29 links across the projects are reported in Table 3.5. All links examined can be found in Appendix A I found inconsistency
Table 3.5: Results of manually examining consistency of up to five links from each project.

<table>
<thead>
<tr>
<th>Project</th>
<th># links examined</th>
<th>Project consistency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cb</td>
<td>1</td>
<td>Consistent</td>
</tr>
<tr>
<td>Flink</td>
<td>5</td>
<td>Inconsistent</td>
</tr>
<tr>
<td>H2o 3</td>
<td>3</td>
<td>Inconsistent</td>
</tr>
<tr>
<td>Maven</td>
<td>3</td>
<td>Consistent</td>
</tr>
<tr>
<td>Mongo Hadoop</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Mongo Java</td>
<td>0</td>
<td>N/A</td>
</tr>
<tr>
<td>Nutch</td>
<td>3</td>
<td>Consistent</td>
</tr>
<tr>
<td>Pentaho</td>
<td>2</td>
<td>Inconsistent</td>
</tr>
<tr>
<td>Sonarqube</td>
<td>5</td>
<td>Inconsistent</td>
</tr>
<tr>
<td>Spring Framework</td>
<td>5</td>
<td>Inconsistent</td>
</tr>
</tbody>
</table>

In four projects: Flink, H2o 3, Pentaho, and Sonarqube. An example of inconsistency is the link from the H2o 3 project where the commit should link to the most specific task (PUBDEV-2746) rather than the less specific task (FLINK-1843):

**Parent** PUBDEV-1843: Grid testing

**Child** PUBDEV-2746: Make sure GridSearch will print a warning for bad hyper parameter names and values

**Commit** 162c909: PUBDEV-1843: grid test, subtask 7. Java error/warning messages are ignored when they are generated for unit tests for gridsearch. Added code to print out these error/warning message during unit tests so we can better debug our code.

In this example, the commit 162c909 is linked by the developer to the issue PUBDEV-1843 even though the child issue PUBDEV-2746 contains a description discussing printing an error message. The commit message mentions “sub-task 7” and “printing error messages.” In addition, the source code of the commit makes changes that relate to printing error messages in grid search. The correct link should be to PUBDEV-2746 instead of PUBDEV-1843.

I then look at the links created by each of the state-of-the-art techniques and report if the commits are linked to the more specific task identified in the previous
step. I found Relink and RCLink both created a link to the more specific issue identified for a link in the H2o 3 project.

**Parent**  
PUBDEV-3482: Supporting GLM binomial model to allow two arbitrary integer values

**Child**  
PUBDEV-3791: Documentation: Add quasibinomomial family in GLM

**Commit**  
3ecf05e: PUBDEV-3482 Per Erin’s request, changed quasi_binomial family name to quasibinomial.

In this case, the commit 3ecf05e is linked by the developer to the issue PUBDEV-3482, and improved by both Relink and RCLink. The child issue PUBDEV-3791, contains comments discussing changing the name in source code from “quasi_binomial” to “quasibinomial.” In addition, the source code of the commit shows there are only rename changes: “quasi_binomial” to “quasibinomial.” Both Relink and RCLink both created a link to the more specific issue PUBDEV-3791. Although both approaches did not solve inconsistency for all links.

### 3.3.3 Summary of Quality Assessment

I assessed four state-of-the-art techniques using two new quality attributes completeness and consistency on 10 different projects. I found a lack of completeness and consistency in the linking relationship of the projects examined. Developers did not link all commits to an issue to create a complete dataset in any of the 10 projects. In addition, in four of the 10 projects, the linking relationship was not consistent. I also found that although state-of-the-art techniques do make improvements to completeness, only Relink and RCLink made improvements to consistency of projects, though they did not make any of the projects fully consistent.
Chapter 4

Creating a Complete and Consistent Dataset

In this thesis, I have motivated the need for improving software process data consisting of links between commits and issues. To help support the assessment of techniques developed to improve the software process data automatically, it would be useful to have a dataset that has a high degree of completeness and consistency. To test a new technique developed to improve software process data, one could remove links from the dataset and test if they can be replaced.

In this chapter, I describe the shortcomings of existing datasets used to test proposed techniques to date and I describe a dataset, consisting of two projects, I have created to provide a suitable dataset.\footnote{See \url{www.cs.ubc.ca/labs/spl/projects/issueRelationships/}}

4.1 Existing Datasets

State-of-the-art techniques to link commits to issues are created and validated using dataset two types of datasets. The first type of dataset is created as a byproduct of developer’s work; while the second type of dataset is created manually by researchers.

The first type of dataset that exists is created from a byproduct of the development process. Developers follow company or project development practices,
manually entering links between issues and commits, in issue descriptions and commit messages\(^2\) (discussed in Chapter 2.2.1). As an example, links for the Connect project are stated by a developer adding an issue id in a fixed format like “CONN-<number>” to a commit message. This link information is used by researchers to create datasets of linked issues and commits from projects such as Apache, Mozilla, and Eclipse [10, 35, 45, 49, 56, 60]. This type of dataset has two limitations, the first limitation is the links stated by developers are not validated to be correct or consistent. The assumption by researchers is developers always state prefect links between issues and commits. The other limitation of this type of dataset, as demonstrated in Chapter 3.3.2 is that developers do not always link commits to the most specific issue.

To create the second type of dataset, researchers choose a project and go through all commits and issues, manually identify the links that exist between commits and issues. This technique has been used by Wu and colleagues on two projects: zxing and openintents [56]. This type of dataset has two limitations, the first is projects selected for these datasets have issue repositories with a flat structure, meaning that there are no identified relationships between issues. Relationships between issues are necessary to assess consistency of a linking relationship. The other limitation of this dataset is there was no described process of how commits are linked to issues, which prohibits reproducibility for future datasets.

If state-of-the-art techniques are created using these incomplete and inconsistent datasets then the heuristics or machine learning algorithms for the techniques may make an incomplete and inconsistent linking relationship. Further, if the state-of-the-art techniques are validated on an incomplete and inconsistent dataset the results may not be valid.

### 4.2 Dataset Creation Process

To create a dataset with more complete and consistent links, I choose two projects that have existing relationships between issues, which is representative of how issue repositories are formed today. A time window of dates is selected from both projects to create a dataset of feasible size, as manually linking commits to issues
Table 4.1: Commit and issue information for each project.

<table>
<thead>
<tr>
<th></th>
<th>Connect</th>
<th>Sonarqube</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits</td>
<td>1707</td>
<td>9343</td>
</tr>
<tr>
<td>% Commits with link</td>
<td>79</td>
<td>62</td>
</tr>
<tr>
<td>Issues</td>
<td>1625</td>
<td>3500</td>
</tr>
<tr>
<td>Relationships per issue</td>
<td>2.1</td>
<td>0.8</td>
</tr>
<tr>
<td>% issues with relationship</td>
<td>89</td>
<td>50</td>
</tr>
</tbody>
</table>

is a lengthy process.

4.2.1 Projects Selected

I selected two projects Connect (data is reported from Dec 2012 to Nov 2014 inclusive) and Sonarqube (data is reported from Apr 2015 to May 2017) because both have an active development process, are programmed in Java, use a Jira issue repository and use a VCS. I chose projects whose code is expressed in Java to remain consistent with the projects used to evaluate the state-of-the-art techniques described in related work. A Jira repository is important to the selection because by default it has issue relationships, as explained in the introduction. A VCS is needed to have a history of commits to link to issues. Only the most recent two years are examined for both projects as it has the most recent representation of the process used by developers.

Table 4.1 shows statistics for each project, including the number of issues and commits. Developers in both projects manually linked over 60% of the commits to at least one issue by adding the issue id in the commit message. Connect has nearly three times as many relationships per issue as Sonarqube with 2.1 for Connect and 0.8 for Sonarqube. Both projects have issues with at least one relationship, 89% in Connect and 50% in Sonarqube.

To create the dataset, a subset of the data from both projects is used. To get an appropriate dataset for each project, a multi-day window is selected; for each repository at least 200 commits were selected, which is chosen as it is larger in

\footnote{Data after Nov 2014 is not used for the Connect project, as Connect changed to a private issue repository.}

\footnote{atlassian.com/software/jira, verified 13 July 2017}
Table 4.2: Comparison of relationship types for the window selected compared to the issue repository as a whole for both Connect and Sonarqube

<table>
<thead>
<tr>
<th></th>
<th>Connect</th>
<th>Sonarqube</th>
</tr>
</thead>
<tbody>
<tr>
<td>contributes to</td>
<td>4 (4%)</td>
<td>5 (0%)</td>
</tr>
<tr>
<td>depended</td>
<td>16 (3%)</td>
<td>84 (7%)</td>
</tr>
<tr>
<td>deprecated</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>duplicated</td>
<td>4 (1%)</td>
<td>5 (0%)</td>
</tr>
<tr>
<td>implements</td>
<td>1 (1%)</td>
<td>15 (1%)</td>
</tr>
<tr>
<td>related</td>
<td>33 (35%)</td>
<td>423 (30%)</td>
</tr>
<tr>
<td>replaces</td>
<td>1 (1%)</td>
<td>2 (0%)</td>
</tr>
<tr>
<td>subtask</td>
<td>142 (31%)</td>
<td>342 (29%)</td>
</tr>
<tr>
<td>superseded</td>
<td>0 (0%)</td>
<td>31 (2%)</td>
</tr>
<tr>
<td>supported by</td>
<td>300 (65%)</td>
<td>752 (64%)</td>
</tr>
</tbody>
</table>

size to those used by Wu and colleagues [56]. The selection also needs to be representative of the whole issue repository with respect to the percentage of each type of issue relationship. Table 4.2 shows the window selected compared to the latest two years of each project. A multi-day window of 58 days for Connect was used to get 204 commits and 20 days for Sonarqube to get 202 commits. These were the minimum window sizes necessary to get windows of 200 commits that were representative of the whole issue repository.

The multi-day window contains a set of commits from the git repository, using the master branch and all branches that have been merged into the master. The set of issues selected are all issues that were created, modified, or resolved within 24 hours before and after the multi-day commit window. 24 hours is chosen because a commit may occur within 24 hours of activity on an issue, according to research by Wu and colleagues [56]. For each issue, its related issues are added to the set of issues, because the related issues may be more specific to a commit with respect to consistency.

\(^{5}\) zxing had 143 commits and and openintents had 129 commits
4.3 Link Identification

In order to create the linking relationship for the dataset, I manually identified the links between commits and issues. To identify the linking relationship, the commits and issues are pre-processed to prepare the links to be manually examined. The next step is to create the link between the commit and issue. A set of guidelines is used to assist in identifying links that should be created. Issues were examined according to three filters, the filters were used to remove issue not likely relevant to the commit.

4.3.1 Preprocessing Dataset

The commits in the VCS contain links to issues that were manually stated by developers. To remove any bias that might be introduced by these stated links, a pre-processing step removes all references to git repositories in the text fields of issues, and removes all references to issue ids in commit messages.

References to a commit in the issue repository are either in the form of a URL linking to the code change in Github (i.e., github.com/CONNECT-Solution/CONNECT/pull/1335) or a hash id, which uniquely identifies the commit (i.e., 40 characters consisting of letters and numbers). The references to commits can occur in the text fields of the issues, such as the resolution (where developers put the description of how the issue was resolved), the comments, or the description. In order to identify and remove all commit references in issues, three patterns were used to remove references to the git repository:

- (\?)https://github.com/<project name>/pull/\d{1,3}
- (\?)https://github.com/<project name>/commit/[a-z0-9]{40}
- (?<=\s)[a-z0-9]{40}(?!\w)

The first two patterns search for URLs to Github for either a pull request or a commit. These are used by developers to identify what commit addresses the issue. The last pattern is to identify hash ids; it looks for 40 characters that are letters and number that starts with a space and doesn’t end with a letter or number. When any of the patterns are identified, the text is removed and replaced with the text commit reference redacted.
To remove issue ids from commit messages the same pattern used to identify stated links in Chapter [3.2.3] was used. The pattern used for each project is $(?i) <\text{projectname}>[\w^/w]*/d+$ the $<\text{projectname}>$is replaced with the name used for the issue id for each project, conn for Connect and sonar for Sonarqube. When any of the patterns are identified, the text is removed and replaced with the text ISSUE#.

4.3.2 Identification of Links

To create the links for a project, each commit in the selected window for a project is considered in order by of date, from the oldest to the most recent. For each commit, the goal is to identify one or more issues that should be linked. A set of guidelines are used to identify whether a commit should be linked to one or more issues. To limit the number of issues examined for each commit, filters are applied to the set of issues, which return a subset of the issues to examine.

The first step I used to identify a link between a commit and an issue is to understand the commit and to understand the issue. To understand the commit, I looked at the commit message, the names of files changed (including the complete path), and all additions or removals in each of the files changed to try to get an understanding of the purpose of the commit. If the commit message is vague (i.e., it does not refer to code changes), I used the previous commit for additional context (if it exists, and is similar in the files altered). The previous commit is the commit made immediately before the commit being inspected. To understand the issue, I looked at all of the text fields of the issue, including the issue title, description and comments.

Once I gained an understanding of the commit, I began to look for issues to link using the following guidelines:

- The meaning of the code changes in the commit, using either filenames, methods, variables, or features of the code, should reflect the meaning of the issue, based on the text in the issue

- The author of the commit should be either the person assigned to the issue, the reporter of the issue, have commented on the issue, or have made a change to the status of the issue.
• Time should be considered while selecting the issue to be linked: the issue should be resolved within seven days of the commit date for Connect and 12 days for Sonarqube. The time of resolution considered for each project is selected by looking at the average time between commit date and issue resolution of developer stated links.

• If two issues are similar and the commit should link to only one of the issues, then use other commits by the same author to determine to which issue the commit should be linked. Examples of other commits by the same author are commits close to the same date, or are commits that changed the same files.

To remove issue for consideration that are not likely relevant to the commit being examined, I used three different filters. The filters returned a subset of issues based on the rules for the filter. For each filter, I examined every issue returned by the filter to determine if the commit should link to the issue, or an issue that is related. After every issue, and related issue, returned by a filter is examined, the commit is linked to the issue or issues that are identified as being the most specific link in meaning to the commit. If no issue is identified from the list of issues returned by a filter, then the next filter is used. The filters are used in order:

1. The first filter orders the set of issues by the cosine similarity [47] of the words in the commit to the words in issues. The words from the commit are selected from the commit message, file names of the changed files, and the added lines of code in each file of a commit. Only code added in the commit is used to remove redundancy and false positives in cosine matching. The file names and added code are tokenized to extract words from camel casing and underscores, while removing java keywords and stopwords. The words in the issues are selected from the description, resolution, title, and comments. The filter returns a list of issues that have a cosine similarity of 0.2 or higher. Cosine similarity of 0.2 was chosen by selecting 50 random commits in the Apache Hive project[6] and identified the average cosine similarity between

---

the developer stated link to an issue. I found that the average cosine similarity was 0.28.

2. The second filter considers the time between the commit and the issue resolved date, and authorship, and is order by resolved date. The filter returns a list of issues where the author of the commit is the person assigned to the issue or the person who reported the issue. The filter selects all issues that are resolved a fixed time before and after a commit is made. The fixed time for Connect is seven days and Sonarqube is 12 days. Assignee and reporter are chosen as filters as these values are the most likely to represent the developers to make a change in a project. The days to filter for each project is selected by looking at average time between commit date and issue resolution of stated links.

3. The third filter is the same as the previous filter with respect to time, but is different with respect to authorship, and is order by resolved date. The filter returns a list of issues where the author of the commit comments on the issue or makes a change to the status of the issue.

This filter is to identify developers that may make a commit to address issues that they are not assigned. For example, a manager may add tests, or another developer makes an edit to update code for the same issue.

### 4.3.3 Dataset Validity

The links created using this method may be biased by the overall knowledge and experience of the person creating the links, and the lack of experience with the particular projects studied. To minimize this bias and ensure the most specific links are identified, I involved another researcher in the linking process. The first researcher identified links for the first 25 commits created using this method. Then, five random commits were selected from the first 25 commits and the other researcher independently applied the same process to link those commits to issues. The five random links were compared to the those identified by the first author and any conflict was discussed. This process was repeated for the first 100 commits. Thereafter, five out of every 50 commits linked were randomly selected and
Table 4.3: Results of linking process in identifying links commits to issues

<table>
<thead>
<tr>
<th></th>
<th>Existing Completeness %</th>
<th>Created Links Completeness %</th>
<th>Improvement %</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connect</td>
<td>69</td>
<td>85</td>
<td>23</td>
</tr>
<tr>
<td>Sonar</td>
<td>49</td>
<td>55</td>
<td>13</td>
</tr>
</tbody>
</table>

validated with the other researcher. This same process was used to validate both datasets Connect and Sonarqube.

Cohen’s kappa was used to measure agreement between the researchers and takes into account the agreements that might occur due to chance. The kappa between the two researchers was 0.82, and 0.67 for Connect and Sonarqube respectively. Using two researchers and Cohen’s kappa is standard in recent software engineering research [7, 30, 48, 58]. According to Landis and Koch, agreement between researchers with a kappa value from 0 to 0.2 is considered as slight, 0.21 to 0.4 is fair, 0.41 to 0.6 is moderate, 0.61 to 0.8 is substantial, and 0.81 to 1 is considered almost perfect [31].

The filters may have caused a researcher to miss an issue to which a commit should have been linked. To help mitigate this risk, the use of three filters is considered to identify different subsets of issues that may be linked to the commit being examined. Two of the filters were also adjusted for each project specifically to account for project specific processes, but still a potential link to an issue may have been missed with the method used.

4.4 Dataset

In both Connect and Sonarqube, there was an improvement to the completeness of the linking relationship in each dataset. Table 4.3 shows the percentage completeness for links stated by developers compared to the links created by myself and the other researcher. In Connect, 85% of the 204 commits were linked to an issue and in Sonarqube 55% of the 202 commits were linked to an issue. This improved completeness compared to developer’s stated links by 23% in Connect and 13% in Sonarqube.

An example in Connect dataset, I found a link between commit cba03ab to
issue GATEWAY-3435 that was not stated by the developer.

**Issue** GATEWAY-3435: use different “soap” namespace for UDDI server

**Commit** cba03ab: modified service port builder to use the new api defined in port-descriptor

Examining the code of commit cba03ab shows three lines in the Java file CXFServicePortBuilder are changed. The comments in issue GATEWAY-3435 discuss setting the code changes in CXFServicePortBuilder. In the end, this is a minor change, but the developer did not link the commit to the issue.

There was also a difference in the dataset compared to stated links with respect to consistency, which is used to identify if commits link to the most specific issues, of those issues that have a relationship. 52% of commits linked to an issue with a relationship in Connect and 34% in Sonarqube. Of the commits that were linked to an issue with a relationship, those links that were different than the stated by developers were 48% in Connect and 54% in Sonarqube.

An example of the difference in the Sonarqube project is the link commit 885033d to issue SONAR-6414.

**Issue** SONAR-6255: Move tests persistence from batch to compute

**Issue** SONAR-6414: Tests - Index DB and ES

**Commit** 885033d: index tests - SONAR-6255

The developer linked the commit to issue SONAR-6255 which is in a “Sub-task” relationship with the child issue SONAR-6414. Examining the code in the commit 885033d shows changes to elastic search (ES), and database (DB) test files. The commit 885033d should be linked to the more specific issue SONAR-6414, rather than the developer stated link to SONAR-6255.

### 4.4.1 Summary

The process of manually creating a dataset took over 212 hours for two researchers to complete. A dataset was created with a focus of making it both complete and
consistent by manually linking commits to issues without using developer’s stated links. The linking relationship in each dataset had an improvement in completeness by 23% and 13% in Connect and Sonarqube respectively. There was a difference with respect to the consistency of the linking relationship; 48% of the commits linked to a issue with a relationship in Connect and 54% in Sonarqube were different from the developer’s stated links. This dataset is available to other researchers who wish to have a more complete and consistent dataset to use for testing linking techniques.

\[\text{github.com/whitecat/linked-dataset} \text{ verified 13 July 2017}\]
Chapter 5

Semantic Relationships between Issues

A better understanding of how relationships are used in issue repositories and an ability to recognize different uses of relationships provides opportunities to create new, and improve existing, techniques to link commits to issues. For many software development projects, issue repositories hold key information defining what the system under development will do, who will work on different parts of the system, what defects occur as the system is being built, and more. When defining issues, software developers often expend manual effort to record relationships between issues, capturing such information as how work is to be broken down, how functionality in the system relates and which defects are similar to each other. When I examined the kinds of relationships in these repositories—by asking project developers on the forums they use and by analyzing documentation—I learned that the most frequently occurring relationships describe work breakdowns\(^1\), causing me to ask “how are software developers using work break down relationships in issue repositories” [51]?

To investigate this question, I performed a qualitative study of a sample of work breakdown relationships from the three repositories: Mylyn, Connect and

\(^1\) In Mylyn, 59% of the relationships are depends-on, which represent work breakdowns. In HBase, sub-tasks are work breakdowns representing 30% of all relationships specified. In Connect, 79% of relationships are work breakdowns via the supported-by and sub-task relationships.
HBase. I had three researchers (authors of the MSR paper [51]) code the how these relationships were being used based on an analysis of the titles of selected issues. Through this coding, we determined six codes that describe the kinds of work breakdown relationships, ranging from describing particular cases in which a more general problem must be solved to describing how functionality should be verified.

5.1 Qualitative Study

The qualitative study involved sampling pairs of related issues from the Mylyn, Connect and HBase issue repositories and performing an open coding of the sampled pairs.

5.1.1 Coding Process

I began by selecting pairs of issues related in work breakdown relationships from the Mylyn and Connect repositories. I chose these two systems to start the open coding process [50] because I had knowledge that they each follow an agile development process and thus might share commonalities in how they use relationships in the issue repository.

Three coders (the first, third and fourth authors of the MSR paper [51]) read the titles of each issue in a selected pair and discussed the meaning of the relationship between the pair. If the meaning had not yet been seen, a code was developed to recognize and describe how the issues are related and was recorded in a codebook.\(^2\)

In the first iteration, 40 issue pairs from Mylyn and 60 issue pairs from Connect were randomly selected and coded.

After coding the first 100 issue pairs, I randomly selected a different set of 60 issue pairs from Mylyn and 90 issue pairs from Connect. Each of the three authors involved in the original iteration then coded 2 sets of 20 issue pairs (for a total of 40) from Mylyn and 2 sets of 30 issue pairs (for a total of 60) from Connect. In this way, each set of 20 or 30 issue pairs was coded by two authors. The pairs of coders compared results and tried to reach a consensus on which code applies, updating the codes as necessary. To ensure the updated codes were appropriately

applied, all three coders then re-coded all previously coded pairs. At this point no new codes were found.

To determine if the codebook was sufficiently general to cover another system for which it had not been developed, two coders separately coded 50 issue pairs from the HBase repository, which was chosen as having different development characteristics from the other two projects. Based on this coding, some guideline refinements to clarify code selection were made to the codebook. The two coders then coded an additional 30 issue pairs from HBase to check if saturation had been reached. To assess the inter-coder reliability, I computed Cohen’s kappa on the final 30 issue pairs coded; the coders achieved a 0.56 kappa value. As will be explained in the next section, some codes are related through a hierarchy; coders sometimes had differences in the level of code in the hierarchy assigned. If all sub-codes are collapsed to the super-code in the hierarchy, the kappa scores rises to 0.88.

5.1.2 Codes

Six codes were identified from the coding process to more precisely describe the meaning of issues related through a work breakdown relationship: specification refinement, instance of parent, expectation, problem, check validity, and reverse specification. Pairs for which the meaning of the relationship could not be determined were coded as unknown. As Figure 5.1 shows, three of the codes are a specialization of the specification refinement code. Any pair of issues coded was assigned only one code from this set.

![Figure 5.1: Codes developed through open coding](image)

I describe the guidelines for each code in turn. For clarity in these guidelines,
I refer to the issue that is the source of the relationship as the *parent* and the issue that is the target of the relationship as the *child*.

**Specification refinement**  This code applies when a child issue describes one step of the work breakdown for the parent issue. The following example from *Mylyn* illustrates this code as the child issue specifies actions to take towards improving tooltip presentation.

**Parent**  205861: Improve tooltip presentation and content

**Child**  238292: Show reporter and beginning of description text on new issue tooltips.

**Instance of parent**  This code applies when each child issue is a particular case in which the work described by the parent issue should be performed. The following example from *Connect* illustrates a child issue that specifies work from the parent is to occur on Windows machines, the same work can be done also on other types of machines.

**Parent**  GATEWAY-1664: Create new VMs for final release installation testing.  
               Needed by Friday 3/9

**Child**  GATEWAY-1667: 4 Windows machines

**Check validity**  This code applies when a child issue describes a verification activity for a parent issue. For instance, in *HBase*, the child issue describes adding tests to show the feature, described by the parent issue, works correctly.

**Parent**  HBASE-10070: HBase read high-availability using timeline-consistent region replicas git

**Child**  HBASE-10791: Add integration test to demonstrate performance improvement
**Expectation**  This code applies when the child issue describes constraints or suggestions on how a parent issue can be fulfilled. The child issue in these cases often uses words like should, must, need, ensure, and improve. The following example from Mylyn shows how the child issue constrains the parent issue requirements.

**Parent**  158921: Improve the issue editor usability and information density  
**Child**  212953: Depends on field in issue editor should fill available horizontal space

**Problem**  This code applies when the child issue describes a problem that occurs in a parent issue. For instance, from Connect, the parent issue describes performing transaction logging and the child issue describes a particular part of the system requiring attention.

**Parent**  GATEWAY-2151: Transaction Logging  
**Child**  GATEWAY-2782: non-unique messageid causes transaction not to be logged in transaction repo

**Reverse specification**  This code applies when a parent issue describes one step of the work breakdown for the child issue. In other words, it is the reverse of specification refinement. For instance, from Connect the parent issue is a specific case of the child issue to investigate tests for concurrent messages.

**Parent**  CONN-910: Execute concurrent tests from 3.3 gateway to 4.3 to ensure turning off of replay attacks fixes the issue  
**Child**  CONN-859: Investigate and research issue when concurrent messages are sent from connect 3.x gateway to connect 4.2 gateway

**Unknown**  When none of the six codes just described apply to an issue pair, or when both reverse specification and specification refinement seem applicable, I consider the relationship meaning for an pair to be unknown. For instance, in the following Connect example, the parent issue describes an action to perform but the child issue is a noun phrase.
Table 5.1: Codes occurring in each repository

<table>
<thead>
<tr>
<th>Code</th>
<th>Mylyn</th>
<th>Connect</th>
<th>HBase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spec. Refinement</td>
<td>48.0</td>
<td>42.0</td>
<td>47.5</td>
</tr>
<tr>
<td>Check Validity</td>
<td>4.0</td>
<td>14.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Instance of Parent</td>
<td>5.0</td>
<td>21.3</td>
<td>7.5</td>
</tr>
<tr>
<td>Expectation</td>
<td>14.0</td>
<td>3.3</td>
<td>21.3</td>
</tr>
<tr>
<td>Problem</td>
<td>22.0</td>
<td>6.7</td>
<td>10.3</td>
</tr>
<tr>
<td>Reverse Spec.</td>
<td>3.0</td>
<td>2.7</td>
<td>0.0</td>
</tr>
<tr>
<td>Unknown</td>
<td>3.0</td>
<td>7.3</td>
<td>6.3</td>
</tr>
<tr>
<td>No consensus</td>
<td>1.0</td>
<td>2.7</td>
<td>1.3</td>
</tr>
<tr>
<td>Total</td>
<td>100</td>
<td>150</td>
<td>80</td>
</tr>
</tbody>
</table>

**Parent**  CONN-1094: Create static screens for Direct configuration in Administrative GUI

**Child**  CONN-1105: Trust Bundles

### 5.1.3 Threats to Validity

The codes may be biased by the knowledge and experience of the coders. The use of three coders helps minimize this bias. By separating into pairs to code after the development of the initial code book, I helped mitigate the persuasive affect of any one coder.

With each repository coded, the coders clarified the code book. More refinements may be necessary if applied to other repositories, limiting the external validity of the results. Coding only the title of an issue also limits the results. More clarity on how a relationship is used might have been gained by using more information from the issue, such as comments about the work actually undertaken as part of the issue.

### 5.1.4 Results

Table 5.1 shows the results of coding 330 issue pairs across the three repositories. The most prevalent code across all repositories was specification refinement. Interestingly, the more specialized version, check validity,
occurs much more often in Connect than in the other repositories; perhaps the other repositories do not explicitly record their quality assurance related tasks. Mylyn contains more issue pairs describing problems than the other repositories; this may be due to the high number of issue reporters who are not contributing developers. The expectation code occurs more frequently in HBase, perhaps because developers perform more analysis of work breakdowns before specifying child tasks. The instance of parent occurs more frequently in Connect, suggesting that the developers more frequently refer to structural parts of the system when specifying work breakdown issues.

5.2 Summary

Developers often expend manual effort to specify how issues in an issue repository relate, especially to express how work is to be broken down and performed on the system. To investigate what kinds of work breakdowns are being expressed, in conjunction with three colleagues, we performed an open coding of a sample of 330 related issue pairs from the issue repositories of three open source systems: Mylyn, Connect and HBase. The open coding progress resulted in six codes that describe a variety of kinds of work breakdowns, including cases where the work breakdowns express steps of verification and express constraints on work to be performed.

This study is the first to provide insight into the richness of information embedded in relationships in issue repositories. This information offers new opportunities to link commits to issues.
Chapter 6

Discussion and Future Work

An understanding of how issue relationships are used may help inform the development of new tools and may help improve the linking of commit and issue data.

6.1 Improving Links of Commits to Issues

Despite the rich amount of relationship information contained in an issue repository, none of the existent techniques use that information to create or refine links. State-of-the-art techniques to improve this link information might use relationships to further improve link information.

For example, the technique introduced by Schermann and colleagues introduces heuristics that reduce the number of unlinked issues in two ways [45]. The first heuristic technique “Phantom” creates links between issues and commits, finding multiple commits that belong to a single issue using existing links. The second heuristic technique “Loaner” links single commits with no issue id to link to an issue with no existing link. Figure 1.1 provides an example of how relationship information might improve these heuristics. The work by Sherman and colleagues, describes the “phantom” commit 14cd8d6b3 should be linked to the issue CAMEL-7354 [45] (see Figure 1.1a). However CAMEL-7354\(^1\) has seven subtasks and an inspection of subtasks shows commit 14cd8d6b3, should be linked to one of the sub-tasks, CAMEL-7675\(^2\). Introducing relationship informa-

\(^1\)issues.apache.org/jira/browse/CAMEL-7354 verified 13 July 2017
\(^2\)issues.apache.org/jira/browse/CAMEL-7675 verified 13 July 2017
tion, such as relationship category, could improve linking techniques by refining the existing links by moving commits to the correct level of work breakdown (see Figure 5.1).

6.2 Techniques to Categorize Issue Relationships

An indication of the semantic relationship between two issues may help further improve techniques to improve linking commits to issues. The assignment of semantic codes to relationships in this thesis is referred to as “categorizing”, building on the manual work described in Chapter 5.1. As developers do not benefit directly from the categorizing relationships, future work may consider automatic categorization of work breakdown relationships.

6.2.1 Algorithm

The issues in issue repositories are entered largely manually by developers to describe work to be performed and problems encountered with the system. As a result of this manual entry, the natural language used to describe the issues varies extensively. Two steps may help provide an automated approach to categorizing issue relationships. First, the language found in issues could be transformed to a more restricted set to identify the similarities in how issues relate. Second, the work breakdown issue pairs in an issue repository could be automatically categorized using the similarities identified and supplementary materials, such as information about the structure of the system.

Algorithm for Transformation

Nasukawa and colleague found transforming large amounts of textual data into a smaller dataset assisted in knowledge discovery [34]. Such an approach may help in categorizing relationships. The language in the issue repository could be transformed using a restricted language created with verbs, code terms, and noun phrases found in the issue repository, in combination with other sources such as web sites and word net corpuses. Then, the restricted language could be used to transform the title of each issue. The output of the transformation could be one or more fields of issues in the issue repository augmented with data containing a
restricted language.

A concrete example of transformation by restricting the language in titles is shown in Table 6.1. In this example, the titles use the set of restriction rules, replacing words with higher level structure which groups terms using different algorithms.

Transforming language, by restricting the set of words found in issue text, can assist a categorizer in identifying how issues relate, though increasing similarities. A group of transformation algorithms could be necessary which focus on reducing the variety of text across an issue repository. The input to these algorithms can be word graphs of title fields generated by NLP, which includes stemming and the grammatical relationships between words [14]. The algorithms were envisioned after creating the semantic codes in Chapter 5, which were chosen by looking at what changed between two titles, and what action is taking place.

The rationale for each algorithm is described in turn:

**Code elements**: This algorithm finds text that refers to elements of code, as developers may refer to specific code elements when doing work. This can be implemented using work by Bacchelli and colleagues who found camel casing and non English dictionary words were an effective indicators of code elements in text [3]. The output of the algorithm replaces code elements with the word *code*.

**Concept**: The goal of this algorithm is to replace project specific program concepts with the word *concept*. The algorithm can be implemented with the assistance of concept location techniques. Multiple techniques to implement concept
location exist such as latent semantic indexing, formal concept analysis, program
dependence analysis, and term frequency. This can be implemented using work
done by Poshvyvanyk and colleagues [37] which uses latent semantic indexing on a
source code corpus with a query. The query for this algorithm is a noun phrase,
and the source code is the version control system related to the issue repository.

**Version:** This algorithm can find noun phrases referring to different versions of
software in an issue repository, for example “Connect 3.1” specifies the 3.1 version
of connect. Using pattern matching version numbers can be found in a noun phrase.
The output of this algorithm replaces noun phrases with the word *version*.

**External system:** This algorithm finds references to external libraries or sys-
tems in the title of issue repositories. Using Stackoverflow tags a list of libraries
and systems can be created. Nouns in text matching a tag would be replaced with
the word *system*.

**Verb grouping:** This algorithm minimizes the number of different verbs in text.
To do this all verbs in an issue repository are clustered using DBSCAN cluster-
ing algorithm [20] and similarity measures between verbs using a word similarity
database. The clusters can be labeled with the most occurring verb in the clus-
ter, verbs are replaced using the cluster label. Multiple word similarity databases
exist such as freebase [12], VerbNet [46], and WordSimSE DB[52]. The WordSimSE
database is used for this algorithm as it is tailored to software engineering words.

**Algorithm for Categorization**

Using the restricted language, a categorizer algorithm could be created, based on
supervised machine learning. The algorithm would consider issue pairs that are
gathered the same way as in Chapter 5. A categorizer could be trained using infor-
mation in each issue pair, such as restricted language and authorship. The catego-
rizer could then be used to categorize all work breakdown issue pairs in an issue
repository.

For instance, an algorithm might learn when an issue in a pair has a title with
the word test is always categorized as check validity. In Figure 6.1, the
categorizer takes the issue pair [GATEWAY-1911, GATEWAY-1996] applies the

---

3 [www.stackoverflow.com; verified 13 July 2017]
algorithm and labels the pair as check validity.

\[\text{Uncategorized}\]

(Issue pair) \{GATEWAY-1911: *component* provide *version* GATEWAY-1996: test for GATEWAY-1911\}

\[\text{Categorized}\]

(Issue pair) \{GATEWAY-1911, GATEWAY-1996, check validity\}

**Figure 6.1:** Example of categorizing with a supervised machine learning algorithm that learned to label issue pairs as check validity if the child issue the contains the word *test*.

Supervised machine learning can be used if categorization of issue relationships is framed as a text categorization problem, which assigns a category to each document in a set from a list of possible categories [25]. For this approach, an issue pair is considered a document and the list of categories comes from the semantic categories discussed in Chapter 5.

**Features for Machine Learning**  Machine learning algorithms need features as input. The features come from two areas of an issue ownership, and transformed title. The transformed title is used because there is more regularity between titles, and the increased regularity potentially increases patterns found using machine learning. Ownership is another important feature to include because work by Hindle and colleagues [23] finds ownership an important feature in categorizing large issues. Features are extracted individually from both issues in an issue pair. Features are explained as follows:

- The *transformed title* is a brief explanation of work to be completed for an issue, using the restricted language. The feature is a vector using frequency of terms in text. The vector is normalized using inter-document frequency, intra-document frequency and based on document length [41].

- The *verb group in issue text* is the action used to describe work needed to
complete an issue. This feature is the verb group of the first verb in the issue title.

- **Code elements in issue text** are references to where changes are, or might be made to complete an issue. This feature is the existence of code elements in the issue title.

- **Sentences that start with noun/verb/neither** is an indicator of the type of work to be performed. The feature is whether a text field in an issue starts with a noun, verb, or neither.

- **Ownership of issue** is the person who is assigned to complete an issue. The feature is which person is assigned to complete the issue.

Each of these features could be extracted from both issues in an issue pair and labeled whether it relates to the start or end node of the issue pair. These could then be used as input to the categorizer.

**Potential Machine Learning Algorithms** Future work could investigate two supervised machine learning algorithms: Naive Bayes, and Support Vector Machine (SVM). Naive Bayes algorithm is of interest because it is possible to implement as an online algorithm, which can simulate being updated as issues are entered into an issue repository. Support vector machine is an obvious algorithm to investigate because it is tailored to do text categorization, and can reduce the need of labeled training data [54]. Recent work by Rahoman and colleagues supports using SVM as a machine learning technique to categorize text [38].
Chapter 7

Conclusion

The goal of this thesis is to improve the quality of software process data in the form of links between commits and issues. There is no standard approach to assess the quality of the links between commits and issues. I introduced the quality attributes of consistency and completeness as means to assess the quality of the existing links between commits and issues. Completeness addresses whether all commits link to an issue in the issue repository. Consistency is used to measure if all commits link to the most relevant issue in the issue repository. I found existing techniques that aim to improve links between issues and commits produced results that do not fully address completeness and consistency.

To help future researchers investigate new techniques for improving link data between commits and issues, I created a dataset consisting of a portion of two open source projects. I found the manually created dataset had an improvement of 23% and 13% in Connect and Sonarqube respectively with respect to completeness when compared to the data entered in the original systems by developers. With respect to consistency, the dataset had many commits which were linked to an issue with a relationship change, 48% in Connect, and 54% in Sonarqube were different from the developer’s stated links. Future researchers can use this dataset to create a technique to link commits to issues.

Relationships that exist between issues have not been considered in research to improve software process data. Issue relationships hold information about how work is broken down in a system or how the functionality in a system is related.
One problem that exists with the relationships between issues is the ambiguity among the different meanings of relationship types which may lead to developers to mislabel a relationship. I explored the different kinds relationship types in issues in three repositories: Mylyn, Connect, and HBase, and found work breakdown relationships were the most common among all three issue repositories. I then conducted a study to identify the underlying meanings of issues that have work breakdown relationships. The study found six primary underlying meanings for work breakdown relationships.

Future researchers can build on these results to build and test enhanced approaches to improve link data between commits and issues.
Bibliography


[58] A. Zagalsky, C. G. Teshima, D. M. German, M.-A. Storey, and G. Poo-Camano. How the r community creates and curates knowledge: a


Appendix A

Links examined for consistency
Table A.1: Links examined for consistency evaluation in Chapter 3. No changes to the links were made by Phantom or Loaner.

<table>
<thead>
<tr>
<th>Project</th>
<th>hash_id</th>
<th>Existing Link</th>
<th>Relink</th>
<th>RCLink</th>
<th>Most specific Link</th>
</tr>
</thead>
<tbody>
<tr>
<td>CB</td>
<td>dca79</td>
<td>CB-11921</td>
<td></td>
<td>CB-11917, CB-9656</td>
<td></td>
</tr>
<tr>
<td></td>
<td>67ca4a4</td>
<td>FLINK-3200</td>
<td></td>
<td>FLINK-3201</td>
<td></td>
</tr>
<tr>
<td></td>
<td>77eb4f0</td>
<td>FLINK-3207</td>
<td></td>
<td>FLINK-3201</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ceb6424</td>
<td>FLINK-3435</td>
<td></td>
<td>FLINK-3201</td>
<td></td>
</tr>
<tr>
<td></td>
<td>d353895</td>
<td>FLINK-2314, FLINK-3717, FLINK-3889</td>
<td></td>
<td>FLINK-3889, FLINK-3717</td>
<td></td>
</tr>
<tr>
<td></td>
<td>dec0d6b</td>
<td>FLINK-4410</td>
<td></td>
<td>FLINK-4697</td>
<td></td>
</tr>
<tr>
<td>Flink</td>
<td>025b3e1</td>
<td>PUBDEVE-2922</td>
<td></td>
<td>PUBDEV-395, PUBDEV-3667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>041bf21</td>
<td>PUBDEVE-3664</td>
<td></td>
<td>PUBDEV-395, PUBDEV-3667</td>
<td></td>
</tr>
<tr>
<td></td>
<td>162c909</td>
<td>PUBDEVE-1843</td>
<td></td>
<td>PUBDEV-2746</td>
<td></td>
</tr>
<tr>
<td></td>
<td>36cf05e</td>
<td>PUBDEVE-3482</td>
<td></td>
<td>PUBDEV-3791</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4aa59cb</td>
<td>PUBDEVE-2729</td>
<td></td>
<td>PUBDEV-3791</td>
<td></td>
</tr>
<tr>
<td>h2o_3</td>
<td>1cb2e92</td>
<td>MNG-6140</td>
<td></td>
<td>MNG-6140</td>
<td></td>
</tr>
<tr>
<td></td>
<td>94bc4de</td>
<td>MNG-6093</td>
<td></td>
<td>MNG-6093</td>
<td></td>
</tr>
<tr>
<td></td>
<td>b80915b</td>
<td>MNG-3507</td>
<td></td>
<td>MNG-3507</td>
<td></td>
</tr>
<tr>
<td>maven</td>
<td>1aa67f7</td>
<td>NUTCH-2221</td>
<td></td>
<td>NUTCH-2144</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5784b64</td>
<td>NUTCH-3664</td>
<td></td>
<td>NUTCH-2144</td>
<td></td>
</tr>
<tr>
<td></td>
<td>f22f2ed</td>
<td>NUTCH-1233</td>
<td></td>
<td>NUTCH-1233</td>
<td></td>
</tr>
<tr>
<td>nutch</td>
<td>ee561dc</td>
<td>PDI-14132</td>
<td></td>
<td>PDI-14800</td>
<td></td>
</tr>
<tr>
<td></td>
<td>410d77c</td>
<td>PDI-14132</td>
<td></td>
<td>PDI-14800</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2715a07</td>
<td>SONAR-7634</td>
<td></td>
<td>SONAR-8241</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2e2d8c7</td>
<td>SONAR-8221</td>
<td></td>
<td>SONAR-8241</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3a56577</td>
<td>SONAR-8028</td>
<td></td>
<td>SONAR-8241</td>
<td></td>
</tr>
<tr>
<td></td>
<td>399038d</td>
<td>SONAR-7970, SONAR-7986</td>
<td></td>
<td>SONAR-7973</td>
<td></td>
</tr>
<tr>
<td></td>
<td>c06807d</td>
<td>SONAR-8120</td>
<td></td>
<td>SONAR-8166</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ec1eb14</td>
<td>SPR-14680</td>
<td></td>
<td>SPR-14865</td>
<td></td>
</tr>
<tr>
<td></td>
<td>994caca</td>
<td>SPR-14680, SPR-14865</td>
<td></td>
<td>SPR-14865</td>
<td></td>
</tr>
<tr>
<td>spring_framework</td>
<td>770f0c0</td>
<td>SPR-13495</td>
<td></td>
<td>SPR-13495</td>
<td></td>
</tr>
<tr>
<td></td>
<td>bc14e5b</td>
<td>SPR-14521</td>
<td></td>
<td>SPR-13495</td>
<td></td>
</tr>
<tr>
<td></td>
<td>59c88eb</td>
<td>SPR-13475</td>
<td></td>
<td>SPR-13486, SPR-13973</td>
<td></td>
</tr>
</tbody>
</table>