Where Does This Code Come from and Where Does It Go? - Integrated Code History Tracker for Open Source Systems -

Katsuro Inoue, Yusuke Sasaki, Pei Xia, and Yuki Manabe
Osaka University
Osaka, Japan
{*inoue, peixia, y-manabe*}*@ist.osaka-u.ac.jp*

***Abstract*—When we reuse a code fragment in an open source system, it is very important to know the history of the code, such as the code origin and evolution. In this paper, we propose an integrated approach to code history tracking for open source repositories. This approach takes a query code fragment as its input, and returns the code fragments containing the code clones with the query code. It utilizes publicly available code search engines as external resources. Based on this model, we have designed and implemented a prototype system named *Ichi Tracker*. Using Ichi Tracker, we have conducted three case studies. These case studies show the ancestors and descendents of the code, and we can recognize their evolution history.**

***Keywords*-Code Search; Software Evolution; Open Source System**

I. Introduction

Open source systems are extremely useful resources for the construction of current software systems. Even software systems in the industry increasingly use open source systems due to their reliability and cost benefits [25].

One of usages of the open source systems is to reuse the source code of the open source systems for other projects. We can easily get the source code files of various projects from the repositories on the Internet, such as SourceForge [29] and Maven Central [26]. Those source code files are copied and modified if necessary, and they are built into a new system. Repeating such inter-project copies of files and code fragments makes a lot of code clones among the open source projects, and the interdependency of those projects is becoming very complex [22].

Consider a case that there is an open source code file used by a system, but we do not know much about the original project. We are wondering if we could safely and effectively reuse that source code file for a new project.

In such a situation, it is very important to identify the origin of the source code. By identifying the original project, we would understand various characteristics of the project, such as developers, copyrights, licenses, created dates, and so on. Also, we would like to know the evolution of the source code, since the reuse and maintenance information of the source code by many other projects is a very important clue for the software developers to make a decision to reuse the source code.

Current software engineering tools do not provide sufficient support to explore code history. To know the code origin, we have to specify project names and/or URLs. Also, to know the code evolution, we have to understand the interrelations of open source projects.

Code search engines such as Google Code Search [10] and Koders [3] are very useful tools to explore open source repositories for the origin and evolution of code. However, current code search engines only allows to get keywords and/or code attributes as their inputs, and they return source code files which contains those keywords and attributes. Selecting appropriate inputs for those search engines is not easy task for general users.

In this paper, we will propose an integrated approach to code history tracking for open source repositories. Also, we will present its prototype system named *Ichi Tracker* (Integrated Code History Tracker). Ichi Tracker takes a code fragment as its query input, and returns a set of cloned code fragments which can be found by popular source code search engines such as SPARS/R [30], Google Code Search [10], and Koders [3]. Ichi Tracker helps us to understand the backward and forward history of the query code fragment.

Using Ichi Tracker, we have performed various case studies. In this paper, we will show three examples of tracking code, `texture.java`, `kern_malloc.c`, and `SSHTools`.

Contributions of this paper are as follows.

- We have proposed and implemented an integrated code history tracking model to find similar code fragments using code search engines. This model is very effective to identify the evolution of code. Also, it would be useful to find plagiarism or illegal reuse of code.
- An analysis method of code history using a similarity metric (called cover ratio) and the last modified time has been proposed and used for the case studies.
- Using these techniques, the code histories of three case studies have been presented.

In this paper, we will first describe the tracking model in Section II. In Section III, detailed processes of Ichi Tracker will be explained. Section IV will show our case studies. Section V will discuss our approach and Section VI will

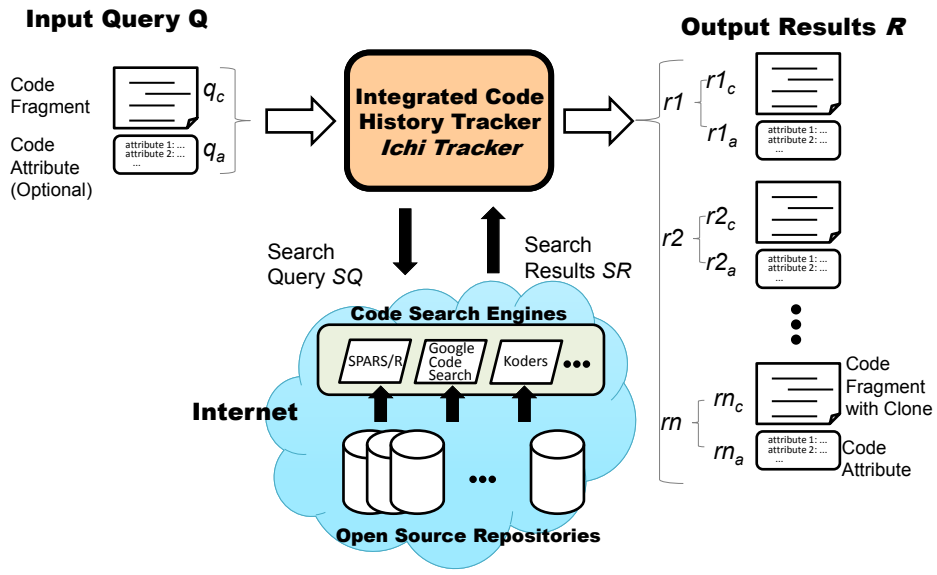

Figure 1. Integrated Model for Code History Tracking with Ichi Tracker

show the related works. In Section VII, we will conclude our discussions with some future works.

II. Integrated Code History Tracking Model

A. Overview

Fig. 1 shows our model for the integrated code history tracking. The core of this model is *Ichi Tracker*, which takes an input query Q and replies an output result set R. The details will be described in Section III.

Input query Q is composed of *code fragment* q_c and *code attribute* q_a. q_c may be a complete source code file or a part of a source code file, which is in question. q_a is a set of associated information characterizing q_c, such as the file name, project name, URL of repository, created time, last modified time, and so on. q_a is optional and could be added to improve the quality of the output results.

Output result R is composed of results $r1, r2, ..., rn$, and each result ri is composed of a code fragment ri_c and its code attribute ri_a. Both q_c and ri_c contain at least one pair of Type 2 code clones[1]. This means that each ri_c contains a clone of the whole or a part of q_c.

Attribute q_a of query Q is optional in the sense that the file name and project name might help to improve the search quality. Attribute ri_a of result ri contains valuable information to know the characteristics of the cloned code fragment ri_c. The last-modified time could be used to identify the ancestor or descendant relations. The project name and URL would be indicators of prevalence and popularity of the query code among the open source projects. Sometimes those attributes might not be obtained easily. We will discuss this issue in Section V-B-3).

In order to track a code history of open source systems, we would need to have a huge repository containing various open source systems and their historical versions. In addition to using our code search engine SPARS/R [30], we use external search engines Google Code Search [10][2], and Koders [3][3], assuming that those engines collect and contain sufficiently enough open source systems.

Google Code Search and Koders are very popular search engines, since they provide a lot of useful information for open source systems. Google Code Search provides search features with keywords associated with optional attributes such as package names, languages, and licenses. Koders provides the keyword search feature with language names and license types. These engines contain huge source code repositories behind them, and those repositories are kept updated by their crawling activities and also the user's con-

[1]Type 2 clones are syntactically identical fragments except for variations in identifiers, literals, types, whitespace, layout and comments [27]. In this paper, we assume that clones are Type 2 unless explicitly stated otherwise.

[2]Google has terminated its service for Google Code Search on Jan. 15, 2012. All the discussion and data here are based on the service when it was available.

[3]Since Koders does not allow to send automated queries, we have manually sent it the keywords obtained by our system.

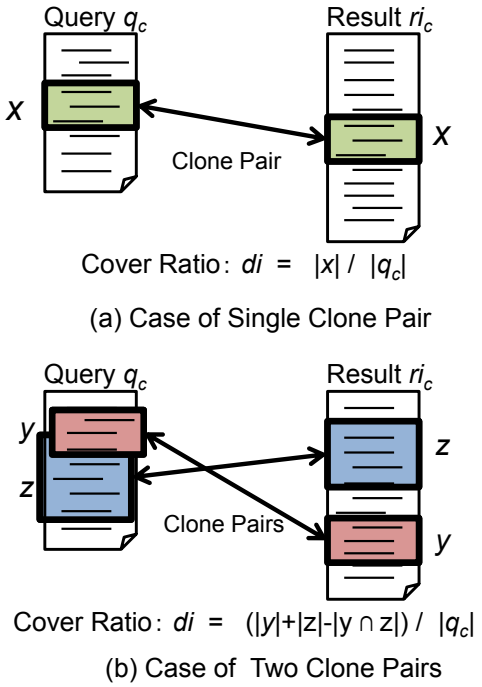

Figure 2. Definitions of Cover Ratios

tributions. SPARS/R is our Java component search engine with the keyword input and component rank mechanism [12]. The Java class repository is kept updated by our research group.

Ichi Tracker gives the search query SQ to those code search engines, and gets the search result SR from those engines.

B. Cover Ratio

Assume that the query code fragment q_c and a result code fragment ri_c share code clone x as shown in Fig. 2 (a). The *cover ratio* di of ri_c for q_c is defined as follows.

$$di = \quad |x| \quad / \quad |q_c|$$

Here, $|x|$ means the size (token length) of x. If there are multiple clone pairs between q_c and ri_c, then we add those clone sizes excluding their overlapping area and divide it by the query code size $|q_c|$, as shown in the case of two clone pairs of Fig. 2 (b).

If the cover ratio is 1.0, then the result fragment contains the overall of the query code q_c, and if it is 0.0 then the result does not contain any part of q_c. Note that each result $r1_c$, $r2_c$ ... may have different cover ratio $d1$, $d2$, ... , since they may share different clones with q_c. The cover ratio can be an important attribute of each result, which indicates how close the result code is to the query code.

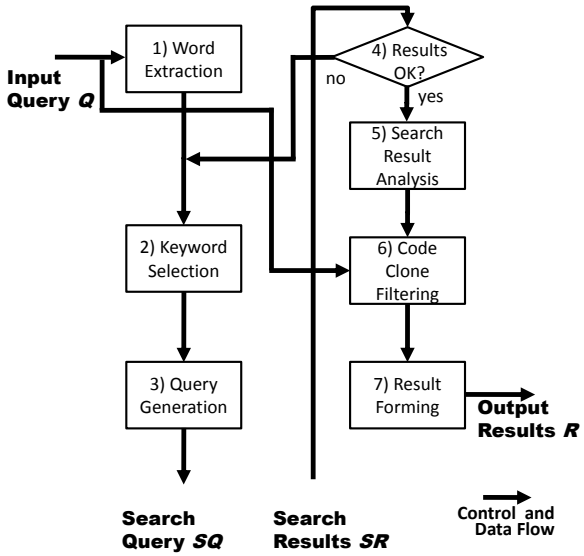

Figure 3. Processes of Ichi Tracker

III. Processes of Ichi Tracker

Fig. 3 shows an overview of the processes of Ichi Tracker. For the simplicity, here we present only one strategy we have taken to choose keywords for the code search engines, but we could consider many variations of different strategies, algorithms, and parameter settings, some of which will be discussed in Section V-B.

1) Word Extraction: At the beginning, code fragment q_c in input query Q is tokenized, and the words from source code part only are extracted. A single word of Camel Case or Snake Case is not decomposed into multiple words. If code attribute q_a is additionally given, those are also extracted.
2) Keyword Selection: Next, the keywords used for the following query generation are selected from the extracted words. Initially, the reserved words of the source code language are removed. Also, short-length words less than 5 characters are deleted. Then we apply a simple word-selection strategy that n most frequently-used words are chosen from the source code part. Finally, we obtain n popular words which are not language's keywords and whose length are 5 or more.
3) Query Generation: Using the selected keywords, a search query SQ for the code search engines is created. As the search engines, we use SPARS/R, Google Code Search, and Koders, because of their availability and flexibility. All of these search engines accept a keyword sequence as their query input, so we use the sequence of n most frequently used words as SQ. If the additional input attribute q_a is accepted by the

search engines, it is also given to the search engines.

4) Result OK?: The header lists of the results of each search engine are received. If the length m of each list is greater than the predetermined maximum limit m_{max}, then the system retries from Step 2) with $n+1$ most frequently used words, assuming that the search query SQ was too broad. This means that we add one word to the previous keyword list, and we expect narrower results than the previous try. As the default setting, we start with $n = 1$ and m_{max} is 50.
5) Search Result Analysis: Using the header lists, each source code sr_1, sr_2, ..., sr_i, ... is downloaded from each search engine. The downloaded files might be complex html forms consisting of not only source code but other information such as frame window, line number, highlighted keyword, file name, project name, language, license, and so on. From such results, the pure code parts and their associated comments are extracted as the resulting code. Also, useful information for the code attributes such as file name, project name, license, URL, and others are extracted if those are available.
6) Code Clone Filtering: The code clones between the input query code fragment q_c and each source code sr_i obtained in Step 5) are computed. If sr_i does not share a clone with q_c, then we delete sr_i from the result list. We have used a code clone detection tool CCFinder [15], with its parameter setting for the minimum token length 10.
   This step works as a filtering out process of SR. In general, code search engines reply many false positive results, and we have to elaborate to choose appropriate query keywords or to structure queries to get better precision [11]. On the other hand, by using this code clone filter, we can easily remove unrelated code, and so we can simply use code search engines without such elaboration of the input query.
7) Result Forming: All the remaining code in Step 6) and their code attributes are combined and packed as the output result R of this system.

In our current implementation, all result code fragments $r1_c$, $r2_c$, ... in R are not a part of files, but they are complete files returned from each search engine.

IV. Experiments

We have conducted several case studies to explore the applicability of Ichi Tracker. All these experiments have been performed under PC Workstation with dual Xeon X5550 2.66GHz processors and 24GB memory between Feb. 2011 and May 2011.

A. Case Study 1: `texture.java`

`texture.java` is a 1,600 LOC Java file to define a graphic texture object in game programs. It was developed by a game engine project *jMonkeyEngine* [14]. This file is popularly used by many 3D games.

We have given Ichi Tracker the overall source code of this file as the input query code fragment q_c. As an optional attribute, the file name `texture.java` is associated when it is needed.

Table I
Number of Output Results $|R|$ and Search Results $|SR|$ for `texture.java` as Query Code Fragment q_c

(1-A) Case of No File-Name Attribute

Iteration	$\|R\|/\|SR\|$ GCS*	Koders	SPARS/R	Subtotal	Keywords in SQ
1	-/3847+	-/1145	-/503	-/5495	"capsule"
2	-/776+	-/67	1/9	1/852	1+"image"
3	-/571+	7/50	1/8	8/629	2+"write"
4	9/56	7/24	1/6	17/86	3+"readint"
5	21/29	4/4	0/0	25/33	4+"memreq"
6	7/7	4/4	0/0	11/11	5+"mipmapstate"
7	7/7	4/4	0/0	11/11	6+"filter"
8	7/7	4/4	0/0	11/11	7+"apply"

* GCS: Google Code Search
\+ These are the numbers of different files. The actual $|SR|$ is much larger.
\- No code clone filtering had been done due to long download time.

(1-B) Case of File Name as Input Attribute

Iteration	$\|R\|/\|SR\|$ GCS*	Koders	SPARS/R	Subtotal	Keywords in SQ
1	-/600	-/127	-/405	-/1132	Texture**
2	21/29	4/4	1/23	26/56	1+"capsule"
3	21/29	4/4	1/9	26/42	2+"image"
4	21/29	4/4	1/8	26/41	3+"write"
5	21/29	4/4	1/6	26/39	4+"readint"
6	21/29	4/4	0/0	25/33	5+"memreq"
7	7/7	4/4	0/0	11/11	6+"mipmapstate"
8	7/7	4/4	0/0	11/11	7+"filter"

** `Texture.java` is used for the file specifier of Google Code Search and Koders, and "`Texture`" is used for a search keyword for SPARS/R.

Table I shows the number of the output results, $|R|$, and also the number of the search results $|SR|$, classified by each search engine. (1-A) is the case where no file name or other attributes is given. (1-B) is the case where the file name "`Texture`" is given as an input attribute. Iteration means the trial process of Ichi Tracker as shown in Fig. 3 Here, we show the details of possible iterations from 1 to 8. This does not mean that Ichi Tracker always tries all of these iterations. It performs the search result analysis (Step 5) and the code clone filtering (Step 6) at a specific iteration only. By default of $|SR| \leq 50$ for each search engine, it stops and performs full processes at iteration 5 (GCS), 3 (Koders), and 2 (SPARS/R) for Case (1-A). For Case (1-B), it stops at iteration 2 for all search engines.

Keywords are the list of keywords given to each code search engine. In the Case (1-B), Google Code Search and Koders allow to input the file name, so we use the full file name as their input. SPARS/R does not accept the file name, so we give word "`Texture`" as one of its input keywords. In both cases, the output results share the code clones with

`Texture.java` whose cover ratio are 0.4 or higher.

As we can see these tables, the iteration converges fairly fast, and we get the output results. It is very clear that search results SR from three search engines initially contain many false positives, i.e., files containing no clones which will be filtered out by the code clone filtering. This situation is exemplified more clearly by the Case (1-A) where no file name is specified, since the file name is very important clue to get the same code fragment. However, even without the file name, we would get sufficient cloned results if we give an adequate keyword set. An interesting observation of Case (1-A) is, that the number of the output results (obtained from Google Code Search and remained after the code clone filtering) first increases along with the iteration, then it decreases. This is because an appropriate number of keywords recalls many possible candidates, but too many ones narrow the search results and eliminate possible candidates files with low cover ratios.

The total execution time for Case (1-A) was about 4 min. and Case (1-B) 1 min. Note that the manual overhead time for Koders is not included here. Most of the time was to search and download the source code files from 3 search engines. For example, in Case (1-B), the time for the word extraction and the code clone filtering was only 1 sec. and 4 sec., respectively. The rest was for searching and downloading.

The execution time is strongly affected by the response time of the code search engines and the network performance, and the total execution time varies time to time significantly.

Fig. 4 shows the distribution of 26 output results of Case (1-B) at Iteration 2. This figure is plotted by two attributes, the last modified time as x axis and the cover ratio as y axis. Project names of all 26 source code files including q_c have been investigated manually and listed on the right-hand side of the figure. They are sorted by the last modified time.

We have found four versions of jMonkeyEngine (#1, 4, 12, and 14-15). The evolution of those four versions is linked by the arrows in the figure. The circles are clusters of similar files which are exactly or 99% of lines are the same.

From this figure, we can observe the following.

- `Texture.java` code evolves along with the project progress. This is seen by the change of cover ratios over the versions. Only an exception is Revision 4490 (#14 and #15) which is almost the same as Revision 4099 (#12) (only 3 line difference).
- Each version of `Texture.java` is copied to many other projects, which are easily identified as similar files in Clusters A, B, and C.
- In the case of Cluster C, there are 6 files exactly the same as the query code q_c, which are lined up on the line of cover ratio 1.0. Five of those have been duplicated just after q_c was created. There is one outlier project #25, which was copied from jMonkeyEngine Revision 3800 fairly later after the new version Revision 4490 (#14 and #15) has been created.

Table II
NUMBER OF OUTPUT RESULTS $|R|$ AND SEARCH RESULTS $|SR|$ FOR `kern_malloc.c` AS QUERY CODE FRAGMENT q_c

(2-A) Case of No File-Name Attribute

Iteration	$\|R\|/\|SR\|$			Keywords in SQ
	GCS*	Koders	Subtotal	
1	-/429000+	-/4793	-/433793	"`freep`"
2	-/2221+	-/93	-/2314	1+"`caddr_t`"
3	-/114+	20/28	20/142	2+"`freelist`"
4	47/47	20/20	67/67	3+"`kb_next`"
5	47/47	20/20	67/67	4+"`WEIRD_ADDR`"

* GCS: Google Code Search
\+ These are the numbers of different files. The actual $|SR|$ is much larger.
\- No code clone filtering had been done due to long download time.

(2-B) Case of File Name as Input Attribute

Iteration	$\|R\|/\|SR\|$			Keywords in SQ
	GCS*	Koders	Subtotal	
1	55/74	22/25	77/99	"`kern_malloc.c`"**
2	53/53	21/21	74/74	1+"`freep`"
3	51/51	20/20	71/71	2+"`caddr_t`"
4	47/47	20/20	67/67	3+"`freelist`"
5	47/47	20/20	67/67	4+"`kb_next`"

** `kern_malloc` is used for the file specifier.

In addition to these attributes, we have extracted the licenses of the input query code and the output results. The input query code is under New BSD License (3-clause BSD License) [4], and the output results are also New BSD License except for two projects (#11 and #20) of zlib-libpng License [37].

Using Ichi Tracker, we are able to find many code clones for the input query code fragment. By analyzing those found files (fragments) and their attributes, we can easily and effectively identify evolution and propagation of the query code fragment.

B. Case Study 2: `kern_malloc.c`

`kern_malloc.c` is a C function, which allocates a specified-size memory block in the kernel address space. We have taken an old code from Lites project where Unix-like operating system had been developed [21]. This source code itself and its file name were used as the input query of Ichi Tracker.

The reason of using this file for the input query is that it was developed based on major Unix systems, 4.4 BSD and Mach microkernel. Also, it is fairly old, and it has been taken over and maintained by many other various projects.

We have executed Ichi Tracker in two cases with and without the file name `kern_malloc.c` associated. In this case study, only Google Code Search and Koders are used as the code search engines, since SPARS/R does not contain C files in its repository. Table II shows the output results and the search results. In the same manner as the previous case

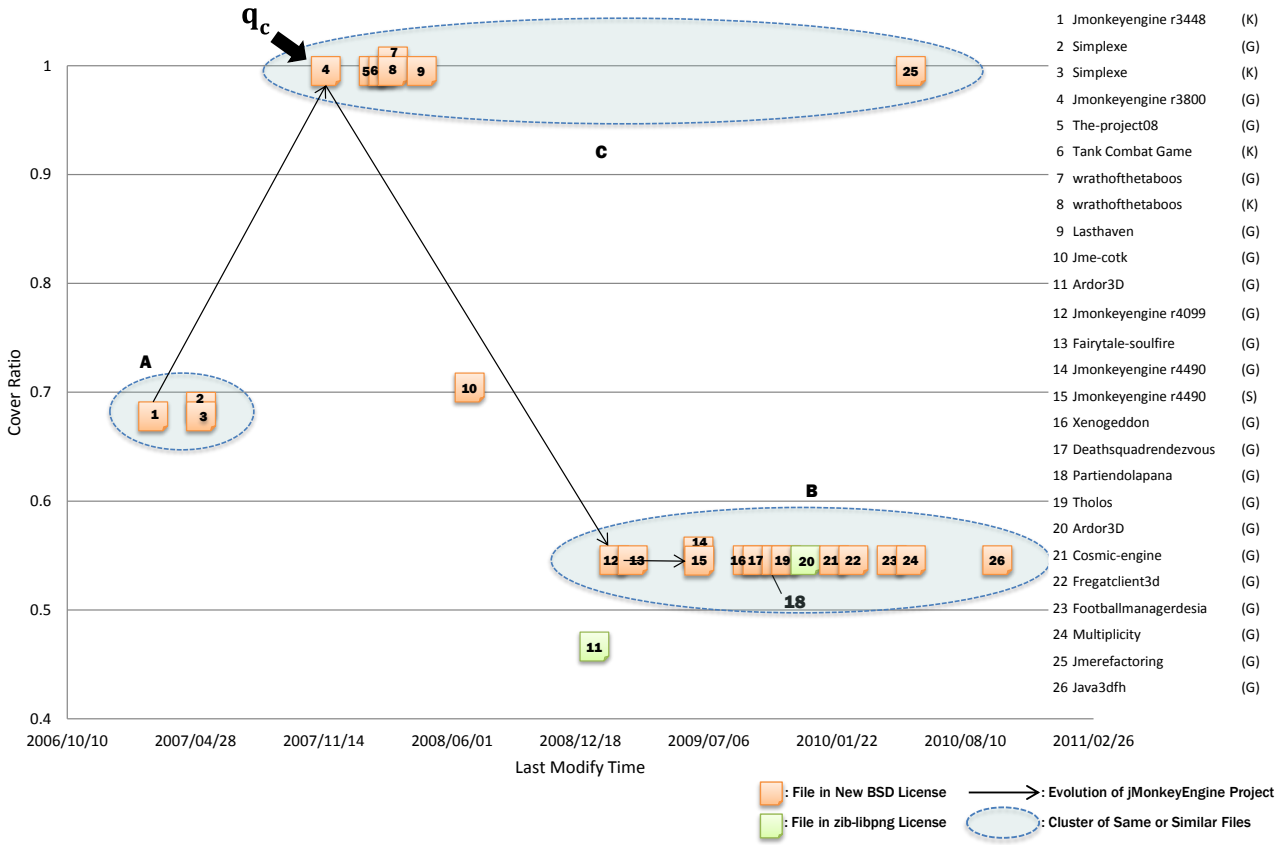

Figure 4. Distribution of Output Results of Case (1-B) at Iteration 2 (`Texture.java` with File Name)

study (Case Study 1), the iteration of the processes converges fairly fast even without the file name attribute specified as the query input. Also, the converged search results contains no false positive answer. This might suggest that the code clone filtering would not be needed if we give sufficient number of keywords and have the iteration converged. However, if we wait for the convergence, the search results become too narrow, and we might lose some cloned results (e.g., in Case (1-B), Google Code Search initially outputs 21 results, but it is reduced to 7 after the convergence).

The execution time was about 2 min. for Case (2-A) and also 2 min. for Case (2-B), where most of the time was occupied by searching and downloading from the code search engines, as described in Case Study 1.

Fig. 5 shows the distribution of 67 output results of Case (2-A) at Iteration 4 for Google Code Search and Iteration 3 for Koders, where no file name had been specified. From this figure, we observe the following.

- The cover ratio of the output results diverges along the time scale. Some of those remain very close to the query code q_c, but others get away from the query code.
- Unlike Case Study 1, there is no clear cluster of similar results. There are many variations of different code fragments, meaning there are many small changes among the projects.

All of these results are under BSD License (either Original BSD License (4-clause BSD License) or New BSD License (3-clause BSD License)). By using Ichi Tracker, we could easily overview the evolution of a core part of the Unix OS kernel code.

C. *Case Study 3:* `SSHTools`

SSHTools is a suite of Java SSH applications providing a Java SSH API, terminal, and so on [31]. Ignoring some tiny sized files, we have selected 339 files of the latest version 0.2.9 (last modified time is 6-23-2007), and made the tracking with those files as the input code fragments and file names. In this case study, we have set up a threshold of the cover ratio 0.4, and the resulting files with more than the threshold are considered as similar files.

Fig. 6 shows the number of similar files found for each query of the 339 SSHTool's files. We observe that 305 out of the 339 files contains code clones with other projects. 275 of them have less than 10 similar files for each, several

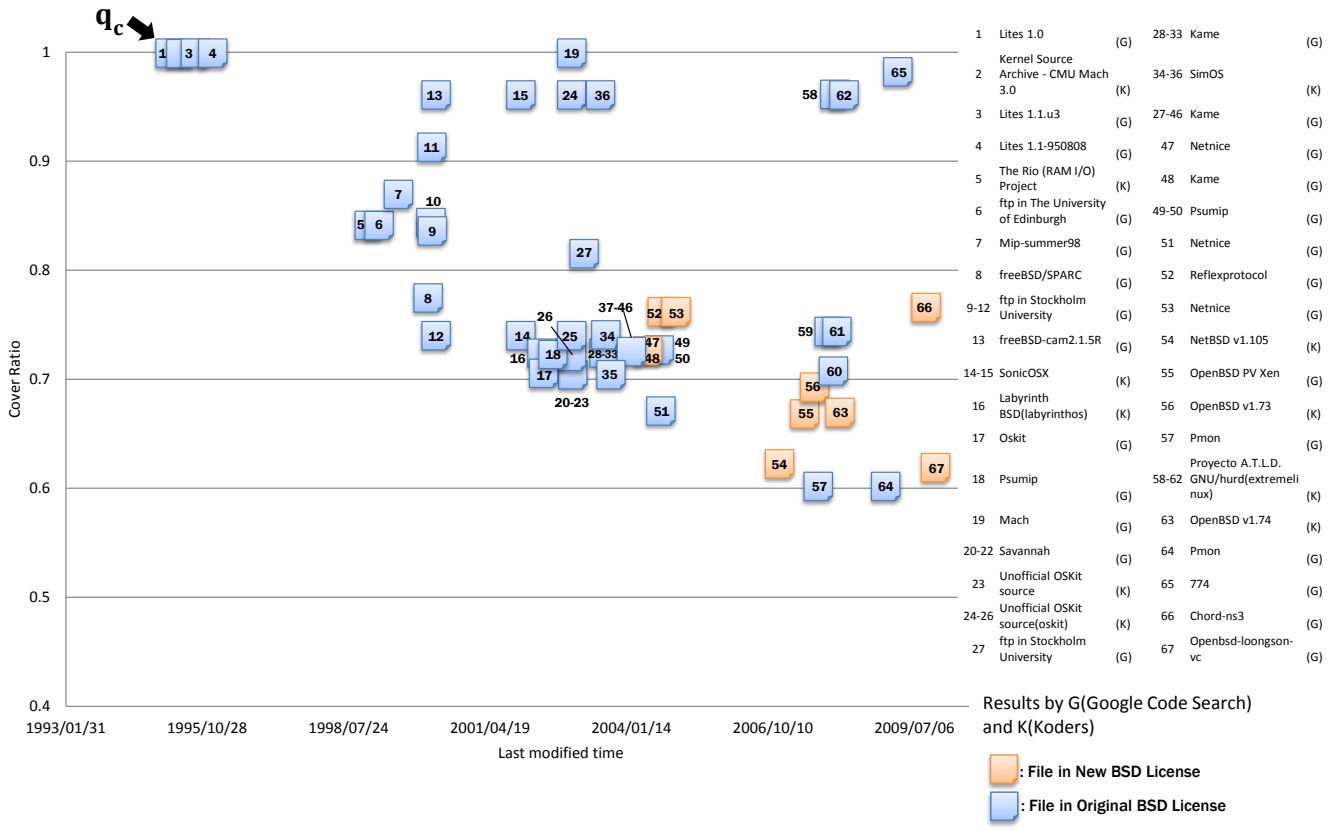

Figure 5. Distribution of Output Results of Case (2-A) at Iteration 4 (GCS) and 3 (Koders)(`kern_malloc.c` without File Name)

Table III
SEARCH RESULTS FOR SEVERAL FILES IN SSHTOOLS

Query File in SSHTools	Project Name of Found File	Cover Ratio of Found File	License	Copyright	Last Modified Time
SocketProxySocket.java	CVS client interface in Java	0.88	GPL 2	1998-99 Mindbright Technology AB	2001/11/12
Sftp.java	Apache Ant	0.62	Apache 1.1	2000-2002 Apache Software Foundation	2002/4/15
StringScanner.java	Programmer's Friend 4.1	0.81	CPL 1.0	2000-2003 Manfred Duchrow	2002/9/29
GeneralUtil.java	Gruntspud CVS Client	0.73	GPL 2	2002 Brett Smith	2003/11/12
Base64.java	Base64 notation	0.62	Public Domain	No copyright	2004/1/6
CharBuffer.java	Java Telnet daemon	0.81	GPL 2	2000 Dieter Wimberge	2004/1/16

All license and copyright of the query code files in SSHTools are in GPL 2 and "2002-2003 Lee David Painter and Contributors", respectively.

files have 10-30 similar files, and one file has more than 30 similar files.

We have also investigated the different licenses appeared in each similar file. SSHTools is under GPL 2 license; however, 298 files out of 339 files have similar files with different licenses from GPL 2. 10 files out of the 298 files have similar files with 2 different licenses, and 1 file has similar files with 3 different licenses.

Table III shows a part of the detailed analysis results. In this table, we present the oldest ancestor project for the query code. As seen in this table, there are many different ancestor projects. This means that SSHTools is a collection of various tools developed by several different projects. For each query file, there are similar files with the high cover ratios in those projects.

An intriguing observation would be evolution of licenses and copyrights of those files. Those found files had different licenses and copyrights, but they have been unified into GPL version 2 and "2002-2003 Lee David Painter and Contributors", respectively. This would suggest that those codes in different projects had been donated, and that their licenses and copyrights had been modified.

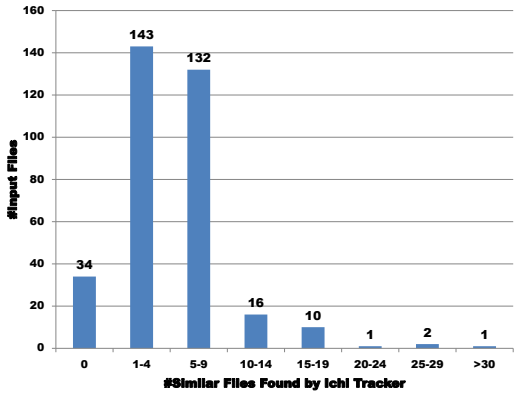

Figure 6. Histogram of Similar Files Found for Each Query File in SSHTools

V. Discussions

A. Application of Integrated Code History Tracking

As shown in the case studies, Ichi Tracker provides an overview history and evolution of a query source code. It spots the projects in which the query code locates, and it identifies the cloned code fragments in the same or different projects, which describe the evolution history of the original project.

This is a very important and needed feature when we reuse the source code at our hand. In Case (1-B) shown in Fig. 4, we can understand that the code q_c had been reused by other projects around late 2007 through early 2008 except for project #25. On the other hand, more than a half part of q_c has been reused by projects #12-#24 and #26 in Cluster B. This would suggest that we would prefer to reuse the newer and recently-created code rather than the older code q_c or its similar ones. If the newer code would provide the sufficient functionalities, we might choose to copy from project #26 rather than #4 (i.e., q_c).

In Case (2-A) in Fig. 5, various projects with different cover ratios are currently active, so we could choose an appropriate new one based on their functionalities.

One important application of using Ichi Tracker is to check license evolution. In our case studies, the licenses had evolved from New BSD License to Zlib-libpng License in Case (1-B), and from Original BSD License to New BSD License in Case (2-A). Those evolutions would be consistent and cause no trouble to reuse them. However, if we would find inconsistent licenses such as BSD License and GPL License in evolution, we have to care about reusing those codes. Ichi Tracker can easily check such inconsistency. In such sense, this system is very effective to identify plagiarism or illegal use of open source systems.

B. Approach and Processes of Ichi Tracker

1) Code Search Engines: In this implementation of Ichi Tracker, we have used 3 engines, Google Code Search, Koders, and SPARS/R. Our system might be seen as a meta code search system over these three search engines. However, the pre and post processes used here are not simple nor straightforward ones as usual meta search systems.

The output results of our case studies show that Google Code Search replies more output results than other engines. However, those output results do not always cover the output results of other engines, so the output results of other engines are still important. We can extend the external search engines for better results, but we need to create the interface program for each new search engine.

We may think that using the Internet search systems such as Google or Bing is more effective, rather than using the code search engines. We can easily try our case studies to those Internet search systems. We have tried those keywords listed in Table I and II with Google and Bing. The results contain various kinds of output, including search results from Google Code Search and Koders, raw source code files in open source repositories, compound source code forms with various extra explanations, e-mail archives, document files related to the query code, and many other unrelated files. Those might contain source code fragments which cannot be found by the code search engines, but extracting useful code parts from those various kinds of files would be an excessive challenge here.

2) Keyword Selection: As described in Section III, we have taken a strategy of selecting keywords for the code search engines, such that we choose n most frequently used keyword in the query code fragment, and n is initially 1 and is incremented by one until the search results become less than 50 for each engine. We could consider many other different algorithms and different parameter settings for the keyword selection.

At the beginning of the development of Ichi Tracker, we took another algorithm, which starts with n keywords (say $n = 20$) and decrements n by one for each loop until a sufficient number of results is obtained. In many cases, this strategy might eventually reach to the same results as the current implementation; however, choosing the initial value of n is not easy. Also, since we get sufficient results with 3–6 keywords in many cases, starting from one and incrementing n are faster than decreasing from 20.

We have investigated a strategy of using keywords only in the comment parts of the query code fragment. The output results heavily depend on the query source code, but a general tendency is that the search engines return many non-cloned source code files for a few keywords as their input. If we give more keywords in comments, those non-cloned files could be eliminated, but the number of the output results become less.

Also, we have investigated keywords from the source code part, which are used less frequently in that file. In such case, the output results heavily depend on the selection of the keywords whose frequency is only one. Most less

frequent keywords appear only once, and if we do not choose appropriate keywords specific to the query code, the query results from code search engines become fairly broad ones which contain many unrelated and non-cloned code fragments.

As an extreme strategy, we have tried randomly selected keywords from the code part. The result also heavily depends on the selected keywords, but a general tendency would be that in many cases, we get weaker search results by the random strategy in the sense that the results contain less cloned code fragments than the strategy of using most-frequently used keywords.

We have used keywords equal to or longer than 5 characters. Our investigation has indicated that including 4 or shorter keywords generates many unrelated search results from the code search engines. Also, breaking into smaller keywords from the camel case keywords (e.g., CamelCase $\rightarrow$ Camel and Case) and snake case keywords (e.g., snake_case $\rightarrow$snake and case) would eliminate the characteristics of the query code and might increase unrelated results.

3) Input and Output Attributes: Current implementation of Ichi Tracker uses the file name as only an input attribute associated with source code fragment. This attribute is passed to Google Code Search and Koders. Those search engines allow other attributes as their extra input such as language and license. If we would extend our system to accept those attributes, then the search results might be refined. However, we have to elaborate individual interface and to tune other parameters for better performance with those attributes.

As the attributes of the output results, we have mainly used the cover ratio and the last modified time. The cover ratio is computed automatically by the query code fragment and the code clone filtering results under the current implementation. The last modified time is obtained manually through the repositories whose location is presented by the code search engines. This extraction might be performed automatically, but there are many different types of repositories so that the implementation would not be simple.

Other output attributes such as license, developer, and project name are also important information to understand the code history and evolution more deeply. An approach of analyzing comments in the source code, similar to an automatic license detection method [8], will help to extract those attributes automatically.

C. Performance Issue

The case studies described in Section IV showed that Ichi Tracker required about 1 to 4 min. to get the output results. These response times might be slow as an interactive tool. However, the current implementation of Ichi Tracker is a prototype to validate our approach to the integrated code history tracker, so the performance of Ichi Tracker is not our main focus now.

There is room for a significant improvement of the performance. Currently, Ichi Tracker sequentially downloads the source code files after it receives the header lists of the search results from the code search engines. This process could be parallelized and speed up by using multiple download threads. However, the overall performance might be bounded by the performance of the code search engines and the network environment.

D. Quality of Search Result

It is not straightforward to evaluate the quality of the search result of Ichi Tracker, since this tool heavily depends on external search engines whose detailed insides are not known to us. Especially, the source code repositories for those external engines should be investigated for the recall computation of our tool, but we cannot do it.

We would consider the ratio $|R|/|SR|$ is an indicator of the precision of the external search engines. For example, as shown in Table I-A, 25/33 at iteration 5 gives 0.76. This does not indicate that we will have 24% false positive results in the final output. The code clone filter will remove those false positive results, so that the remaining final results will always contain code clones for the input query q_c.

E. Threats to Validity

The empirical studies we have conducted have several threats to validity.

First, we have selected three targets for the case studies. If we had chosen different targets, then we might have different consequences. We have several other trials with different target files, and have found no significant differences from those case studies. However, we would need to accumulate the experiences of various kinds of targets.

We have used CCFinder as a code clone detector for filtering out the unrelated query results, and it may report false positives, as mentioned before. We would assume the ratio of the false positive is fairly low, but we might need to check with different code clone detectors.

VI. Related Works

A. Origin and Evolution of Code

There are many research studies on analyzing and tracing code origin, provenance, evolution, genealogy, and so on through code clone analysis [9], [16], [17], [19], [24], [32]. Duala-Ekoko et. al propose Clone Tracker to trace and manage code clone history [7]. They have developed a tool for supporting clone tracking, with an abstract clone information named clone region descriptor. Davies et. al. propose Software Bertillonage for determining the origin of code entities with anchored signature matching method [6].

These researches are closely related to our work. However, their objectives are different from ours in the sense that they analyze various characteristics of code fragment in their local repositories. In our case, we analyze the origin and evolution of the query code in Internet repositories.

B. Code Search Engines

Code search is not only a very emerging research area, but also a very useful resource for software engineers these days [1]. We have used Google Code Search, Koders, and SPARS/R as the code search engines here. In addition to these ordinary keyword-based search engines, many complicated search mechanisms have been proposed. Javacio is a meta search engine for source code, JAR files, and documents, which executes a query for a keyword set and returns search results using Google Code Search, Koders and others [13]. Exemplar is a code search engine which expands the user's query keywords to API calls by a dictionary made by help documents [11]. CodeBroker is an interactive development tool to support code completion by searching and providing useful code fragments in the repository, which flexibly extracts various information from a partial code fragment on edit, and finds appropriate artifacts [35]. There are many other approaches to code search, and Grechanik el. al. have well summarized and classified those engines in [11]. However, none of these engines have features of accepting code fragments as their inputs or filtering out non-clone search results.

PARSEWeb is similar to our system in the sense they use code search engines for collecting source code [33]. It uses a type matching query such as $Source \rightarrow Destination$, and generates method invocation sequences as the output. Also, PARSEWeb performs a code analysis to extract the method invocation sequences. Our approach uses the code search engines as the resource of code collection too, but the query input is a code fragment, and the resulting output is the code fragments containing code clones with the query code fragment. Also, we use the code clone filter for the elimination of unrelated code fragments returned from the code search engines.

LChecker takes a similar approach to the license compliance of the target source code file [36]. It tokenizes the input source file, and makes a query to Google Code Search. The resulting license information is compared to the query file. This system targets only the license compliance without tracking overall history of code.

There are various different code search engines with different types of query inputs and search mechanisms, but none of those provides the code search features with both the code fragment query input and the code clone filter.

C. Open Source Repositories

There are many useful open source repositories on the Internet. One example is SourceForge [29], in which we can find thousands of very active open source development projects. However, it does not provide a precise code search feature but it shows the overall project information. Schwarz et. al. proposes an idea of linking method of clones across repositories [28], without actual implementation. Sourcerer is a large-scale software repository with keyword and fingerprint based search features [23]. These repositories will be very important and useful resources of the open code clone search if they would provide sufficient features to locate the specific source code fragments.

D. Code Clone Detection and Management

There are many active researches on code clone detection and analysis [5], [27]. Among those, there are works focusing on code clone search with scalability and performance for the large scale repositories. Lee et. al. proposes a clone indexing method for detecting similar code fragment in a large repository [20]. Keivanloo et. al. also proposes a hybrid approach to real-time and scalable code clone search using two types of indexing [18]. Those are important and useful techniques for the code clone search for the local repositories; however, to explore code history of open source systems, we have to collect a huge amount of code and to keep updated everyday by ourselves. Our approach does not require such overhead.

VII. Conclusion

In this paper, we have proposed an integrated model to code history tracking, and presented the detailed processes of Ichi Tracker which is a prototype system for the model. We have conducted experiments with several case studies, which show the applicability and effectiveness of our approach.

There are several future works. One is to improve the performance and usability of the current prototype implementation of Ichi Tracker, by which users can use the system interactively. Another would be to explore a unified approach of local repositories and Internet repositories, by which we might get more better recall with sufficient performance.

One interesting idea to extend the history tracking is to use the search results as the new search queries. Repeated tries of this loop would change the queries gradually from the original code, and might produce morphed code. Tracking such code chain will be a new challenge.

Acknowledgment

We are grateful to the anonymous reviewers for their useful comments. Our colleagues, Yoshiki Higo, Norihiro Yoshida, and Takashi Ishio, have contributed to this work for their valuable suggestions and supports. This work has been partially supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (A) (No.21240002), Exploratory Research (No. 23650015), and Global COE Program (Founding Ambient Information Society Infrastructure), and also by Mext for the Development of Next Generation IT Infrastructure (the Stage Project).

REFERENCES

[1] S. Bajracharya, A. Kuhn, and Y. Ye (ed.), "Workshop on Search-driven Development: Users, Infrastructure, Tools and Evaluation", Cape Town, South Africa, May 2010.

[2] S. Bellon, R. Koschke, G. Antiniol, J. Krinke, E. Merlo, "Comparison and Evaluation of Clone Detection Tools", IEEE Trans. on Software Engineering, Vol. 33, No. 9, pp. 577-591, Sep. 2007.

[3] Black Duck Koders, http://www.koders.com/.

[4] The BSD License, http://www.opensource.org/licenses/bsd-license.

[5] J. Cordy, K. Inoue, R. Koschke, and S. Jarzabek (ed.), "5th International Workshop on Software Clones (IWSC 2011)", Honolulu, Hawaii, May 2011.

[6] J. Davies, D. M. German, and M. W. Godfrey, "Software Bertillonage: Finding the Provenance of an Entity", Proc. of Working Conference on Mining Software Repositories (MSR 2011), pp. 183-192, Honolulu, Hawaii, May 2011.

[7] E. Duala-Ekoko, M. P. Robillard, "Clone Region Descriptors: Representing and Tracking Duplication in Source Code", ACM Tran. on Software Engineering, Vol. 20, No. 1, Article 3, pp. 3.1-3.31, Jun. 2010.

[8] D. German, Y. Manabe, and K. Inoue, "A Sentence-Matching Method for Automatic License Identification of Source Code Files", Proc. of Automatic Software Engineering, Antwerp, Belgium, pp.437-446, Sep. 2010.

[9] M. Godfrey, and L. Zou, "Using Origin Analysis to Detect Merging and Splitting of Source Code Entities", IEEE Tran. on Software Engineering, Vol. 31, No. 2, Feb. 2005.

[10] Google Code Search, http://www.google.com/codesearch.

[11] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. M. Cumby, "A Search Engine for Finding Highly Relevant Applications", Proc. of 32th International Conference on Software Engineering (ICSE 2010), pp. 475-484, Cape Town, South Africa, May 2010.

[12] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking Significance of Software Components Based on Use Relations", IEEE Trans. on Software Engineering, Vol. 31, No. 3, pp. 213-225, Mar. 2005.

[13] javacio.us, http://javacio.us/.

[14] jMonkeyEngine 3.0, http://jmonkeyengine.com/.

[15] T. Kamiya, S. Kusumoto, K. Inoue: "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code", IEEE Trans. on Software Engineering, Vol. 28, No. 7, pp. 654-670, July 2002.

[16] C. Kapser, and M. W. Godfrey, "'Cloning considered harmful' considered harmful: Patterns of cloning in software", Empirical Software Engineering, Vol. 13, No. 6, pp. 645-692, 2008.

[17] S. Kawaguchi, P. K. Garg, M. Matsushita, and K. Inoue, "MUDABlue: An Automatic Categorization System for Open Source Repositories", J. of Systems and Software Vol. 79, No. 7, pp.939-953, 2006.

[18] I. Keivanloo, J. Rilling, and P. Charland, "SeClone - A Hybrid Approach to Internet-Scale Real-Time Code Clone Search", Proc. of 19th International Conf. on Program Comprehension, pp. 223-224, Kingstion, Canada, June, 2011.

[19] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," Proc. of Foundations of Software Engineering (ESEC/FSE 2005), Vol. 30, No. 5, pp. 187-196, Lisbon, Portugal, Sep. 2005.

[20] M. Lee, J. Roh, S. Hwang, and S. Kim, "Instant Code Clone Search", Proc. of 18th International Symposium on Foundations of Software Engineering, pp. 167-176, Santa Fe, NM, Nov. 2010.

[21] Utah Lites Release 1.1.u3, http://www.cs.utah.edu/flux/lites/html/.

[22] S. Livieri, Y. Higo, M. Matsushita, K. Inoue, "Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder", Proc. of 29th International Conference on Software Engineering (ICSE 2007), pp.106-115, Minneapolis, MN, May 2007.

[23] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi,"Sourcerer: Mining and Searching Internet-scale Software Repositories",Data Mining and Knowledge Discovery, Vol. 18, No. 2, pp. 300-336, April 2009.

[24] A. Lozano, M. Wermelinger, B. Nuseibeh, "Evaluating the Harmfulness of Cloning: A Change Based Experiment", Proc. of Mining Software Repositories (MSR 2007), p. 18-21, Minneapolis, MN, May 2007.

[25] C. Ebert (ed.), "Open Source Software in Industry", IEEE Software, Vol. 25, No. 3, pp. 52-53, May/June 2008.

[26] Maven Central Repository, http://search.maven.org/.

[27] C. K. Roy, James R. Cordy, R. Koschke, "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach", Science of Computer Programming, Vol. 74, No. 7, pp. 470-495, 2009.

[28] N. E. Schwarz, E. Wernli, and A. Kuhn, "Hot Clones, Maintaining a Link between Clones across Repositories", Proc. 4th International Workshop on Software Clones (IWSC 2010), pp. 81-82, Cape Town, South Africa, May 2010.

[29] SourceForge, //http://sourceforge.net/.

[30] SPARS Project, http://sel.ist.osaka-u.ac.jp/SPARS/index.html.en.

[31] SSHTools Source Repository, http://sourceforge.net/projects/sshtools/.

[32] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta, "An empirical study on the maintenance of source code clones", Empirical Software Engineering, Vol. 15, No. 1, pp. 1-34, 2009.

[33] S. Thummalapenta, T. Xie, "PARSEWeb:A Programmer Assistant for Reusing Open Source Code on the Web", Proc. of 22nd International Conference on Automated Software Engineering (ASE 2007), pp. 204-213, Atlanta, GA., Nov. 2007.

[34] S. Uchida, A. Monden, N. Ohsugi, T. Kamiya, K. Matsumoto, and H. Kudo, "Software Analysis by Code Clones in Open Source Software", Journal of Computer Information Systems, Vol. 45, No. 3, pp. 1-11, 2005.

[35] Y. Ye, and G. Fischer, "Supporting Reuse by Delivering Task-Relevant and Personalized Information", Proc. of International Conference on Software Engineering (ICSE 2002), pp. 513-523, Orlando, FL, May 2002.

[36] H. Zhang, B. Shi, and L. Zhang, "Automatic Checking of License Compliance", Proc. of 26th International Conf. on Software Maintenance, Timisoara, Romania, Sep. 2010.

[37] The zlib/libpng License, http://www.opensource.org/licenses/zlib-license.