Recovering Traceability Links between Source Code and Fixed Bugs via Patch Analysis

Christopher S. Corley
Dept. of Mathematics & Computer Science
University of North Alabama
Florence, Alabama, USA
cscorley@una.edu

Nicholas A. Kraft
Department of Computer Science
The University of Alabama
Tuscaloosa, Alabama, USA
nkraft@cs.ua.edu

Letha H. Etzkorn
Department of Computer Science
The University of Alabama in Huntsville
Huntsville, Alabama, USA
etzkorl@uah.edu

Stacy K. Lukins
Department of Computer Science
The University of Alabama in Huntsville
Huntsville, Alabama, USA
slukins@cs.uah.edu

ABSTRACT

Traceability links can be recovered using data mined from a revision control system, such as CVS, and an issue tracking system, such as Bugzilla. Existing approaches to recover links between a bug and the methods changed to fix the bug rely on the presence of the bug's identifier in a CVS log message. In this paper we present an approach that relies instead on the presence of a patch in the issue report for the bug. That is, rather than analyzing deltas retrieved from CVS to recover links, our approach analyzes patches retrieved from Bugzilla. We use BugTrace, the tool implementing our approach, to conduct a case study in which we compare the links recovered by our approach to links recovered by manual inspection. The results of the case study support the efficacy of our approach. After describing the limitations of our case study, we conclude by reviewing closely related work and suggesting possible future work.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*documentation, version control*

General Terms

Documentation, Management, Measurement

Keywords

Traceability, link recovery, trace automation, mining software repositories, bug assignment, bug mapping

1. INTRODUCTION

Traceability links between software artifacts such as code and documentation are needed to perform various software evolution tasks. For example, traceability links between fixed bugs and changed source code elements are required to construct defect prediction models [13, 17]. Such traceability links also have value as research instrumentation, because they are needed to evaluate new techniques that address problems such as bug localization [23]. However, even in the presence of a rigid bug fixing process, lack of integration between the revision control system (e.g., CVS) and the issue tracking system (e.g., Bugzilla) makes documentation of traceability links difficult [3]. In response to this problem, researchers have developed approaches that use bug identifiers present in CVS log messages to recover missing traceability links [3, 5, 12, 14, 18, 29]. To complement these approaches, we have developed an approach that recovers traceability links between fixed bugs and affected methods via analysis of patches stored in the issue tracking system.

Ayari et al. [3] report on their investigation of threats and difficulties associated with integrating CVS and Bugzilla repositories. They note that existing approaches to recovering traceability links for fixed bugs begin with a search for (potential) bug identifiers in CVS log messages and that not all fixed bugs can be traced using these log messages. In particular, each fixed bug in a Bugzilla repository belongs to one of the following sets:

B_{log}	The bug ID is found in a CVS log message but no patch is available in Bugzilla.
B_{patch}	The bug ID is not found in a CVS log message but a patch is available in Bugzilla.
B_{both}	The bug ID is found in a CVS log message and a patch is available in Bugzilla.
B_{none}	The bug ID is not found in a CVS log message and no patch is available in Bugzilla.

The four sets are mutually exclusive.

Existing approaches to recovering traceability links are applicable to fixed bugs in sets B_{log} and B_{both}, but not to the fixed bugs in B_{patch}. Yet, Ayari et al. [3] report that for the Mozilla browser, over 17,000 (nearly 20%) of the fixed bugs belong to B_{patch}, whereas less than 10,000 (about 10%) of the fixed bugs belong to B_{log}. Indeed, a link recovery tool with added support for B_{patch} would be applicable to over 56% of the 92,858 fixed bugs for Mozilla (note that over 25,000, or nearly 28%, of the fixed bugs are in B_{both}).

TEFSE'11, May 23, 2011, Waikiki, Honolulu, HI, USA

The findings of Ayari et al. [3] suggest that a significant number of fixed bugs belong to B_{patch} and thus that an automated approach to recovering traceability links for these fixed bugs is necessary for correct and efficient performance of software evolution tasks. We have developed such an approach, and in this paper we present its design, implementation, and evaluation. BugTrace, the tool implementing our approach, analyzes patches available in Bugzilla and is applicable to fixed bugs in B_{patch} and B_{both} (as well as certain fixed bugs in B_{log} and B_{none}).

Our approach is similar to that of Eaddy et al. [12]. Their tool, BugTagger, takes as input a bug ID found in a CVS (or SVN) log message, uses the bug ID to retrieve the associated delta(s) from CVS, and analyzes the delta(s) to recover traceability links. BugTrace takes as input a Bugzilla repository and finds issue reports with status RESOLVED/ VERIFIED/CLOSED and resolution FIXED and with at least one non-obsolete patch attached.Next, BugTrace retrieves the patches associated with each identified issue report, and analyzes the patches to recover traceability links.

The key advantage of our approach is that patches are often more fine-grained than CVS commits. That is, a patch, particularly one that has been verified by a quality assurance (QA) team, is likely to contain only the changes required to fix the bug. On the other hand, CVS commits commonly include changes unrelated to the bug [6, 27] or changes required to fix multiple bugs. The key disadvantage of our approach is that its accuracy is more reliant on the quality of information in the issue tracking repository than are other approaches. Unfortunately, issue tracking repositories for open source projects are known to contain incorrect metadata (e.g., enhancements classified as bugs [3, 12, 29]).

We evaluate our approach by comparing traceability links recovered automatically, via BugTrace, and manually, via inspection by two of the authors, for over 300 fixed bugs in the Bugzilla repositories for Eclipse and Rhino. Our analysis of the results reveals a number of human errors, but also a number of scenarios that are intractable to our tool. Overall, the results support the efficacy of our approach in increasing the quantity of recovered traceability information for fixed bugs in B_{patch} and suggest that its combination with existing approaches may improve the quality of recovered traceability information for fixed bugs in B_{both}.

2. BACKGROUND

In this section we describe the life cycle for a Bugzilla issue report, focusing on the stages of the life cycle that are most relevant to our work.

A user creates an issue report upon discovering a new bug (or formulating a new enhancement request). Upon creation the issue report has status UNCONFIRMED. A quality assurance (QA) engineer (in the open source community, this is typically a project maintainer) evaluates the issue report and updates the status to NEW if the bug is confirmed (or the enhancement approved) or ASSIGNED if the bug is assigned to a developer. Once development related to the bug is complete, the status is changed to RESOLVED and the resolution is set to one of seven possible values according to the outcome of development. For our work, the relevant resolution is FIXED. Issue reports are often abandoned after reaching status RESOLVED and resolution FIXED. However, a QA engineer should verify the changes and update the status to VERIFIED or CLOSED.

An issue report undergoes several changes during its life. The reporting user provides a summary of the bug (which Bugzilla stores in the `short_desc` field) and the details of the bug (which Bugzilla stores in a `long_desc` field). After the user submits the issue report, Bugzilla assigns it a bug ID. Subsequent viewers of the issue report (including developers and other users) may add comments (which Bugzilla stores in additional `long_desc` fields). Typical comments may include questions about the initial summary and details, or new information about the bug, such as additional ways in which the bug can be triggered. Comments may also contain fixes for the bug. Those fixes may be described in natural language or in source code, and may be included in the comment field or be included as an attachment. Each attachment in Bugzilla has an ID, a description, a file name, and a set of flags that indicate its status. Two of the flags are relevant to our work: `is_obsolete`, which indicates whether the attachment has been withdrawn or superceded, and `is_patch`, which indicates whether the attachment is a patch.

Ideally, for every patch uploaded to Bugzilla as an attachment, `is_patch` is true, and for every attachment with `is_patch` set to true, the patch is: valid (e.g., is not a '.java' file), verified (i.e., has been verified by a QA engineer), and current (unless `is_obsolete` is true). Unfortunately, patches are sometimes provided as comments, attached patches do not always have the correct flags set, and `is_patch` is true for some attached source files and zip archives. Moreover, in the ideal scenario, when a developer commits a bug fixing change to a CVS repository, he includes 'bug *ID*' in the CVS log message (for each bug fixed in the commit, where the ideal number of bugs fixed per commit is one). However, not all CVS log messages for commits including bug fixing changes list the corresponding bug ID.

3. APPROACH

In this section we describe the design and implementation of BugTrace, the tool implementing our approach. BugTrace takes as input a Bugzilla repository, the corresponding CVS repository, and (optionally) a bug ID of interest. The output of BugTrace is a set of recovered traceability links. Each link associates a bug ID with a method modified (added, changed, or removed) to fix the bug.

3.1 Patch Retrieval

BugTrace begins the patch retrieval phase by searching a Bugzilla repository for issue reports with status RESOLVED/ VERIFIED/CLOSED and resolution FIXED. For each report that meets this criteria, BugTrace extracts the bug ID and builds a list containing the ID of each attachment for which `isobsolete` is false and `ispatch` is true. If the resulting list is non-empty, BugTrace retrieves the patches from the Bugzilla repository and passes them to the patch analysis phase.

If BugTrace fails to identify at least one non-obsolete patch, it builds a list containing the ID of each attachment for which `isobsolete` is false. If the resulting list is non-empty, BugTrace retrieves the attachments from the Bugzilla repository and attempts to parse each one as a patch. BugTrace discards any attachment for which parsing fails, and passes the remaining attachments (patches) to the patch analysis phase.

If BugTrace has yet to retrieve a patch, it searches the comments of the issue report in reverse order, attempting to identify and extract an embedded patch. If a patch is identified, BugTrace passes it to the patch analysis phase. Otherwise, patch retrieval fails.

The two fallback options (considering attachments not marked as patches and searching comments) are not critical to the approach. However, using these options BugTrace can recover traceability links for fixed bugs in B_{log} and B_{none}. We exercised both options in our case study, and in each instance BugTrace recovered correct traceability links. Nevertheless, these fallback options could potentially cause BugTrace to recover significant numbers of incorrect links (false positives). An extensive study of fixed bugs for which these options are triggered is needed to evaluate their impact on the accuracy of BugTrace.

3.2 Patch Analysis

In this phase each retrieved patch is analyzed. A patch comprises one or more diffs, and each diff is in context format or unified format. BugTrace splits a patch into its constituent diffs, each of which lists changes to a single file, and analyzes the diffs individually. When analyzing a diff, BugTrace first extracts the CVS path of the file from which the diff was generated. This extraction is simple when analyzing a diff generated using `cvs diff`, because such diffs include the full CVS path of the file. However, we found that with minimal information about a project's file structure, BugTrace reliably extracts correct file paths from a diff, whether it is generated manually using `diff` (against a CVS checkout) or automatically using `cvs diff` (against the CVS server or a local mirror).

After extracting the CVS path of the file, BugTrace determines the timestamp of the original file (i.e., the date and time at which the diff was generated). In the special case that a diff introduces a new file, BugTrace instead determines the timestamp of the updated file. BugTrace handles three different timestamp formats that we observed in our case study. Once BugTrace has the date and time at which the diff was generated, it searches the information returned by `cvs log` for the file of interest to determine the CVS revision number of the file at the given date and time.

BugTrace uses the CVS revision number discovered in the previous step to checkout the file revision from which the diff was generated (i.e., the revision of the file containing the bug). To validate that it has the correct file revision, BugTrace compares the context lines from the diff — unchanged or removed lines from the original file that are included in the diff to provide context — to the corresponding lines in the file. Note that the corresponding lines are determined using the line range provided by the diff. For our case study, we found matching three context lines from the end of the diff to be sufficient for validation. We expect lines at the end of the diff to be least likely to match an incorrect file revision.

If validation of the checked out file revision fails, BugTrace checks out subsequent file revisions until a revision is validated successfully, or until the HEAD revision of the CVS repository is reached. In the latter case, BugTrace checks out prior file revisions until a revision is validated successfully, or until the initial revision of the CVS repository is reached. BugTrace passes all successfully validated files for a patch to the link recovery phase.

3.3 Link Recovery

In this phase BugTrace uses a patch and the original files (the files from which the patch was generated) to recover traceability links between a bug and the methods modified (added, changed, or removed) to fix the bug. First, BugTrace parses the original files, storing the following information for each method: the method signature, the start line, and the end line. The method signature consists of the qualified method name, which includes the names of enclosing classes and methods (but not the name of the enclosing package), and the parameter types. Next, BugTrace applies the patch to the original files to obtain the patched files, and then parses the patches files, storing the previously described information for each method.

BugTrace uses the stored method information for the original and patched files to detect the set of added methods and the set of removed methods. In particular, given the set of methods in the original files, M_o, and the set of methods in the patched files, M_p, the set of added methods, M_a, is defined as $M_a = M_p - (M_o \cap M_p)$. Similarly, the set of removed methods, M_r, is defined as $M_r = M_o - (M_o \cap M_p)$. Because we use method signatures to identify methods, a change to a method signature is detected as a method removal and a method addition.

To detect the set of changed methods, BugTrace inspects the patch, storing the line numbers of lines that begin with `-` or `!`. For lines that begin with `+`, BugTrace stores the line number of the first immediately preceding line that does not also begin with `+`. Note that all line numbers are computed as offsets from the start line provided by the current diff. BugTrace uses the stored line numbers and the stored method information for the original files to detect the set of changed methods, M_c. The final set of methods modified to fix the bug, M, is defined as $M = M_a \cup M_r \cup M_c$. For each method $m \in M$, BugTrace recovers a traceability link between the bug and m.

3.4 A Final Note on Implementation

During link recovery we compute and store a line range for each method, and we chose to exclude the last line of the method (the line containing '`}`') from this line range. We made this decision to avoid false positives caused by certain diffs. For example, the diff in Figure 1 shows that a new method, `bar`, has been added between the existing methods `foo` and `baz`. If we include the last line of a method in its line range, we will detect a change to `foo`. To avoid this false positive, we exclude the last line of each method from consideration. Our decision can result in false negatives (examples of which are described in Section 4.3), but we believe the trade-off is appropriate.

```
1  @@ -1,6 +1,9 @@
2      public class Example {
3          public void foo() {
4              int i;
5  +       }
6  +       public void bar() {
7  +           int j;
8          }
9          public void baz() {
10             int k;
```

Figure 1: Example of a Problematic Diff

Table 1: Rhino Results

Version	Bugs	$\|L_i\|$	$\|L_b\|$	$\|L_a\|$	$\|L_d\|$
1.4R3	4	6	5	5	2
1.5R1	7	31	35	29	8
1.5R2	3	16	9	6	13
1.5R3	10	31	31	31	0
1.5R4	13	87	114	84	33
1.5R5	32	321	491	289	234
1.6R1	13	42	45	41	5
1.6R2	6	18	18	18	0
1.6R3	1	1	3	1	2
1.6R4	12	95	107	90	22
1.6R6	1	1	1	1	0
1.6R7	2	6	6	5	2
Total	104	655	866	600	321

(a) High Level Comparison

Version	$\|D_0\|$	$\|D_1\|$	$\|D_2\|$	$\|D_3\|$	$\|D_4\|$
1.4R3	1	0	0	1	0
1.5R1	2	0	0	5	1
1.5R2	10	0	0	2	1
1.5R3	0	0	0	0	0
1.5R4	0	3	1	29	0
1.5R5	29	3	1	134	67
1.6R1	1	0	0	3	1
1.6R2	0	0	0	0	0
1.6R3	0	0	0	2	0
1.6R4	3	1	1	8	9
1.6R6	0	0	0	0	0
1.6R7	1	0	0	1	0
Total	47	7	3	185	79

(b) Categorized Disagreements

4. CASE STUDY

In this section we describe the case study that we conducted to evaluate the accuracy of our approach. We compare the traceability links recovered by BugTrace to the traceability links recovered by manual inspection, noting any disagreements. We then analyze the results.

4.1 Data Examined

The data examined includes over 300 issues from the Bugzilla repositories for Eclipse[1] and Rhino[2]. Each issue has status RESOLVED/VERIFIED/CLOSED and resolution FIXED. Most issues have at least one non-obsolete patch as an attachment. However some issues either have a patch embedded in a user comment, or have a non-obsolete attachment that contains a patch (but is not marked as such). We previously used these issue reports to evaluate a bug localization technique based on latent Dirichlet allocation [23].

Antoniol et al. [1] classify reported issues as bugs or non-bugs, where "bugs" are requests for corrective maintenance and "non-bugs" are requests for other activities, such as perfective or adaptive maintenance. We examined 203 issues for Eclipse, all of which are bugs, and 104 issues for Rhino, some of which are bugs and some of which are non-bugs (though for ease of explication, we hereafter refer to all examined issues as bugs). The Eclipse bugs span 13 versions (from 3.0 to 3.4) and the Rhino bugs span 12 versions (from 1.4R3 to 1.6R7).

4.2 Procedure

For each of the 307 bugs, we have two sets of traceability links: those recovered manually by inspection (L_i) and those recovered automatically by BugTrace (L_b). Each set contains only links between the bug and the methods modified (added, changed, or removed) to fix the bug. That is, the sets do not include links between the bug and source code elements such as a classes. A link is expressed as a tuple with a bug ID as the first entry and a method signature as the second entry.

[1]`http://www.eclipse.org`
[2]`http://www.mozilla.org/rhino`

We first compare the corresponding sets at a high level, noting the number of links in each set ($|L_i|$ and $|L_b|$), the number of links agreed upon ($|L_a|$ where $L_a = L_i \cap L_b$), and the number of links for which there is a disagreement ($|L_d|$ where $L_d = (L_i \cap L_b)^c$). We next compare the disagreements between corresponding sets at a low level. We inspect each disagreement, placing it into one of five sets:

- D_0 Incorrect links in L_i (false positives from manual inspection)
- D_1 Correct links in L_i (false negatives from BugTrace)
- D_2 Incorrect links in L_b (false positives from BugTrace)
- D_3 Correct links in L_b (false negatives from manual inspection)
- D_4 Task mismatch

We indicate that a link results from a task mismatch in cases where BugTrace recovers a link that was intentionally ignored during manual inspection. That is, the two authors who performed the manual inspection were to "identify the methods modified (added, changed, or removed) to fix the bug." However, the patches for some bugs include modifications to unit test methods, and BugTrace accurately recovered from those patches links between the bug and the unit test methods. Because such links could have been recovered by the two authors, or filtered/ignored by BugTrace, we classify them as a task mismatch.

4.3 Results

Rhino.

Table 1 lists the results for Rhino. Across the 12 versions of Rhino, BugTrace recovered an average of 8.33 links per patch. Further, BugTrace recovered 866 links to 740 unique method signatures, and 600 of these links exactly matched a link recovered via manual inspection. Columns 3 and 4 of Table 1b list the number of false negatives and false positives in L_b, respectively. These results demonstrate that BugTrace is accurate for the 104 Rhino bugs.

Columns 2 and 5 of Table 1b indicate unusually high numbers of false positives and false negatives in L_i for the 45 bugs

Table 2: Eclipse Results

Version	Bugs	$\|L_i\|$	$\|L_b\|$	$\|L_a\|$	$\|L_d\|$
3.0	13	53	88	48	45
3.0.1	12	25	44	19	31
3.0.2	6	11	12	11	1
3.1	18	106	260	94	178
3.1.1	15	29	46	27	21
3.1.2	12	24	44	19	30
3.2	16	71	138	65	79
3.2.1	20	77	117	68	58
3.2.2	19	70	181	68	115
3.3	20	310	704	301	412
3.3.1	19	36	81	24	69
3.3.2	14	50	77	48	31
3.4	19	91	146	55	127
Total	203	953	1,938	847	1,197

(a) High Level Comparison

Version	$\|D_0\|$	$\|D_1\|$	$\|D_2\|$	$\|D_3\|$	$\|D_4\|$
3.0	4	1	0	33	7
3.0.1	2	4	0	11	14
3.0.2	0	0	0	1	0
3.1	12	0	0	46	120
3.1.1	0	2	1	4	14
3.1.2	3	2	0	4	21
3.2	4	2	0	28	45
3.2.1	9	0	0	36	13
3.2.2	2	0	0	12	101
3.3	7	2	0	91	312
3.3.1	12	0	0	44	13
3.3.2	2	0	0	4	25
3.4	36	0	0	40	51
Total	93	13	1	354	736

(b) Categorized Disagreements

in Rhino 1.5R4 and 1.5R5. The authors who recovered the links in L_i listed, for each recovered link, the bug ID and the method signature, in which they included the method name qualified by the name of one containing class. On the other hand, for each recovered link, BugTrace lists the bug ID and the complete method signature, including the fully qualified method name. For example, for bug 49350 the `equals` method of class `ClassSignature`, which is nested in class `JavaAdapter`, is changed. In this case L_b includes a link to method `JavaAdapter.ClassSignature.equals`, whereas L_i includes a link to method `ClassSignature.equals`. Because the method name used by the link in L_i may be ambiguous, we judge the link to be a false positive (and the missing correct link is a false negative).

Modifications to methods in class `BodyCodegen` (in Rhino 1.5R5) resulted in several more false negatives being added to L_i. The source file `Codegen.java` contains two top-level classes: `Codegen` (which has public visibility) and `BodyCodegen` (which has package visibility). All links recovered via manual inspection for methods in `Codegen.java` list the containing class as `Codegen`, but most of the modified methods actually reside in class `BodyCodegen` (which BugTrace correctly notes). Misspellings/typos are a third common source of human errors in L_i for Rhino.

As described in Section 3.1, BugTrace can parse diffs embedded in an issue report's user comments. This functionality is exercised by Rhino bugs 38384, 252122, 253323, and 302501. In all cases BugTrace recovers correct links from the embedded diffs.

Eclipse.

Table 2 lists the results for Eclipse. Across the 13 versions of Eclipse, BugTrace recovered an average of 9.55 links per patch. Further, BugTrace recovered 1,938 links to 1,926 unique method signatures, and 847 of these links exactly matched a link recovered via manual inspection. Columns 3 and 4 of Table 2b list the number of false negatives and false positives in L_b, respectively. As with the Rhino results, these results demonstrate that BugTrace is accurate for the 203 Eclipse bugs.

BugTrace misses a traceability link (has a false negative) for bug 133738 due to the implementation decision described in Section 3.4. In particular, the method changed by the patch is structured such that the last line contains both a statement and the '`{`' which closes the method body. The patch adds new lines to the end of the method and moves the '`{`' to its own line. The combination of the method's structure, the new lines being added to the end of the method, and our decision to exclude the last line of a method from the line range computation causes BugTrace to miss the change (and thus to miss the traceability link).

5. THREATS TO VALIDITY

Our study has limitations that impact the validity of our findings, as well as our ability to generalize them. In this section we describe some of these limitations and discuss their impact on our study.

BugTrace relies on the accuracy of the bug metadata stored in a Bugzilla repository. Unfortunately, issue tracking repositories for open source projects are known to contain incorrect metadata [3, 12, 29]. Our reliance on potentially flawed metadata is a threat to the internal validity of our study. This threat is shared by other studies that analyze bugs by mining software repositories, (e.g., [8, 23, 26]).

Another threat to internal validity relates to the use of patches to recover links between bugs and methods. For example, a patch may include modifications unrelated to the bug [6, 27]. However, we believe that patches are less likely to exhibit this property than are the CVS commits used by related approaches (e.g., [3, 5, 12, 14, 18, 29]). Thus, we believe that our approach mitigates this threat in relation to approaches that use CVS commits to recover links.

A third threat to internal validity arises from our approach to identifying the correct CVS revision of an original file (i.e., the revision of a file containing the bug) during patch analysis. BugTrace uses context lines from the patch to validate the identified file revision, but few context lines are included in each patch, and BugTrace may successfully validate an incorrect revision of the file. However, our approach's accuracy, as demonstrated by a comparison to human data and as verified by manual inspection of any disagreements, suggests that BugTrace's validation process may be sufficient.

External validity is the degree to which general conclusions can be drawn from our results. We studied two Java projects, so we cannot generalize our results to projects implemented in other languages. Moreover, both projects use CVS for revision control and Bugzilla for issue tracking, which may limit the applicability of our results to projects using other repositories such as Subversion (for revision control) or Jira (for issue tracking).

6. RELATED WORK

Our work is closely related to research on automated traceability link recovery and mining software repositories. Overviews of these research areas are available. Spanoudakis and Zisman [30] provide a roadmap for software traceability research, while Cleland-Huang et al. [7] provide an overview of best practices for automated traceability. Kagdi et al. [19] provide an overview of approaches for mining software repositories, and other work by Kagdi and his collaborators [20, 21] describes how software repositories support recovery of traceability links.

As described in Section 1, there are many approaches to merging data from revision control systems, such as CVS, and issue tracking systems, such as Bugzilla. A common application of this merged data is automatic recovery of traceability links [3, 5, 12, 14, 18, 29]. Once recovered, these links can be used to address a number of problems, including: change impact analysis [4], defect modeling [12], defect prediction [18], and experimental validation [23].

Many approaches have been proposed for automatic recovery of traceability links between source code and documentation. Antoniol et al. [2] address this problem using a vector space model, and subsequent work by Marcus and Maletic [24] substitutes LSI for the vector space model. De Lucia et al. [11] add traceability link generation and management to ADAMS [10], an process support system which emphasizes the artifact life cycle. Their extension to ADAMS is based on LSI. More recent work by McMillan et al. [25] combines textual and structural analysis to recover links indirectly. In particular, McMillan et al. combine LSI and JRipples, a structural analysis tool based on an evolving interoperation graph (EIG) [28], to construct a traceability link graphs (TLG) and to infer new traceability links from the TLG.

Merging data from multiple software repositories is another area of active research. Gall et al. [15] detect evolution patterns in CVS repositories. Using module revision numbers, they assess growth, identify changes in behavior, and detect common changes patterns across modules in object-oriented programs. Ying et al. [31] mine change history to detect logical couplings. They compute association rules between files to predict future source code changes. Similarly, Zimmerman et al. [32] describe an approach to detection of change couplings. They predict future source code changes by detecting causal couplings between entities such as classes, methods, or fields. German [16] retrieves and analyzes "modification records" that fix bugs. He detects co-changing files and identifies the developers most likely to modify particular files. Finally, recent work by De Lucia et al. [9] combines traceability information with information retrieval to help developers improve the source code lexicon for a project.

7. CONCLUSIONS AND FUTURE WORK

We presented an automated approach to traceability link recovery. Specifically, our approach merges data from a revision control system and an issue tracking system and uses the merged data to recover links between bugs and the methods modified to fix the bugs. We implemented our approach as a tool, BugTrace, that mines and merges data from a CVS repository and a Bugzilla repository. Further, we evaluated our approach by comparing links recovered by BugTrace to links recovered by two of the authors using manual inspection. The results indicate that BugTrace produces few false positives and few false negatives for the 307 bugs in our test suite. We also noted limitations of our approach and sources of human error that we discovered when verifying our results.

Future work includes comparing the accuracy of BugTrace to the accuracy of tools that use identifiers present in CVS log messages to recover missing traceability links. Whereas BugTrace uses patches retrieved from Bugzilla to recover traceability links, other approaches use deltas retrieved from CVS. A comparison of links recovered for fixed bugs in B_{both} by BugTrace and another tool would help to determine how frequently the use of CVS commits increases false positives (i.e., how frequently CVS commits include changes unrelated to the bug). Other future work includes an expanded study using projects implemented in multiple programming languages.

Broader plans for future work include investigating models and tools for managing links between bugs and methods. In addition, integrating research results related to program differencing (e.g., [22]) could benefit our approach and others like it. For example, the ability to reliably detect method renamings from mined repository data would permit the recovery of correct traceability links between the original method and the renamed method. In turn, these new links would improve the quality of models (such as defect prediction models) that rely on rich traceability information.

8. ACKNOWLEDGMENTS

This paper is based upon work supported by the National Science Foundation under Grant No. 0851824. We thank the anonymous reviewers for their helpful suggestions.

9. REFERENCES

[1] G. Antoniol, K. Ayari, and M. Di Penta. Is it a bug or an enhancement? A text-based approach to classify change requests. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 1–15, 2008.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. Recovering traceability links between code and documentation. *IEEE Trans. Softw. Eng.*, 28(10):970–983, 2002.

[3] K. Ayari, P. Meshkinfam, G. Antoniol, and M. Di Penta. Threats on building models from CVS and Bugzilla repositories: the Mozilla case study. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 215–228, 2007.

[4] G. Canfora and L. Cerulo. Impact analysis by mining software and change request repositories. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, page 29, 2005.

[5] G. Canfora and L. Cerulo. Where is bug resolution knowledge stored? In *Proceedings of the 3rd International Workshop on Mining Software Repositories*, pages 183–184, 2006.

[6] G. Canfora, L. Cerulo, and M. Di Penta. On the use of line co-change for identifying crosscutting concern code. In *Proceedings of the International Conference on Software Maintenance*, 2006.

[7] J. Cleland-Huang, R. Settimi, E. Romanova, B. Berenbach, and S. Clark. Best practices for automated traceability. *Computer*, 40(6):27–35, 2007.

[8] D. Cubranić, G. Murphy, J. Singer, and K. Booth. Hipikat: A project memory for software development. *IEEE Trans. Softw. Eng.*, 31(6):446–465, 2005.

[9] A. De Lucia, M. Di Penta, and R. Oliveto. Improving source code lexicon via traceability and information retrieval. *IEEE Trans. Softw. Eng.*, 2010. doi:10.1109/TSE.2010.89.

[10] A. De Lucia, F. Fasano, R. Francese, and R. Oliveto. ADAMS: An artifact-based process support system. In *Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering*, pages 31–36, 2004.

[11] A. De Lucia, F. Fasano, R. Oliveto, and G. Tortora. Recovering traceability links in software artifact management systems using information retrieval methods. *ACM Trans. Softw. Eng. Methodol*, 16(4), 2007. doi:10.1145/1276933.1276934.

[12] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, and A. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, 2008.

[13] M. English, C. Exton, I. Rigon, and B. Cleary. Fault detection and prediction in an open-source software project. In *Proceedings of the 5th International Conference on Predictor Models in Software Engineering*, 2009. doi:10.1145/1540438.1540462.

[14] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from version control and bug tracking systems. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 1–10, 2003.

[15] H. Gall, K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *Proceedings of the 19th International Conference on Software Maintenance*, pages 190–197, 1998.

[16] D. German. An empirical study of fine-grained software modifications. *Empirical Software Engineering*, 11(3):369–393, 2006.

[17] T. Graves, A. Karr, J. Marron, and H. Siy. Predicting fault incidence using software change history. *IEEE Trans. Softw. Eng.*, 26(7):653–661, 2000.

[18] T. Gyimóthy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, 2005.

[19] H. Kagdi, M. Collard, and J. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 19(2):77–131, 2007.

[20] H. Kagdi and J. Maletic. Software repositories: A source for traceability links. In *Proceedings of the 4th ACM International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 32–39, 2007.

[21] H. Kagdi, J. Maletic, and B. Sharif. Mining software repositories for traceability links. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 145–154, 2007.

[22] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319, 2009.

[23] S. Lukins, N. Kraft, and L. Etzkorn. Bug localization using latent Dirichlet allocation. *Information and Software Technology*, 52(9):972–990, 2010.

[24] A. Marcus and J. Maletic. Recovering documentation-to-source-code traceability links using latent semantic analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–134, 2003.

[25] C. McMillan, D. Poshyvanyk, and M. Revelle. Combining textual and structural analysis of software artifacts for traceability link recovery. In *Proceedings of the 5th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 41–48, 2009.

[26] D. Poshyvanyk, T. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Softw. Eng.*, 33(6):420–432, 2007.

[27] R. Purushothaman and D. E. Perry. Toward understanding the rhetoric of small source changes. *IEEE Trans. Softw. Eng.*, 31(6):511–526, 2005.

[28] V. Rajlich and P. Gosavi. Incremental change in OO programming. *IEEE Software*, 21(4):62–69, 2004.

[29] J. Śliwerski, T. Zimmerman, and A. Zeller. When do changes induce fixes? In *Proceedings of the 2nd International Workshop on Mining Software Repositories*, pages 1–5, 2005.

[30] G. Spanoudakis and A. Zisman. Software traceability: A roadmap. In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering*, pages 395–428. World Scientific Publishing Co., 2005.

[31] A. Ying, G. Murphy, R. Ng, and M. Chu-Carroll. Predicting source code changes by mining change history. *IEEE Trans. Softw. Eng.*, 30(9):574–586, 2004.

[32] T. Zimmerman, P. Weisgerber, S. Diehl, and A. Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, 2004.