Identifying Source Code Reuse across Repositories using LCS-based Source Code Similarity

Naohiro Kawamitsu*, Takashi Ishio*, Tetsuya Kanda*, Raula Gaikovina Kula*, Coen De Roover*†, Katsuro Inoue*

* Graduate School of Information Science and Technology
Osaka University
1–5 Yamadaoka, Suita, Osaka 565–0871, Japan
Email: {n-kawamt, ishio, t-kanda, raula-k, coen, inoue}@ist.osaka-u.ac.jp

† Software Languages Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussels, Belgium
Email: cderoove@vub.ac.be

***Abstract*—Developers often reuse source files developed for another project. In order to update a reused file to a newer version released by the original project, developers have to track which revision of a file was reused and how its content was modified. However, such tracking is tedious for developers. Many projects keep older versions of files whose bugs are already fixed in the original project. In this paper, we propose a technique to automatically identify source code reuse relationships between two repositories. Using a similarity metric based on longest common subsequence, we identify pairs of similar revisions of files across the repositories. To evaluate our approach, we have analyzed eight project pairs of open source software projects and compared the result with the recorded information in the repositories. As a result, we have identified 1394 file revisions as instances of source code reuse. While 75.3% of the instances are recorded in the repositories, 20.1% of the instances are unrecorded but recovered by our approach.**

I. Introduction

Clone-and-own is one of the popular approaches to source code reuse [1][2]. Developers copy source files developed by another project into their own. Reusing existing libraries reduces cost and enables quick software development. This even happens for libraries which are intended to be shared between projects. Rather than linking against a compiled version of the library, developers sometimes copy its source code into the project's. For instance, when a project-specific enhancement is required.

Developers should keep track of the library version they copied from. This facilitates keeping copies up-to-date. For instance, to patch newly discovered security vulnerabilities. Nevertheless, 18.7% of the projects studied in [3] had no records of the actual versions of the third-party code that they are reusing. A simple `diff` is often insufficient to identify the origin of a copy, because it cannot distinguish project-specific changes from updates to the original project. Indeed, many copies are modified for project-specific enhancements. The practice of clone-and-own reuse therefore requires tool support for identifying the original version of the copy.

In general, there are two approaches to the problem. Forward engineering approaches aim to provide tool support for a more systematic clone-and-own reuse. Jablonski *et al.*, for instance, proposed an extension of the Eclipse IDE that automatically records copy-and-paste activities [4]. Reverse engineering approaches, in contrast, aim to identify instances of reused code in given code bases. Our approach is one of these.

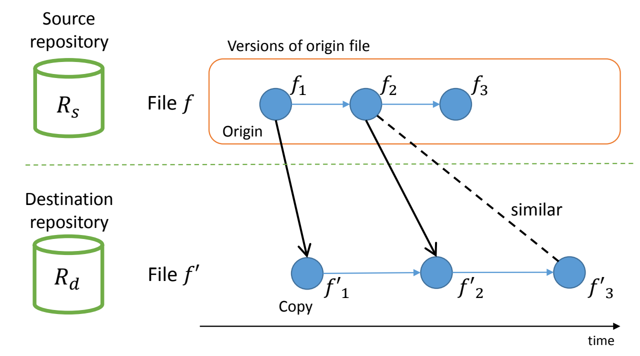

Fig. 1. Separate evolutions of a copy and the origin file.

We propose a file-level analysis technique for detecting copies and the origin files they are copied from across project repositories. For each revision of a source file in a project, we determine the file revision it originated from in another project —while accounting for project-specific changes to the copy.

Figure 1 depicts separate evolutions of a copy and the origin file it is copied from. The file f in the source repository R_s is copied into the destination repository R_d as file f'. A version f_1 of f is updated in the source repository to f_2, which is copied to the destination repository as f'_2. Next, the destination repository adds a new feature to f'_2 resulting in a new revision f'_3. As f'_3 is newer than f_3, it might seem up-to-date upon a casual inspection. In reality, however, it contains outdated source code that still stems from f_2. This example illustrates that commit timestamps do not suffice for determining which revision in a source repository is copied.

Our approach therefore detects instances of clone-and-own reuse based on the similarity between a copy and a candidate origin file revision instead. More concretely, we use Longest Common Subsequence (LCS) [5] as a similarity metric. LCS is common in clone detection [6] and product evolution analyses [7]. We implemented our approach as a tool. Given a source repository and a destination repository, our tool identifies pairs of similar versions of files across the repositories. For each copy in a destination repository, our tool reports project version numbers of the original file revision in a source repository.

To evaluate our approach, we applied our tool to eight open source software projects of which six are known to have reused libpng and two are known to have reused libcurl. The tool reported 1394 file revisions in a destination project and their origin file versions in a source project. We manually verified the reported revisions using the directory structure of the source and destination projects, the commit log of the destination project, and the contents of the copy and the origin file. As a result, 1004 file revisions are correctly recorded in the destination projects. 46 of the reported origin file revisions revealed potential problems that developers recorded different version numbers from actual origin file revisions. Furthermore, 201 of the reported instances are not recorded in the destination projects, even though the instances have the same contents except for white space and code comments. Note that the instances of source code reuse are included in 73 commits in the repositories. Because 23 (31.5%) of them do not record version numbers, our automatic analysis is important to recover source code reuse information across repositories.

The contributions of the paper are summarized as follows.

- An automated analysis method is proposed to detect instances of source code reuse between software repositories.
- The approach is evaluated using the source code reuse information recorded in publicly available software repositories.
- Actual instances of source code reuse in eight projects are analyzed.

Section II shows motivating examples of our approach. The approach itself is detailed in Section III. Section IV presents the evaluation of our approach using a prototype implementation and the aforementioned open source projects. Before concluding in Section VI, we describe related work in Section V.

II. Motivating Examples

When reusing source files from another project, developers may modify these files for their own purpose. An example is found in the V8Monkey project. The project includes a file `pngget.c` which stems from the libpng project. The V8Monkey project modified this file to support PNG animations. When the libpng project released a new version of `pngget.c`, the V8Monkey project merged these changes with theirs. Another example is found in the source code of Wolfenstein: Enemy Territory. The project includes variants of libcurl files that have been modified to satisfy the project's own code formatting conventions.

To maintain all of these copies, knowing the project a file originates from does not suffice. Developers require knowledge about the concrete revision each copy was copied from. Otherwise, it is not clear what revision to update each copy to, since two or more releases of a library may be available at the same time. For example, Cocos2D-iPhone project tried to update their libpng files from 1.2.38 to 1.4.1, but downgraded to 1.2.43 because of source incompatibility. This traceability information is recorded in several manners in practice. Commit messages in the repository of V8Monkey indicate project version numbers of libpng files. For example, the message of commit ID `9def47a86c95fd5f`[1] says "updated libpng to 1.4.3." In the source code of Wolfenstein, on the other hand, the name of directory `curl-7.12.2` hints at the included version of libcurl.

However, traceability information recorded in this manner is not always available. Xia *et al.* [3] reported 18.7% of projects had no version information of the third-party code. Even if traceability information has been recorded when files are copied, the history may not be directly visible to developers. For example, the commit `3a6c8755c4b08c2c` in V8Monkey project records "Move libpng to `media/libpng`." Hence, an older version must be analyzed to identify what version of libpng are used in the project. In addition, incorrect traceability information may be recorded in the repository. For example, the git repository of the Haiku-services-branch project includes a commit `cc57c65424afbcb7` of which the message states "updated libpng to 1.2.31." The commit updated three files. While two of them are exactly the same as files involved in libpng 1.2.31, one file named `png.h` is the same as `png.h` in libpng 1.2.30 except for additional code comments. While this case does not cause a serious problem, a newer version number incorrectly recorded may prevent developers from updating their vulnerable copy. To avoid manual error-prone recording of traceability information, an automated technique for tracking source code reuse between repositories is required.

Two clues for recovering traceability of source code reuse are the time the copy is made and the content of the copy. A simple heuristic is that developers copied the latest version at the time. However, developers sometimes intentionally copy an older version. For example, fs2open project started to use files copied from libpng 1.2.42, although the latest version 1.4.0 has been also available.

Another clue is the content of a copy. Since developers may modify the copy, developers have to compare a copy with candidates of the original version. Identifiers play an important role in this comparison, because most of bug fixes are implemented in a small number of lines of code [8]. For example, commit `f2e2833f28fa11ba` of libpng patched a bug using the following string replacement:

```
- png_ptr->transformations |= PNG_STRIP_ALPHA;
+ png_ptr->flags |= PNG_FLAG_STRIP_ALPHA;
```

Small differences are therefore often vital to identifying which revision a copy stems from. Unfortunately, these are exactly the differences code clone detection tools are intentionally oblivious to [9].

It should be noted that code comment in a copy may indicate a different version from its actual version. For example, the header comment of `png.c` in libpng 1.2.31 says that the file is 1.2.30. The header comment of `pngmem.c` in libpng 1.0.38 also says that the file is 1.2.30. A correct version number of a copy must be verified by the content of the copy.

III. Our Approach

Our approach determines which source code from a *source* repository S (*e.g.,* a library project) is reused into a *destination*

[1]In this paper, a commit ID for a git repository is represented by its first 16 characters that uniquely specify the commit.

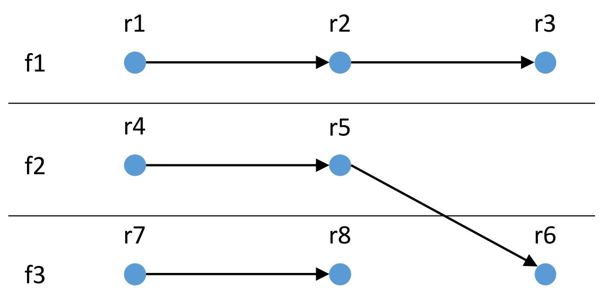

Fig. 2. An example repository including three files f_1, f_2, and f_3.

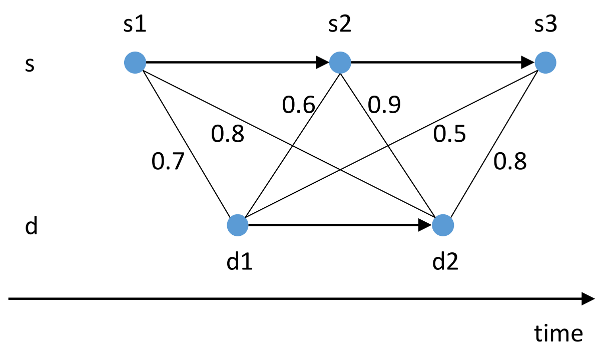

Fig. 3. An example of a file pair (s, d).

repository D (*e.g.*, an application project). An instance of source code reuse is a link between a file revision (a version of a file) in S and a file revision in D. The approach is based on the following assumptions about source code reuse:

- Developers copy a file from a release version of a library.
- Version numbers are available as tags in a source repository.
- Developers do not modify the content of a copy significantly.
- An origin file exists in a source repository before a copy appears in a destination repository.

Based on these assumptions, instances of source code reuse can be detected using the following step-wise approach:

1) Identify file pairs (s, d) such that a revision of s in S is similar to a revision of d in D.
2) Exclude file pairs (s, d) for which d is older than s.
3) Identify the most similar revision i of s for each revision j of d. Output project version numbers corresponding to s_i as the origin of d_j.

The resulting set includes all instances of source code reuse between the repositories. It enables developers of a destination project to know which file revisions have been derived from the source project and should be updated. It also enables developers of a source project to understand how their files are used and extended in the destination project and to improve functionalities in origin files.

We regard a repository as a directed graph of which the vertices represent revisions of files, and of which the edges represent successors of revisions. Figure 2 shows an example of a repository including three files f_1, f_2, and f_3. In the figure, a circle represents a file revision. In the repository, r_1 is the first revision of file f_1. r_2 and r_3 are modified revisions of r_1. The file f_2 is modified once and renamed to f_3. The edge r_4 to r_5 represents the modification, the edge r_5 to r_6 represents the rename, respectively. The lack of an edge from revision r_8 to r_6 indicates that file revision r_8 has been deleted and then replaced by r_6.

$R_k(f)$ refers to all file revisions related to file f in the repository k. To include renamed files in the analysis, $R_k(f)$ includes versions in which the file is named to f and also versions weakly connected to them. For example, in Figure 2, $R_k(f_1) = \{r_1, r_2, r_3\}$, $R_k(f_2) = \{r_4, r_5, r_6\}$, $R_k(f_3) = \{r_4, r_5, r_6, r_7, r_8\}$. Although $R_k(f)$ may include a file revision whose name is not f, we denote the file revisions in $R_k(f)$ as f_i for simplicity.

A. File Pair Extraction

A file d is likely a reuse of a file s if a similarity between a revision of s and a revision of d is above certain predetermined threshold. We extract a set of file pairs C that are likely instances of source code reuse as follows.

$$C = \{(s, d) \mid \exists s_i \in R_S(s)\ \exists d_j \in R_D(d) : sim(s_i, d_j) \geq th\}$$

$sim(s_i, d_j)$ is a similarity metric between revisions. For sim, our technique employs a metric that stems from product evolution analysis [7]. This token-based metric is computed as follows:

$$sim(s_i, d_j) = \frac{|LCS(s_i, d_j)|}{|s_i| + |d_j| - |LCS(s_i, d_j)|}$$

where $|s_i|$ and $|d_j|$ are the numbers of tokens in the file revisions, $|LCS(s_i, d_j)|$ is the length of the LCS of tokens in the file revisions. In this comparison, each file revision is normalized to a sequence of tokens excluding code comments and white space. All other tokens including keywords, macros, and identifiers are kept as is. A threshold th is a configurable parameter of the metric, but we have arbitrarily chosen $th = 0.8$ for our implementation.

This step compares all the file revision pairs between the repositories. To avoid unnecessary computation of LCS for a pair of less similar files, we have employed an optimization. In short, we compare term-frequency vectors of two file revisions to estimate the similarity. If two file revisions have only a small number of common tokens, we do not need to compute the LCS for the revisions, as they cannot have a long common subsequence. By normalizing identifiers in source code, each file is translated into a fixed-length vector representing the frequencies of each lexical element such as keywords and operators in a programming language. In addition to the optimization, we have employed a LCS algorithm suitable for similar strings [10] in order to efficiently compare many similar file revisions.

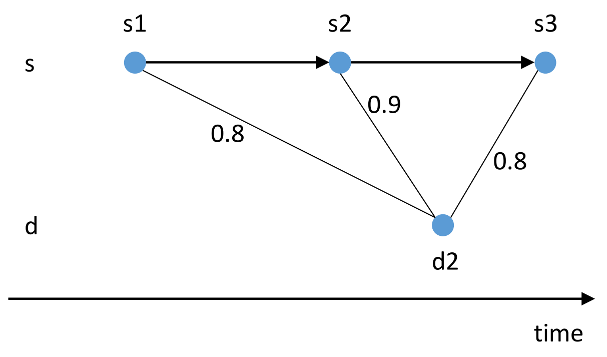

Fig. 4. An example of files that have similar revisions.

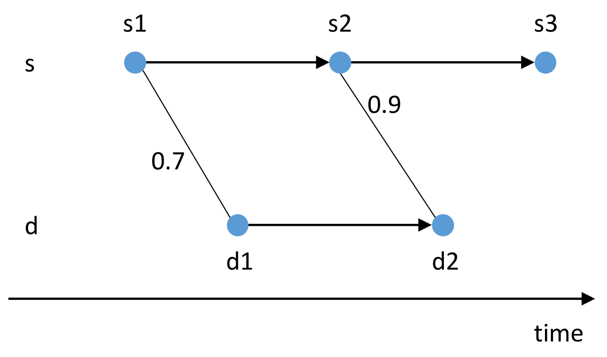

Fig. 5. An example of the result of the final step. Each file revision in the destination repository is linked to its most similar file revision in the source repository. Two revision pairs (s_1, d_1) and (s_2, d_2) are the output of our approach for this file pair.

Figure 3 shows an example of a pair of a file s in a source repository and a file d in a destination repository. In the figure, the file s has three revisions s_1, s_2, and s_3, the file d has two revisions d_1 and d_2. The directed edges between revisions indicate the successor relationship between revisions. The revisions are placed left to right according to their commit time. Cross-repository links between a revision of s and a revision of d carry the similarity between the file revisions. For example, $sim(s_1, d_1) = 0.7$, $sim(s_1, d_2) = 0.8$, and so on. Note that these links are undirected. We regard the file pair (s, d) as a candidate instance of source code reuse, because there are several links whose similarity values are equal to or greater than our 0.8 threshold.

B. Filtering by Commit Time

Having computed the set C of pairs (s, d) of files whose revisions are similar to each other, we filter out file pairs in C that are less likely reuse instances using a heuristic involving commit times. If a file d in D has been created by reusing s, the oldest revision of s that is similar to a revision of d should have been committed earlier than the revision of d.

To compare the commit time, we identify the oldest similar file revisions of s and d. We select file revisions in both repositories that are similar to each other. Formally, we select the following revisions.

$$T_S(s) = \{s_i \in R_S(s) \mid \exists d_j \in R_D(d) : sim(s_i, d_j) \geq th\}$$
$$T_D(d) = \{d_j \in R_D(d) \mid \exists s_i \in R_S(s) : sim(s_i, d_j) \geq th\}$$

We compare the oldest revisions of s and d in the sets. If the oldest revision of d is older than the oldest revision of s, the file pair (s, d) is removed from C. The filtered file pair set $C_{filtered}$ is obtained as follows.

$$C_{filtered} = \{(s, d) \in C \mid$$
$$\exists s_i \in T_S(s)\ \forall d_j \in T_D(d) : \quad t(s_i) \leq t(d_j)\ \}$$

where $t(r)$ is the commit time of a file revision r.

The selected revisions have at least one similar revision in the peer repository. In the case of the pair (s, d) shown in Figure 3, $T_S(s) = \{s_1, s_2, s_3\}$, $T_D(d) = \{d_2\}$. Figure 4 shows the selected revisions. This figure can be obtained by removing links whose similarity is less than the threshold, and removing the revisions which have no links. In the figure, the oldest revisions of file s and d are s_1 and d_2, respectively. According to their commit time, s_1 is older than the oldest revision d_2, *i.e.*, $t(s_1) < t(d_2)$. Hence, the pair (s, d) is kept for the next step.

C. Identify Revision Pairs

We identify similar file revision pairs in the history of file pair (s, d) in $C_{filtered}$ and translate the source file revisions into version numbers of the source project. For each revision $d_j \in R_D(d)$, we identify the most similar revisions s_i in $R_S(s)$ as the original revision corresponding to d_j using the following criteria. While these criteria rely on the same similarity metric as before, they incorporate additional criteria for tie-breaking:

1) Select source revisions that are the most similar to d_j.
2) If two or more source revisions have the same similarity to d_j, select revisions that are most similar to d_j using a text-based similarity metric that is a variant of sim using lines instead of tokens. It takes into account code comments and white space, while it ignores line separators. This rule focuses on file revisions in a source repository whose difference is only code comments, e.g. version numbers in their headers. We still ignore line separators because two projects may use different line separators.
3) If two or more source revisions still have the same similarity to d_j, select the oldest one.

This step generates a set of reuse instances (s_i, d_j) between two repositories. Finally, we translate s_i into version numbers V_i by extracting tags associated with s_i. Since multiple file revisions have the same content in several branches, we list up all file revisions having the same content as s_i and their tags as version numbers corresponding to s_i. We output (V_i, d_j) representing that d_j is likely a copy of the original file revision in versions V_i. If tags are unavailable, we output the commit ID of s_i as an original file revision of d_j. An example is shown in Figure 5, which is computed for the file pair from Figure 3 and Figure 4. Two links $\{(s_1, d_1), (s_2, d_2)\}$ are produced by the step. If version numbers v_1 and v_2 are available for s_1 and s_2, the final output is $\{(\{v_1\}, d_1), (\{v_2\}, d_2)\}$.

D. Implementation

We have implemented our approach as a tool for C programs under version management by Git. The tool takes as input a pair of Git repositories and compares `.c` and `.h` files in the repositories.

The similarity metric is not defined for a file without source code such as an empty file. Hence, the tool excludes such files from analysis.

The output of the tool are pairs of a destination file revision and the source file revision from which it was copied. A file revision is identified by a file path and a commit ID in a repository. The tool also outputs similarity values between revisions to facilitate subsequent analyses. A similarity value less than 1 indicates a destination file revision modified from the original so that developers can compare the contents of the revision pair. Below we list an extract of the results for a source repository libpng and a destination repository fs2open:

Source Path	Tags	Destination Path	Commit	*sim*
`png.c`	v1.2.42	`libpng/png.c`	`101018d`	1
`png.c`	v1.5.2	`libpng/png.c`	`623b6ad`	1
`png.c`	v1.5.7	`libpng/png.c`	`58f9e77`	1
`png.h`	v1.2.42	`libpng/png.h`	`101018d`	1
`png.h`	v1.5.2	`libpng/png.h`	`623b6ad`	1
`png.h`	v1.5.7	`libpng/png.h`	`58f9e77`	1
`pngrio.c`	v1.0.52, v1.2.42	`libpng/ pngrio.c`	`101018d`	1

Due to the limited space, the commit IDs in the list are shortened. The first line indicates that `png.c` tagged as version `1.2.42` in the libpng repository is similar to `libpng/png.c` in the commit `101018d` in the fs2open repository. The full result indicates that developers of the fs2open project copied files in libpng to their repository three times. Because the similarity values are always 1.0, it is likely that the copies remained unmodified afterwards (though code comments could have been modified). In the repository of fs2open, the three commits record the version numbers of libpng in the messages: 1.2.42, 1.5.2, and 1.5.7. We can verify the correctness of the recorded version numbers from the output of the tool. Note that the tool outputs a number of tags for a destination file revision, if the same file content has been included in several releases. For example, the tool reports that `libpng/pngrio.c` in the commit `101018d` is a copy of `pngrio.c` in the versions 1.0.52, 1.2.42, and several other versions (omitted in the above list).

IV. Evaluation

To evaluate the effectiveness of our approach, we have applied our tool to 10 of the projects in Xia's work [11] which identified projects of which files are potentially copied between projects. Two projects libpng and libcurl are selected as source repositories of reused code. Six projects using libpng and two projects using libcurl are selected as destination repositories. Table I shows the project names, URLs, and the summary of the repository attributes. We have assigned IDs for referring to the destination repositories in this section. The columns "Duration," "Latest commit ID," and "#Commits" show the analyzed history of the repositories. The column "LOC" indicates the number of lines of code of `.c` and `.h` in the latest version. Although several destination projects include `.cpp` files, we have analyzed only C files because the source repositories use `.c` files for their implementation.

We have evaluated precision of the output of our tool by comparing with instances of source code reuse that are recorded by developers. As the ground truth, we identify project version numbers of origin files as follows.

- In destination repositories 1-6, we identify a version number from the commit message recorded when the copy has been committed. For example, the version number of a destination file revision committed with a message "Updated to libpng 1.2.31" is 1.2.31.
- In destination repository 7, all files are located in a `curl-7.12.2` directory. Hence, we assume that a destination revision committed in the directory is copied from version 7.12.2.
- In destination repository 8, all files are located in a `curl` directory. We identify a version number in a file named `CHANGES` in the `curl` directory that is committed with other source files at the same time.

Since it is hard to analyze the whole repository, we have analyzed only commits and directories including an instance of source code reuse reported by our tool.

A. Research Method

An instance of source code reuse reported by our tool is correct if the same information is recorded in the destination repository, i.e. the reuse instance is *consistent* with the recorded information. We classified instances of source code reuse reported by our tool into four groups: *consistent*, *inconsistent*, *unrecorded*, and *redundant*.

- A reported instance is *consistent* with the recorded information if the reported destination file revision has been recorded as a copy of version v, and the reported version numbers include the version v.
- A reported instance is *inconsistent* with the recorded information if the reported version numbers have not included the version recorded by developers.
- A reported instance is *unrecorded* if no version number is identified for the reported destination file revision.
- A reported instance is *redundant* if another report links the destination file revision to another source file version that is more appropriate. For example, if two similar files A and B are copied to a destination repository as A' and B', our tool reports four pairs (A, A'), (B, B'), (A, B'), and (B, A'). In this case, (A, B') and (B, A') are regarded as redundant pairs.

The classification process compares the output of the tool with the ground truth. When a destination file revision is similar to revisions of two or more files in a source repository, we manually check source file revisions in the reported versions to identify redundant instances. We select the most similar file revision based on their file paths and a visual inspection as the original revision.

TABLE I. ANALYZED PROJECTS

#ID	Project name	Repository URL	Duration	Latest commit ID	#Commits	LOC
	`libpng`	git://libpng.git.sourceforge.net/gitroot/libpng/libpng	Jul 1995 - Nov 2013	`0e60f06b7c14e698`	3517	76419
	`libcurl`	https://github.com/bagder/curl	Dec 1999 - Oct 2013	`72f850571d24ae48`	16891	152515
1	`cocos2d-iphone`	https://github.com/hansoninteractive/cocos2d-iphone.git	Jun 2008 - Jul 2010	`b47eab8e90f6ba3f`	3754	83111
2	`apitrace`	https://github.com/apitrace/apitrace	Jul 2008 - Oct 2013	`a2ad18752c5692a0`	2694	113472
3	`guliverkli2`	https://github.com/athomasm/guliverkli2.git	Sep 2007 - Feb 2010	`5e6d5d4caa2cf74d`	107	420486
4	`fs2open`	https://github.com/sobczyk/fs2open.git	Jun 2002 - Jul 2013	`156565c8e94b1f58`	8423	197532
5	`v8monkey`	https://github.com/zpao/v8monkey.git	Mar 2007 - Feb 2012	`0280cf71d2c36e0e`	87435	2709919
6	`Haiku-services-branch`	git://github.com/Barrett17/Haiku-services-branch.git	Jul 2002 - Feb 2013	`85e3cb0c85752c45`	44842	5205544
7	`Enemy-Territory`	https://github.com/id-Software/Enemy-Territory.git	Jan 2012	`40342a9e3690cb5b`	1	553309
8	`doom3.gpl`	https://github.com/TTimo/doom3.gpl.git	Nov 2011 - Apr 2012	`8047099afdfc5c97`	39	252618

TABLE II. THE ANALYSIS RESULT

#ID	Source	Destination	Reported	Consistent		Inconsistent		Unrecorded		Redundant	
1	`libpng`	`cocos2d-iphone`	147	127	(86.4%)	0	(0.0%)	20	(13.6%)	0	(0.0%)
2	`libpng`	`apitrace`	78	39	(50.0%)	0	(0.0%)	39	(50.0%)	0	(0.0%)
3	`libpng`	`guliverkli2`	131	71	(54.2%)	0	(0.0%)	60	(45.8%)	0	(0.0%)
4	`libpng`	`fs2open`	57	57	(100.0%)	0	(0.0%)	0	(0.0%)	0	(0.0%)
5	`libpng`	`v8monkey`	275	183	(66.5%)	18	(6.5%)	74	(26.9%)	0	(0.0%)
6	`libpng`	`Haiku-services-branch`	306	218	(71.2%)	1	(0.3%)	87	(28.4%)	0	(0.0%)
7	`libcurl`	`Enemy-Territory`	210	150	(71.4%)	26	(12.4%)	0	(0.0%)	34	(16.2%)
8	`libcurl`	`doom3.gpl`	190	159	(83.7%)	1	(0.5%)	0	(0.0%)	30	(15.8%)
		Total	1394	1004	(72.0%)	46	(3.3%)	280	(20.1%)	64	(4.6%)

TABLE III. THE RESULT OF CONTENT COMPARISON OF CONSISTENT REUSE INSTANCES.

#ID	Destination	Not Modified	Modified
1	`cocos2d-iphone`	127	0
2	`apitrace`	39	0
3	`guliverkli2`	68	3
4	`fs2open`	57	0
5	`v8monkey`	63	120
6	`Haiku-services-branch`	218	0
7	`Enemy-Territory`	54	96
8	`doom3.gpl`	156	3
Total		782	222

TABLE IV. THE RESULT OF CONTENT COMPARISON OF UNRECORDED REUSE INSTANCES.

	Reuse w/o Version Number		Others	
#ID	Not Modified	Modified	Not Modified	Modified
1	0	0	20	0
2	39	0	0	0
3	0	0	56	4
4	0	0	0	0
5	14	38	9	13
6	21	0	42	24
7	0	0	0	0
8	0	0	0	0
Total	74	38	127	41

B. Results

We have applied our tool to eight project pairs. Table II shows the result of the classification. The column "Reported" shows the number of reuse instances extracted by our approach. The columns "Consistent," "Inconsistent," "Unrecorded," and "Redundant," show the classified result. Our tool reported 1394 pairs as instances of source code reuse.

1) Consistent Instances: 1004 instances (72.0%) of the reported 1394 instances are consistent with the recorded information. To analyze how often developers modified file revisions after copying, we have compared the contents of source file revisions and destination revisions. In Table III, the consistent instances are classified into two groups: *Not Modified* and *Modified*. A reuse instance is classified as *Not Modified* if the similarity metric value of the pair is 1.0. In other words, the differences are limited to code comments and white space. Otherwise, a reuse instance is classified as *Modified*. Our results indicate that projects #5 (V8Monkey) and #7 (Wolfenstein: Enemy Territory) often modify the source code after copying from the source repositories. As described in Section 2, V8Monkey modified copied revisions to handle Animation PNG. Wolfenstein also modified various functions in the files. Except for these projects, developers tend to reuse source code as is.

2) Inconsistent Instances: 46 instances (3.3%) are inconsistent with the recorded information. While this group may include reuse instances incorrectly reported by the tool, it also includes reuse instances incorrectly recorded by developers. By analyzing the instances, we have identified potential problems in the destination repositories. The first example is found in project #5. A commit message says that files are updated to libpng 1.2.31, whereas some of the committed file revisions are included in either 1.0.38 or 1.2.30. Those files are regarded as 1.2.31 partly because the files in the versions have the same contents except for code comments. The second example is `png.h` in the project #6 that is described in Section 2. The content of the file is the same as 1.2.30 but recorded as 1.2.31.

3) Unrecorded Instances: 280 instances (20.1%) are unrecorded because of the lack of version numbers in commit messages. 112 of them are recorded as source code reuse in the commit message, although there are no version numbers. The remaining 168 instances have no explicit information about source code reuse. We have compared the contents of source revisions and destination revisions and classified the instances to two categories: *Not Modified* and *Modified*. A reported reuse instance is *Not Modified* if the similarity of the source revision and the destination revision is 1.0. Otherwise, a reuse instance is *Modified*. The result is shown in Table IV. The columns "Reuse w/o Version Number" show the number of reuse instances whose commit message mentioned source code reuse. The columns "Others" show the number of other reuse instances. 201 "Not Modified" instances are likely represent instances of source code reuse, since the file revisions have the same contents between repositories. Although some instances may point to different source versions from actual versions, we expect that most of them are likely correct as the number of consistent instances are 22 times greater than the number of inconsistent instances.

Developers did not record version numbers for these instances partly because the developers focused on source code management issues rather than source code reuse. For example, project #1 updated libpng copies with a message "v0.99.1 release tag." The project #3 has a similar commit updating libpng files with a message "Guliverkli revision 611." In addition, developers did not record version numbers when they did not change the contents of revisions. For example, project #2 and #5 moved libpng copies in their repositories. They recorded only what they did, *e.g.* "Move libpng to `media/libpng`," without version numbers. The version numbers of those files recovered by our tool would help developers to understand what files are included in the repositories.

4) Redundant Instances: We have identified 64 redundant reuse instances in projects #7 and #8. One cause is the lack of history of file moving and renaming in the libcurl repository. For example, a file `getpass.c` is located in `lib` directory in libcurl 7.10.6 and located in `src` directory in libcurl 7.11.2. Our tool could not identify the revision in `src` as a successor version of the file in the `lib` directory. Hence, our tool reported two instances of source code reuse for a single version of the file `getpass.c` in the project #7. In the analysis, we manually identified the correct version, because the project #7 stores the file revision in `libcurl-7.12.2/src` directory. In addition, the file content is the same as `src/getpass.c` in 7.11.2.

Redundant instances are also reported when two or more similar files are included in a source repository. For example, `lib500.c` and `lib501.c` in libcurl 7.12.2 defined test functions whose contents are almost the same except for two lines. Since the files are not linked in a repository, our tool reported two redundant reuse instances: `lib500.c` is reused as `lib501.c`, and vice versa. As similar to the first case, we have manually identified correct reuse instances by file paths.

C. Precision

We calculate precision of our tool by comparing the reported instances with the recorded information as follows.

$$P = \frac{c}{c+i+r}$$

where c is the number of consistent reuse instances (1004), i is the number of inconsistent reuse instances (46), and r is the number of redundant reuse instances (64). The resulting value is $P = 0.901$. The number of unrecorded reuse instances is excluded from this calculation because we cannot verify whether they are actual source code reuse or not.

We have assumed that similar source files are source code reuse, but the assumption is not always true. One example is `md4.c` in the project #7. While all the analyzed projects put their library copies in directories having special names, *e.g.* `external/libpng/` and `thirdparty/libpng`, it is the only one case reported for a file revision outside of such a directory. Hence, the file is likely a third party code used by the project #7 and libcurl. Another case is a file named `lib/getdate.c` in the project #8. The file is generated by a code generator during the build process of libcurl. Since the file is involved in both source and destination repositories, our tool reported the file as an instance of reuse even if developers did not directly copy the file. We think these are false positives of our approach. However, we could not verify the latter case in the experiment. In the case, two file revisions are involved in both repositories and marked as reuse by developers. There are no differences from other consistent instances. We accidentally identified the case simply because the source repository included a commit message telling the file is generated one and removed in the future release. We could not count the number of this kind of false positives because of our limited effort.

TABLE V. THE NUMBER OF UNREPORTED FILE REVISIONS INCLUDED IN DIRECTORIES THAT CONTAIN COPIES FROM LIBRARIES

#ID	#Revisions	Source	No Source
1	5	5	0
2	3	0	3
3	0	0	0
4	2	0	2
5	19	1	18
6	9	9	0
7	6	6	0
8	3	3	0
Total	47	24	23

D. Recall

Although we cannot know the number of files actually reused, developers tend to put copies of a library in the same directory. Indeed, only 12 directories and their subdirectories are used for the 8 projects. We have analyzed all the file revisions in the directories, because they are also likely copies from the library. Table V shows the result. The column "#Revisions" shows the number of file revisions that are not reported by our tool. The file revisions are classified into "Source" and "No Source" according to whether the same file name is found in a source repository or not.

The file names of 24 file revisions are found in source repositories. 12 of 24 revisions include only comments instead of source code. Our tool simply removed such files from the analysis. One example is `pnggvrd.c` found in the project #1. The file includes only a single comment as follows.

```
/* pnggvrd.c was removed
   from libpng-1.2.20. */
```

Other file revisions are modified from the original version and similarity metric values for them are less than the threshold. For example, the similarity between a copy of `pngconf.h` in the project #6 and its most similar revision in libpng is 0.77. 14 of those 24 revisions are recorded as reuse in their commits. Hence, they are likely false negatives accidentally missed by our tool.

23 file revisions are not found in source repositories. The file revisions are associated with two files: `pnglibconf.h` and `mozpngconf.h`. The former file is created by a script in libpng project, although the project #4 recorded the revisions as source code reuse from libpng. The latter file is created by Mozilla project to replace function names for reuse

TABLE VI. THE NUMBER OF COMMITS INCLUDING INSTANCES OF SOURCE CODE REUSE

#ID	#Total	#Consistent	#Inconsistent	#Unrecorded
1	8	7	0	1
2	4	2	0	2
3	8	4	0	4
4	3	3	0	0
5	25	10	4	11
6	23	17	1	5
7	1	0	1	0
8	1	0	1	0
Total	73	43	7	23

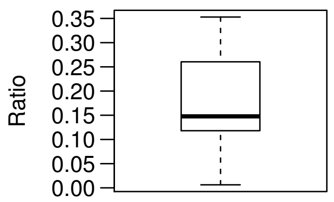

Fig. 6. The ratio of inconsistent instances against the number of reuse instances in a commit

in the project. For example, a macro replaces a function `png_read_data` in libpng with `MOZ_PNG_read_data`. We do not regard these file revisions as false negatives because they are not copied from source repositories.

By assuming there are no potential copies in other directories, recall of our tool can be estimated as follows:

$$R_{estimated} = \frac{c}{c+i+n}$$

where c is the number of consistent pairs (1004), i is the number of inconsistent pairs (46), and n is the number of false negatives we identified (14). The resultant value is $R_{estimated} = 0.943$.

E. Commit-level Analysis

Since developers may copy a number of files at once from a source repository, we have analyzed the number of commits including reuse instances for each project. Table VI shows the result. The column "#Total" indicates the number of commits that include at least one reuse instance. The column "#Consistent" indicates the number of commits with correct version numbers. The column "#Inconsistent" indicates the number of commits including at least one inconsistent reuse instance. The column "#Unrecorded" indicates the number of commits without version numbers. The result shows that an automatic analysis is important for developers, since 23 (31.5%) of 73 commits do not include version numbers.

Inconsistent reuse instances are always committed with several consistent reuse instances. Figure 6 shows the ratio of inconsistent instances against the number of reuse instances in a commit. For example, if a commit updated three files obtained from a library but one of them was copied from another version of the library, the ratio is 0.33. The result shows that 7 (9.6%) of 73 commits include 18.1% of files from different versions of a library accidentally. In such a case, our tool reports different version numbers for each file in a commit. Consequently, developers can recognize such potential problems using our tool.

TABLE VII. THE EXECUTION TIME

#ID	Execution Time	#LCS Computed
1	40min51sec	87,280
2	55min6sec	76,066
3	38min13sec	77,525
4	23min43sec	28,378
5	225min33sec	307,910
6	139min45sec	175,588
7	5min26sec	10,691
8	4min35sec	10,162

F. Performance

Our tool compares all file revision pairs between repositories in the worst case, while we have employed an optimization. Table VII shows the time taken to execute the analysis using a single thread on Intel(R) Xeon(R) CPU E5-1603 2.80GHz. The time does not include the time to copy Git repositories to a local storage. We have two large repositories in the analysis. Repository #5 involves 87,435 commits. The total amount of code we have analyzed is 62.9 MLOC. Repository #6 involves 44,842 commits. The total amount of code we have analyzed is 37.8 MLOC. Our tool finished analysis for those large products in several hours. The time depends on the number of similar file pairs between repositories that require computing the longest common subsequence between file revisions. Table VII indicates the number of computed longest common subsequences. The numbers are much smaller than the number of possible revision pairs between repositories because of our optimization. In addition, the computed results can be reused for an incremental analysis, since similarity values between existing file revisions do not change in the future.

G. Threats to Validity

The precision of our approach is computed against the information recorded in the repositories. If developers of the analyzed projects often copied file revisions without recording this and subsequently modified most copies significantly, such modified files are not taken into account. As described in Section IV-E, several false positives may be regarded as true positives in our analysis. On the other hand, inconsistent reuse instances are regarded as false positives. Some of them could not be identified because of incorrectly recorded version numbers.

We have used a single threshold value $th = 0.8$ in our implementation. While another threshold value may change the detection result, we believe that the value is not so significant because most of the consistent reuse instances are file copies without modification. Indeed, our manual analysis of directories identified only 14 false negatives that are accidentally filtered out.

We have selected two library projects for source repositories: libpng for manipulating image files and libcurl for transferring data using various protocols. Both of them are utilities to implement features of applications. The source code reuse activities of developers in eight projects may be limited to a particular style for using such a utility library.

V. Related Work

A. *Software Product Lines*

Software Product Line Engineering involves much forking and copying of files across variants of a software product. Tracing similarities between variants has its benefits for selection of the most appropriate variant. Duszynski [12] proposed Variant Analysis to compare source code of product variants to understand product-specific features and common (reused) features.

Hemel [13] showed that the tracing of the evolution also has reverse and forward engineering benefits. Nonaka [14] visualized the relationships of variants and analyzed corrective maintenance data for a particular product family. Yamamoto [15] defined similarity between source code of software products. Kanda [7] defined similarity between software products using a file-level similarity metric to identify the origin of the variants with their evolution. We have employed the file similarity metric to compare file revisions. Kanda [7] also introduced an optimization using term frequency vectors to avoid unnecessary comparison of files. We normalize identifiers to compute fixed-length term frequency vector for source code.

Software Product Line targets reuse between variants of the same products or from projects originating from the same family. Ray [16] analyzed how forked projects import source code changes from other projects. While the analysis compare patches between projects, our analysis compares file revisions between projects. One reason is that files copied from a library are updated less frequently.

B. *Code Clones and Origin Analysis*

Similar fragments of code are considered to be code clones. The longest common subsequence has also been implemented for 'suffix' based clone detection tools. Recently, clone detection work efforts have concentrated around the improvement of the state of the art through scalability, speed and precision [17], [18].

Other studies have done cross-project clone detection, however with different intentions. Bauer [19] investigated the extraction of similar pairs of files between projects to create a library. Our approach extracts source files copied from a library. Kim [20] coined clone genealogy to help understand clone management and refactoring. Our analysis might enable a genealogy analysis between projects, since the output of our tool links the history of files in a library and their copies in an application. Krinke [21] proposed to use version information in repositories to distinguish original code from its clones across projects. We employs source code similarity without normalizing identifiers differently from code clone detection tools as described in Section II. Al-Ekram [22] and Kawaguchi [23] all studied cloning for patterns. Ichi-tracker [24] is an example of the many code search research that search for clones across various repositories on the Internet. While Ichi-tracker extracts similar file revisions in repositories, it does not tell which one is likely the original revision. Our analysis employs heuristics for identifying an original file revision.

According to Godfrey [25], the merge and splitting of source code entities is a common activity during the lifespan of a software system. During this process, the original context may be lost over time. They then show how tracking origins is beneficial for ownership, code comprehension, refactoring and software evolution research. Hashimoto [26] proposed an AST-based comparison of file revisions to track co-evolution of two branches in a project. Our analysis enables to track the history of source code reuse between two projects. German and colleagues [27] proposed Software Bertillonage tracing licensing implications of copied code fragments.

C. *Software Reuse*

Software reuse has been becoming standard practice in software engineering. Most research has focused on social, extent and nature of software reuse. Most studies borrow code clone detection tools for identifying actual source code reuse. Due to the complex nature of white-box reuse, a manual verification aspect is usually required. For example, Heinemann [28] analyzed white-box source code reuse among Java projects by manually inspecting source code with the result of a code clone detection tool. Xia [3], [11] also manually analyzed how source code is reused, while the source code was obtained by a clone detection technique. In this study, we present and evaluate an automatic detection technique for white-box reuse. The manual analyses conducted in [3], [11] can be automated by our tool. In those work, an original file revision was manually identified in files reported by a clone detection tool. Our tool can automatically identify the most similar file revision as the origin of the reused file.

German *et al.* [29] analyzed code siblings copied across open source software projects. They detected code clones for certain releases and then investigated their history when the code clones are introduced. Our tool analyzes the entire repository of projects so that developers can analyze file revisions that are modified after copy.

VI. Conclusion

Developers often reuse source code developed by another project. Using a source code similarity metric, we have automatically extracted revision pairs that are likely source code reuse. In the experiment, we have extracted 1394 revision pairs from eight project pairs. The estimated precision and recall of the tool are 0.901 and 0.943. 1004 (72.0%) of the pairs are consistent with the information recorded in the repositories. We have identified several inconsistent pairs that are caused by incorrect information in the repositories. 201 unrecorded reuse instances pointed to file revisions in source repositories whose contents are the same as revisions in destination repositories. 31.5% of commits for source code reuse have no version numbers in the commit messages.

Developers may use the tool as a static checker before a release to avoid security vulnerability and known issues of a library in their software. Even if developers did not record a version number of a library, our tool reports version numbers of the library using the source code in their repository. The version numbers help developers to decide whether files reused from the library should be updated or not. The tool also reports a similarity between a reused file and its original file so that developers can integrate their project-specific enhancement into the latest version of the library.

In the future work, we are planning to improve the execution time of the tool. We also would like to automatically identifying how a file revision is modified in a destination repository. Since developers may import small changes such as security fixes without changing the other source code, automatically generating an explanation how a copy has been modified from the most similar revision is valuable for developers to analyze the current status of source code of their project. Finally, we are also interested in cross-project analysis including more than two projects. Because our current implementation reports the same library file used in two projects as source code reuse, we would like to include multiple projects in analysis so that our tool can report more accurate and useful results.

Acknowledgment

We would like to thank Dr. Daniel M. German for his valuable comments on this research.

The work was supported by JSPS KAKENHI No.25220003 and Osaka University Program for Promoting International Joint Research.

References

[1] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik, "Managing forked product variants," in *Proceedings of the 16th International Software Product Line Conference*, 2012, pp. 156–160.

[2] T. Mende, R. Koschke, and F. Beckwermert, "An evaluation of code similarity identification for the grow-and-prune model," *Journal of Software Maintenance and Evolution*, vol. 21, no. 2, pp. 143–169, 2009.

[3] P. Xia, M. Matsushita, N. Yoshida, and K. Inoue, "Studying reuse of out-dated third-party code in open source projects," *JSSST Computer Software*, vol. 30, no. 4, pp. 98–104, 2013.

[4] P. Jablonski and D. Hou, "Aiding software maintenance with copy-and-paste clone-awareness," in *Proceedings of the 18th International Conference on Program Comprehension*, 2010, pp. 170–179.

[5] D. Gusfield, *Algorithms on Strings, Trees, and Sequences*. Cambridge University Press, 1997.

[6] C. K. Roy and J. R. Cordy, "NiCad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proceedings of the 16th International Conference on Program Comprehension*, 2008, pp. 172–181.

[7] T. Kanda, T. Ishio, and K. Inoue, "Extraction of product evolution tree from source code of product variants," in *Proceedings of the 17th International Software Product Line Conference*, 2013, pp. 141–150.

[8] Lucia, F. Thung, D. Lo, and L. Jiang, "Are faults localizable?" in *Proceedings of the 9th Working Conference on Mining Software Repositories*, 2012, pp. 74–77.

[9] C. K. Roy and J. R. Cordy, "Scenario-based comparison of clone detection techniques," in *Proceedings of the 16th International Conference on Program Comprehension*, 2008, pp. 153–162.

[10] S. Wu, U. Mamber, and G. Myers, "An o(np) sequence comparison algorithm," *Information Processing Letters*, vol. 35, no. 6, pp. 317–323, 1990.

[11] P. Xia, "An empirical study of out-dated third-party code in open source software," Master's thesis, Osaka University, 2013.

[12] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential," in *Proceedings of the 18th Working Conference on Reverse Engineering*, 2011, pp. 303–307.

[13] A. Hemel and R. Koschke, "Reverse engineering variability in source code using clone detection: A case study for Linux variants of consumer electronic devices," in *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 357–366.

[14] M. Nonaka, K. Sakuraba, and K. Funakoshi, "A preliminary analysis on corrective maintenance for an embedded software product family," *IPSJ SIG Technical Report*, vol. 2009-SE-166, no. 13, pp. 1–8, 2009.

[15] T. Yamamoto, M. Matsushita, T. Kamiya, and K. Inoue, "Measuring similarity of large software systems based on source code correspondence," in *Proceedings of the 6th International Conference on Product Focused Software Process Improvement*, 2005, pp. 530–544.

[16] B. Ray and M. Kim, "A case study of cross-system porting in forked projects," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[17] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[18] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 1289–1292.

[19] V. Bauer and B. Hauptmann, "Assessing cross-project clones for reuse optimization," in *Proceedings of the 8th International Worskhop on Software Clones*, 2013, pp. 60–61.

[20] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering*, 2005, pp. 187–196.

[21] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between GNOME projects," in *Proceedings of the 7th Working Conference on Mining Software Repositories*, May 2010, pp. 98–101.

[22] R. Al-Ekram, C. Kapser, R. C. Holt, and M. W. Godfrey, "Cloning by accident: an empirical study of source code cloning across software systems," in *Proceedings of the 4th International Symposium on Empirical Software Engineering*, 2005, pp. 376–385.

[23] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: an automatic categorization system for open source repositories," *Journal of Systems and Software*, vol. 79, no. 7, pp. 939–953, 2006.

[24] K. Inoue, Y. Sasaki, P. Xia, and Y. Manabe, "Where does this code come from and where does it go? – integrated code history tracker for open source systems –," in *Proceedings of the 34th International Conference on Software Engineering*, 2012, pp. 331–341.

[25] M. Godfrey and L. Zou, "Using origin analysis to detect merging and splitting of source code entities," *IEEE Transactions on Software Engineering*, vol. 31, no. 2, pp. 166–181, 2005.

[26] M. Hashimoto and A. Mori, "A method for analyzing code homology in genealogy of evolving software," in *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering*, 2010, pp. 91–106.

[27] J. Davies, D. M. German, M. W. Godfrey, and A. Hindle, "Software bertillonage: Finding the provenance of an entity," in *Proceedings of the 8th Working Conference on Mining Software Repositories*, 2011, pp. 183–192.

[28] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," in *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, 2011, pp. 207–222.

[29] D. M. German, M. D. Penta, Y.-G. Gueheneuc, and G. Antoniol, "Code siblings: Technical and legal implications of copying code between applications," in *Proceedings of the 6th Working Conference on Mining Software Repositories*, 2009, pp. 81–90.