Branching and Merging in the Repository

Chadd Williams
Pacific University
2043 College Way
Forest Grove, OR 97116
+1 503-352-3041
chadd@pacificu.edu

Jaime Spacco
Colgate University
13 Oak Dr
Hamilton, NY 13346
+1 315-228-7650
jspacco@mail.colgate.edu

ABSTRACT
Two of the most complex operations version control software allows a user to perform are branching and merging. Branching provides the user the ability to create a copy of the source code to allow changes to be stored in version control but outside of the trunk. Merging provides the user the ability to copy changes from a branch to the trunk. Performing a merge can be a tedious operation and one that may be error prone. In this paper, we compare file revisions found on branches with those found on the trunk to determine when a change that is applied to a branch is moved to the trunk. This will allow us to study how developers use merges and to determine if merges are in fact more error prone than other commits.

Categories and Subject Descriptors
D.2.7 [Distribution, Maintenance, and Enhancement]: Version control – *Restructuring, reverse engineering, and reengineering*.

General Terms
Management, Measurement, Documentation, Design.

Keywords
Repository, mining, Subversion, diff, change.

1. INTRODUCTION
Branching the source code and then merging changes applied to the branch back to the trunk is a very difficult operation. The developer must determine how changes from a branch interact with a trunk that may have been heavily modified since the branch was created. Given such a tedious task, we want to determine if merges are especially error prone when compared to other commits in the repository. Unfortunately this information is not tracked by version control systems and developers are inconsistent about marking merges in commit messages.

Branches may be used for many different purposes in the repository but are typically used to shield the trunk from unstable code. Some projects use branches to work on bug fixes, only merging back changes once the bug is fixed. Other projects use branches for adding new features, keeping unfinished code away from the trunk. Still other projects use branches to represent a release of the software which allows maintenance work to continue without intrusion from the main line development.

The goal of the paper is to identify when and where merges take place. Specifically, we are looking for *parallel file revisions*. These are pairs of file revisions, one on a branch and one on the trunk, where the content of the changes applied is the same. This is useful for identifying places where *some* changes made to a branch are applied to the trunk. This is the first step to identifying merges in the repository.

The method presented here relies heavily on a deep syntactic analysis of source code changes and on having a strong line number mapping algorithm that allows us to track specific lines of code. The syntactic change analysis allows us to determine that the content of two file revisions is the same. The line number mapping between versions allows us to ensure that changes are applied to the same lines in each version of the file.

2. IDENTIFIYING SYNTACTIC CHANGES
The deep syntactic analysis we perform on the source code changes is done using the DiffJ tool [9]. This is an open source tool which will perform a Java-aware diff, as opposed to a textual diff, between two source files. The set of changes produced by DiffJ describe the changes made in terms of the Java syntax. Samples of the set of changes DiffJ will identify are listed in Table 1. In addition to line number information provided by every DiffJ change type, certain DiffJ changes provide extra context data as noted in the table.

We did modify the DiffJ tool to provide an API interface rather than running it via the command line. Our changes also added some additional context information to the changes.

Table 1: Sample of DiffJ Change Types

Change Type	Context Data
importAdded	Name of imported package
methodAdded	Name of added method and class
parameterTypeChanged	Old and new types
codeChange	None
codeRemoved	None

MSR'08, May 10–11, 2008, Leipzig, Germany.

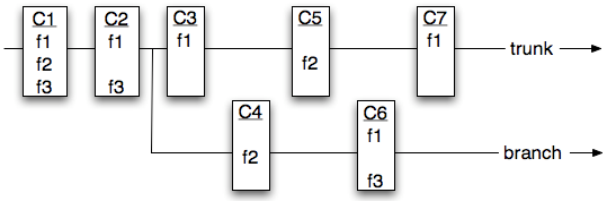

Figure 1: Commits on the Trunk and Branch

3. LINE MAPPING

We use a line-mapping algorithm to track unique source lines as they evolve across many revisions. Our work is based on the line number matching scheme described by Spacco, et al., in [10] and Canfora, et al. in [4].

The line number mapping is done on a per-method and per-class basis. For this we use the normalized Levenshtein [8] edit distance (true edit distance / maximum possible edit distance between the two lines based on the length of each line) between the lines of source code in the method or class in each version of the file to provide weights for the edges of a bipartite graph. We then find a minimum weight bipartite matching (using the Kuhn-Munkres algorithm) to determine the best mapping between the two versions of the code.

The Kuhn-Munkres algorithm finds the best mapping for all sets of lines [7]. This means that it may map two significantly different lines to each other when there is no better match available. To combat this, we store the Levenshtein edit difference for every pair of mapped lines. When using the line mapping data we only consider mappings where the normalized Levenshtein edit distance is sufficiently low (less than 0.4 as in [4]).

4. FINDING SIMILAR FILE REVISIONS

Similar file revisions are defined as file revisions that have the same, or nearly the same, types of changes at the same line numbers in two different versions of the same file. It should be noted that we are looking for similar file revisions, not similar commits. A *file revision* is a change to a single file; a *commit* is a set of file revisions made at the same time by the same author. Figure 1 shows repository commits (C#) containing file revisions to individual files (f#) on the trunk and branch.

4.1 Algorithm

The goal of this analysis is to identify pairs of file revisions, one from a branch and one from the trunk, applied to a file that contain the same, or nearly the same, set of changes. First the *change profile* of each file revision is determined. The change profile is simply the number of changes per type of syntactic change in a file revision. Each file revision on each branch is then compared with all file revisions to the same file on the trunk. Pairs of file revisions with similar change profiles are further inspected by comparing individual syntactic changes. Syntactic changes are compared using contextual information (as described in Table 1) when it is available, and through the line number mapping if the contextual information is missing or ambiguous.

The change profiles for file f2 in commit C4 and C5 are shown in Figure 2. All but one change, *code changed in m3(),* in C5 can be matched (via their change type) with a change in C4. This

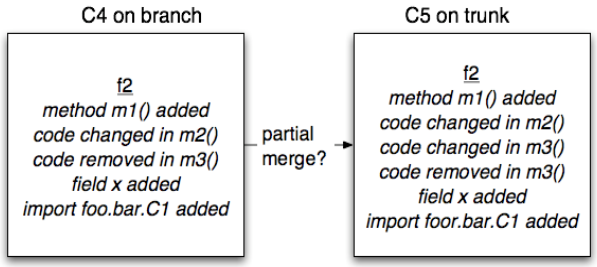

Figure 2: Parallel File Revision, Detail

indicates that this pair of file revisions is a potential parallel file revision and a deeper analysis is done. The details of the comparison are described below.

4.1.1 Change Analysis

Consider Figure 3, where we see changes to file f1 with similar profiles, in commit C6 on a branch and also C7 on the trunk. Upon closer inspection of the context information provided by DiffJ, we see that, while both C6 and C7 change some code in method m1() and add a method, C6 adds method m2() while C7 adds method m3(). Additionally, C6 changes the return type of method m4(). Thus these two revisions are unlikely to be parallel file revisions.

There are situations where the context information returned by DiffJ is insufficient to determine whether two change profiles are the same; for example, *parameterTypeChanged* contains context information that denotes the old and new type of the parameter but does not specify the method or parameter involved. In these situations, we use the line number mapping information to ensure that two revisions actually touch the same lines of a file.

4.1.2 Line Number Mapping Analysis

Each contextually ambiguous change in the file revision on the branch is compared with each similarly typed change from the file revision on the trunk to determine if they are changes to the same line(s) of source code. To do this checking, the line number mapping information from section 3 is used. The lines involved in the change on the branch are traced back through previous versions until the line's ancestor on the trunk is found. If the line was created on the branch, its history will terminate before the trunk is reached. In this case the change on the branch cannot match to a change in the trunk file revision. Once the line's ancestor on the trunk is found, the line is traced forward on the trunk until the file revision is question is reached. If the line is deleted from the trunk (or altered as to become unrecognizable to the line mapper) before the trunk file revision is reached the change on the branch cannot match the change on the trunk. If the line does have a history that extends to the file revision in

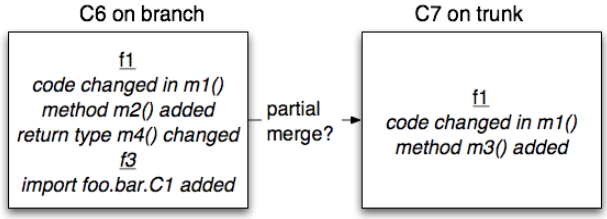

Figure 3: False Parallel File Revision

Similarity = (totalChanges – unpairedChanges) / totalChanges

Confidence = (totalChanges – unpairedChanges – matchedOpaqueChanges) / totalChanges

Figure 4: Metrics

question on the trunk, then the similarly typed changes in the trunk file revision are checked to determine any of them contain this line. If the line is found to be involved one of these changes, then a match is made and the next change is investigated.

4.1.3 Metric

The result of this analysis is the *Similarity* metric. This calculation is shown in Figure 4. This value represents the percent of the total changes in the two file revisions that are matched successfully. This gives a normalized value between 0 and 1.

4.2 Opaque Changes

Unlike ChangeDistiller [5], DiffJ does not perform any analysis of the changes to the abstract syntax trees (ASTs) of method bodies other than identifying places in the method where code was added or changed, resulting in the *opaque* change types *codeChange* and *codeAdded*. A change type is classified as *opaque* if it does not provide enough syntactic data to determine if two changes of that type contain the same change to the code. The difficulty is that even if we can use the line-number mapping information to infer that pairs of these opaque changes occur on the same line, we still don't have enough syntactic information to determine if the same change is being applied to each line or if two different changes are being applied. Figure 5 demonstrates this problem. The two changed lines originate from the same source line in they trunk. However, the line in the trunk has had an extra change applied (C2). Even though the changes in C3 and C4 are the same, the resulting lines are different and we cannot easily determine that the changes are the same.

To compensate for this, a further metric is calculated, *confidence*. The *confidence* metric takes into account the number of opaque changes (*codeChange* or *codeAdded*) that are matched successfully. The *confidence* metric is calculated in a similar manner to *Similarity*, however all of the *codeChanges* and *codeAddeds* are assumed to have *not* matched. This gives a worst case confidence value. This calculation is shown in Figure 4.

A pair of file revisions containing opaque changes is shown in Figure 2. Assume that the line numbers involved in all changes match, except for the *codeRemoved* (which cannot possibly match since it only happens in one file revision). The similarity for this pair of file revisions is 0.91, as 10 of 11 changes match. The confidence, however, is only 0.72, since we automatically assume the two *codeChanges* do *not* match (8 of 11).

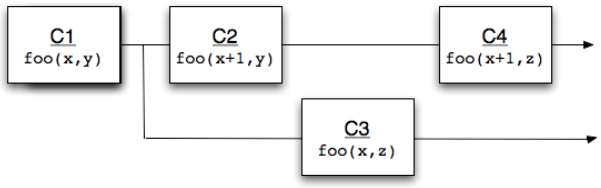

Figure 5: Opaque Change

5. RESULTS

We performed our analysis on the ArgoUML repository [1]. We converted the initial ArgoUML repository from CVS to Subversion using the cvs2svn converter tool [2], which recovers the atomic commits [11]. The developers of ArgoUML have converted their CVS repository to Subversion using the same cvs2svn converter. All new work in the ArgoUML project is stored in this converted Subversion repository

The changes we have mined span from January 1998 to December 2005. In that time there were 9,284 commits made to the repository which contained 54,510 file revisions. This repository contained 70 branches plus the main trunk of development. Of the 70 branches, 49 were branched from the trunk, 16 were branched from other branches, and 6 were created as an artifact of the CVS to Subversion conversion. Most of the branches contain a small number of commits. Only 15 of the 70 branches contain 10 or more *commits*, however 45 branches contain 100 or more *file revisions* and only 15 branches had 50 or fewer file revisions.

5.1 Parallel File Revisions

In the ArgoUML repository we found 386 pairs of file revisions that have a *confidence* metric of 0.75 or higher. We will focus on these file revisions which touch 24 commits on the trunk and 52 commits on a branch. All but one of these file revision pairs was a change on a branch that was later applied to the trunk.

The average time between similar file revisions was found to be 7.4 days; the median time between similar file revisions was 6.1 days. Interestingly, the pair of file revisions with the largest gap, 123.3 days, appears to be a very similar file revision, both are adding the method *getContainer*() to the same class. Each file revision also appears to be the result of a bug report (issue 1722 and issue 2128, respectively).

5.2 Suspected Merges

We inspected all 23 trunk commits that matched with a commit that was first made on a branch. Of these, 16 of the commits had a commit message that reported the commit was the result of a merge. Two of the commits had a commit message stating that the changes fixed a particular issue number from the bug database, where that issue number was used as the name of a branch. One other commit claimed to be an upgrade of *antlr* which appears to be a merge of branch devoted to *antlr* work.

We also identified a number of parallel file revisions with a *confidence* of less than 0.75. These pairs of file revisions touched 34 commits on the trunk. We inspected these file revisions and found that an additional 4 had commit messages that denoted a merge and another 10 mentioned the fix of a particular issue number from the database that had a branch named after it.

6. RELATED WORK

Canfora et al [4] describe an algorithm for tracking unique line numbers across many versions of software. Their algorithm uses Levenstein edit distance to compute similarity of lines and matches "hunks" of changed code between two versions of code. A similar approach to tracking lines was also described in [10].

Clone detection is similar to tracking unique lines in that both try to identify segments of code where, although not exactly the same, one code segment has evolved from the other. Kim et al provide an excellent overview of the four basic approaches to

clone detection [6], all of which rely on the analysis of higher-level language constructs than lines of source code, such as program dependence graphs or abstract syntax trees.

Zimmerman et al [12] describe an algorithm for tracking source lines across versions to discover lines that are frequently changed together, changed by different authors, or changed most frequently. Their approach does not use edit distance to try to match individual lines of a modified chunk, but rather assumes that any modified line in version one of a file can be matched with any modified line in version two of a file. Large modifications are ignored in order to make the analysis more tractable.

Godfrey at al describe "origin analysis" [13], a general technique for tracking entities across multiple revisions of a source code base. The key idea behind origin analysis is to store inexpensively computed and cheaply comparable "fingerprints" of interesting software entities for each revision of a file. These fingerprints can then be used first to identify areas of the code that are likely to match before applying more expensive techniques for tracking an entities, such as ASTbased clone detection techniques..

7. FUTURE WORK

We need to look further at inspecting the opaque changes *codedChanged* and *codeAdded* at a deeper level. This will allow our analysis to more accurately determine which file revisions contain the same changes. This will also allow us to remove the *confidence* metric and only use the *Similarity* measure. This may mean moving to a tool that gives more fine-grained results than DiffJ currently does.

The current analysis works only on the file revision level. This identifies instances when a large revision is made on the branch and a similar revision is made on the trunk. This, however, is not the only expected merge behavior. It may be the case that the branch contains many small revisions to a file and the sum of those revisions is merged to the branch at once. If each of the individual revisions to the branch are small, and there are many of them causing the commit to the trunk to be large, either no parallel file revision will be found for the trunk revision or the *confidence* will be very low. To properly identify this case, we need to look at the sum of the revisions on the branch and compare that sum to the revision made on the trunk. In that scenario, we would not be relying on the fact that the branch contains a few large revisions that are easily matchable to the trunk. The difficulty of determining the sum of changes to a branch arises from the fact that the same line in a file may be changed multiple times in a branch before the correct fix is found. This will require a deeper analysis of the changes on the branch rather than a simple summation.

The normalized nature of the *Similarity* and *confidence* metric allows parallel file revisions of different sizes (different number of changes) to be compared. However, this also has the effect of making a pair of file revisions with a single matching change in each file revision appear to be a very strong match (*similarity* of 1). We need to investigate further if file revisions with more change types indicate a better match.

So far we have only applied this technique to one open source project, ArgoUML. We will need to apply this technique to a wide range of projects to gain a better understanding of how branches and merges are used in the wild.

This analysis currently only looks at similar file revisions between branches and the trunk. We also would like to study similar file revisions between two branches. Software version control packages allows branches to be created from the trunk or other branches and so it will be instructive to how many similar file revisions can be found on different branches.

8. ACKNOWLEDGMENTS

Our thanks to Jeff Pace for making DiffJ available and useful and the reviewers for their helpful comments.

9. REFERENCES

[1] ArgoUML, http://argouml.tigris.org, 2007.

[2] CVS to Subversion Repository Converter, http://cvs2svn.tigris.org 2007.

[3] T. Apiwattanapong, A. Orso, and M. J. Harrold. Jdiff: A differencing technique and tool for object-oriented programs. *ASE Journal of Automated Software Engineering*, (1):3–36, March 2007.

[4] G. Canfora, L. Cerulo, and M. D. Penta. Identifying changed source code lines from version repositories. In MSR '07: *Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 14, Washington, DC, USA, 2007. IEEE Computer Society.

[5] B. Fluri and H. C. Gall. Classifying change types for qualifying change couplings. In ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), pages 35–45,Washington, DC, USA, 2006. IEEE Computer Society

[6] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An empirical study of code clone genealogies. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 187–196, New York, NY, USA, 2005. ACM.

[7] Kuhn, H., "The Hungarian Method for the assignment problem", *Naval Research Logistics Quarterly*, 2:83-97, 1955.

[8] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. *Cybernetics and Control Theory*, (10):707–710, 1966.

[9] J. Pace. A tool which compares java files based on content. http://www.incava.org/projects/java/diffj, 2007.

[10] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In MSR '06: *Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM.

[11] C. Williams and J. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. IEEE Trans. Softw. Eng., 31:466–480.

[12] T. Zimmermann, S. Kim, A. Zeller, and J. E. James Whitehead. Mining version archives for co-changed lines. In MSR '06: *Proceedings of the 2006 international workshop on Mining software repositories*, pages 72–75, New York, NY, USA, 2006. ACM.

[13] L. Zou. Using origin analysis to detect merging and splitting of source code entities. IEEE Trans. SoftwEng., 31(2):166–181, 2005. Member-Michael W.Godfrey.