A Method for Analyzing Code Homology in Genealogy of Evolving Software

Masatomo Hashimoto[1] and Akira Mori[2]

[1] AIST Tokyo Waterfront, 2-3-26 Aomi, Koto-ku, Tokyo 135-0064, Japan,
m.hashimoto@aist.go.jp
[2] AIST Tokyo Akihabara Site, 1-18-13 Sotokanda, Chiyoda-ku,
Tokyo 101-0021, Japan, a-mori@aist.go.jp

Abstract. A software project often contains a large amount of "homologous code", i.e., similar code distributed in different versions or "species" sharing common ancestry. Homologous code fragments are prone to incur additional maintenance efforts since changes of their common characters must be replicated on each code fragment to keep the system consistent or up to date. We propose an automated method for detecting and tracking homologous code in genealogy of evolving software using fine-grained tree differencing on source code. Locating homologous code and tracking its course of change would help software developers/maintainers to better understand the source code and to detect/prevent inconsistent modifications that may lead to latent errors. To show the capability of the method, the results of experiments on several large-scale software are reported including BIND9 DNS servers, a couple of Java software systems jEdit and Ant, and Linux device drivers.

1 Introduction

A large software system often contains a large number of similar code fragments across many versions or branches. They are typically introduced when the code is inherited from previous versions, duplicated for programming convenience, and patched to correct common defects. We call such correspondence in the code descended from a common ancestry *homology* of code, by analogy with biology [1].

Homologous code fragments or *homologues* are similar code fragments distributed in different versions or "species" sharing common ancestry. They may evolve in uniform or divergent manner as the development proceeds. If the evolution is uniform, it is likely that there exists a common programming logic and additional maintenance efforts are necessary [2] since further changes must be replicated on each code fragment to keep the system consistent. Even when the evolution is divergent, common characters of the code remain in later developments [3].

Locating a homologue code and tracking its course of change would help software developers/maintainers to better understand the source code and to detect/prevent inconsistent modifications that may lead to latent errors. The

task can be difficult even when a versioning system such as CVS is in place to record change descriptions since such information is too coarse to compare and usually not associated with reference to concrete source code entities such as functions and methods[2].

Code clone is a well-researched topic relating to code homology. Many algorithms and tools have been proposed for detecting code clones [4]. However, those methods are not well-suited for analyzing how clone regions evolve over time since maintaining clone relations is difficult when regions go through different modifications and do not remain the same. Clone detection must be performed on each version and discovered clone regions must be tracked in the later versions. To distinguish newly introduced clone regions from those lasting from previous versions involves an awkward task of adjusting similarity thresholds by heuristics [3]. The method for tracking cloned code is relatively less explored with several exceptions [5, 3, 6–8] despite its practical importance.

In this paper, we propose an automated method for detecting and tracking homologues in genealogy of evolving software using a fine-grained tree differencing tool called Diff/TS[9] for source code. The method also reconstructs semantic change histories from raw edit sequences computed by differencing on abstract syntax trees and identifies inconsistent changes by comparing change histories of homologues. By following ideas from biology, we classify homology into three categories: *orthology*, *paralogy*, and *xenology*. Orthology describes homology arising from branching activity, xenology from exchange of code across different branches, and paralogy from duplication in a single branch. See Fig. 1 for illustration.

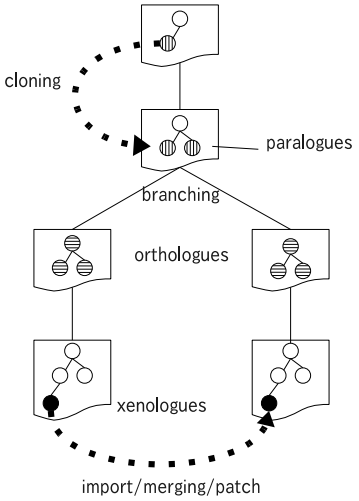

Fig. 1. Types of Code Homology

We implemented procedures for analyzing homologues. To show the capability of the method, the results of experiments on several large-scale software are reported including BIND9 DNS servers, a couple of Java software systems jEdit and Ant, and Linux device drivers. Xenology is investigated with BIND9 and paralogy is investigated with jEdit, Ant, and Linux device drivers.

The results shows that the proposed method is efficient enough to analyze the device drivers in 32 versions of Linux from 2.6.0 to 2.6.31, each of which consists of millions of lines of source code. Several inconsistencies in Linux serial drivers have been detected which violate a development policy concerning kernel locking. It is also shown that the method produces better analysis results compared to existing code clone trackers. In fact, the system could track not only all clone regions reported in the previous literature [5, 6, 3, 7, 8], but also regions that escaped from previous analysis [6].

To summarize, the contributions of the paper are:

- proposal of the notion of code homology to categorize similar code in genealogy of evolving software,
- development of an automated method for detecting and tracking homologues, and
- development of an automated method for reconstructing and comparing fine-grained change histories for homologues.

The rest of the paper is organized as follows. Prerequisites for tree differencing used in code homology analysis are explained in Sect. 2. Section 3 describes the method for code homology analysis. The results of the experiments is reported in Sect. 4. After related work is reviewed in Sect. 5, we conclude in Sect. 6.

2 Tree Differencing

We regard a version of a software system as a set of abstract syntax trees (ASTs) which correspond to source files (compilation units) and also as a directory tree which consists of source files. Tree differencing plays important role in analyzing code homology. Code homology analysis consists of three steps: detecting homologous code, tracking homologous code, and tracking changes of homologous code. We enumerate required functions in each step, whose details are explained in Sect. 3.

1. Homologous code detection
   (a) discovering common structures between a pair of ASTs — for orthologues
   (b) differencing a pair of ASTs — for xenologues
   (c) detecting code clones — for paralogues
2. Code fragment tracking
   (a) mapping nodes in one AST to corresponding nodes in another AST
   (b) detecting cloning activities
   (c) mapping line ranges in source code to the corresponding sub-ASTs and vice versa
3. Change history reconstruction
   (a) deriving higher level descriptions from low level descriptions for changes between a pair of ASTs

Tree differencing algorithms employed in code homology analysis are responsible for 1-(a), 1-(b), 2-(a), and 3-(a).

A basic function of a tree differencing algorithm is to calculate a sequence of *edit operations* (called *edit sequences*) that transforms T_1 into T_2 for a pair of trees T_1 and T_2. The basic edit operations are label renaming, deletion, and insertion of nodes. Some algorithms can also detect moves of subtrees. We regard an edit sequence between two ASTs as difference between them. Another basic output of the algorithm is a set of matched pairs of nodes between target trees. We call such set of matched pairs of nodes a *node map*, which may be regarded as a (partial) finite map. Note that labels of a matched pair of nodes

do not necessarily coincide. Basically, node maps are constrained to preserve the structure of target trees up to relabeling. A common part of a couple of ASTs T_1 and T_2 with respect to a node map M is defined as a pair of sets of nodes $(\mathrm{dom}(M), \mathrm{cod}(M))$, where $\mathrm{dom}(X)$ and $\mathrm{cod}(X)$ denote domain and codomain of mapping X, respectively.

We can use any tree differencing algorithms or tools which satisfy requirements for code homology analysis. In this study, we used a tree differencing system called Diff/TS[9]. Diff/TS extends a general tree comparison algorithm with heuristics driven control configurable for multiple programming languages. Diff/TS is capable of processing Python, Java, C and C++ projects, and provides all required functions but 3-(a). In addition to low level edit sequences, higher level description of source code changes is necessary for change history construction. We added a module for classifying changes of C programs into Approximately 80 change types defined following Fluri and Gall [2]. Change types are defined by edit operations and syntactical information embedded in ASTs. For example, a change type **function call inserted** is defined as insertion of a subtree corresponding to function call and **return value changed** as some edit operation(s) on AST node(s) in a subtree corresponding to a return value.

3 Code Homology Analysis

This section presents procedures for detecting and tracking homologous code in a given set of branches. An overview of our key techniques for code homology analysis is also provided.

The tree differencing algorithm allows us to distinguish between the preserved portion (up to relabeling) and the added/deleted portion between a pair of versions, and to compute edit operations which transform one into another. We employ a "double-differencing" technique to identify xenologues. First, we pairwise compute preserved portions between relative versions and conclude that the preserved portion found closest to the most recent common ancestor represents orthologues. Then, we compute differences between preserved portions to identify added code fragments as xenologues. See Fig. 2 for illustration, where version A branches into versions B and C, which then evolve into $B1$ and $C1$, respectively. A vertical dashed line denotes difference, in which a black triangle represents code addition and a white triangle represents code deletion. A horizontal dashed line denotes common code segments, i.e., homologues between relative versions. Segments that have been newly added in the homologues suggest existence of merged or patched code, which we call xenologues (e.g., a shaded black triangle in Fig. 2). Other segments that have disappeared from the homologues suggest existence of individually modified code. In general, we cannot decide whether xenologues are originated from code exchange or simultaneous patch application without manually inspecting available documents such as change-logs or development histories.

Paralogues, that is, code clones or duplicated code, are detected in a different manner. For the (given) initial version of the software, we apply existing code

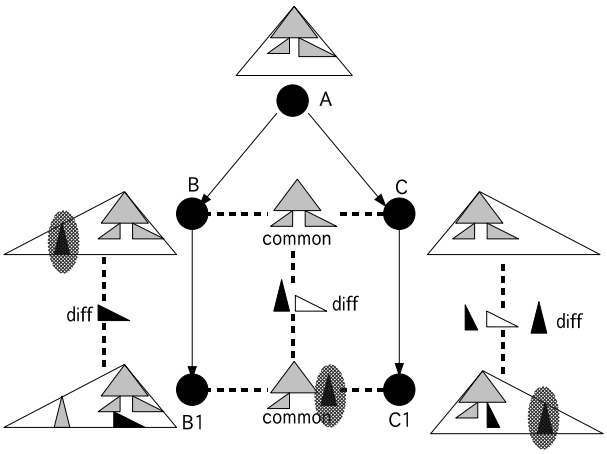

Fig. 2. Double-Differencing Between Branches

clone detection tools and accept the outputs as clone groups generated before the first version. For identifying cloning activities after the first version, we first compute added code fragments by tree differencing, and then find its potential duplication origins in the previous versions. A common token sequence matching algorithm is used for this.

We introduce several notations and terms needed for the rest of the paper. The set of nodes in A is denoted by $\mathcal{N}(A)$. By $\mathcal{I}(A)$, we denote the set of post-order indexes of an AST A. We identify nodes with their post-order indexes. Let v be a version of software system. We denote the set of ASTs corresponding to source files contained in v by $\mathcal{A}(v)$. For a tree A and $S \subseteq \mathcal{N}(A)$, $A|_S$ denotes the tree obtained from A by removing all nodes that do not belong to S. We introduce two wrapper functions of Diff/TS denoted by Δ for ASTs and Δ_d for directory trees. For ASTs A_1 and A_2, $\Delta(A_1, A_2)$ computes a triple (M, D, I), where M denotes a node map such that $M \subseteq \mathcal{I}(A_1) \times \mathcal{I}(A_2)$, D a set of deleted components, and I a set of inserted components. For versions v_1 and v_2, $\Delta_d(v_1, v_2)$ computes (M, D, I), where M denotes a node map such that $M \subseteq \mathcal{A}(v_1) \times \mathcal{A}(v_2)$, D a set of deleted source files, and I a set of inserted source files. Note that internal nodes (directories) are omitted from M. We extend the domain of Δ to the set of versions: $\Delta(v_1, v_2) = \{(A_1, A_2, M, D, I) | (M, D, I) = \Delta(A_1, A_2)$, where $(A_1, A_2) \in M_d, (M_d, D_d, I_d) = \Delta_d(v_1, v_2)\}$.

In order to detect homologues, we compute *common code structures* (CCSs) between versions. A CCS between two ASTs A_1 and A_2 with respect to a node map M, denoted by $\mathrm{CCS}_M(A_1, A_2)$, is defined as a pair of trees (S_1, S_2) where $S_1 = A_1|_{\mathrm{dom}(M)}$ and $S_2 = A_2|_{\mathrm{cod}(M)}$. A CCS between two versions is defined as a set of CCSs between ASTs which corresponds to source files matched by Δ_d. We can compute a CCS between versions v_1 and v_2 by first computing $(M_d, D_d, I_d) = \Delta_d(v_1, v_2)$, and then computing $(M, D, I) = \Delta(A_1, A_2)$ and $(A_1|_{\mathrm{dom}(M)}, A_2|_{\mathrm{cod}(M)})$, for each $(A_1, A_2) \in M_d$.

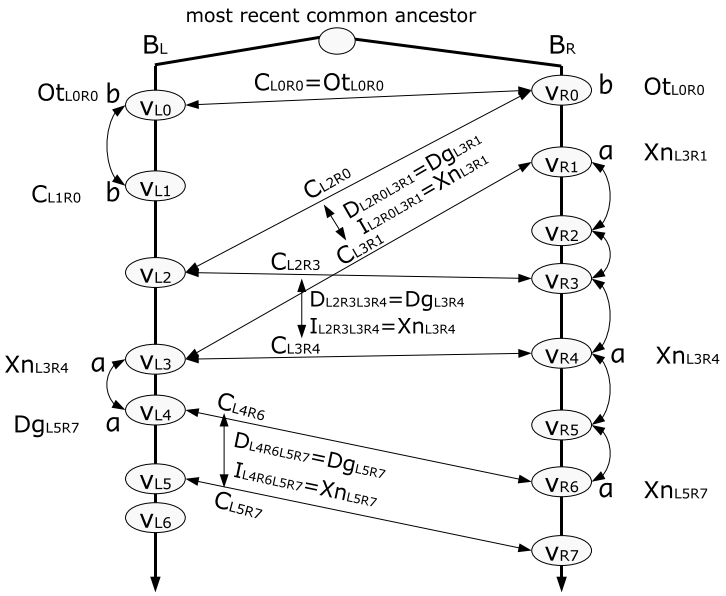

Fig. 3. Detecting Homologues

3.1 Detecting Homologues

An orthologue is defined between versions of different branches. For a pair of branches B_L and B_R diverged from a common ancestor, an orthologue between the oldest versions of B_L and B_R is defined as a CCS between the oldest versions. See Fig. 3, where C_{L0R0} and Ot_{L0R0} denote a CCS and an orthologue between versions v_{L0} and v_{R0} of branches B_L and B_R. Once we have obtained an orthologue between the oldest versions, orthologues between other versions of B_L and B_R is obtained by tracking the oldest orthologue. Our method of tracking code fragments is described in Sect. 3.2.

In order to detect xenologues, we must perform differencing one more time on CCSs. Let B_L and B_R be branches that stem from a common ancestor (See Fig. 4). For versions v_{Lb} and v_{Rb}, we compute $\mathrm{Xn}(v_{Lb}, v_{Rb})$ which denotes the set of xenologues between v_{Lb} and v_{Rb}. We let $\Delta_d(v_{Lb}, v_{Rb}) = (M_{LbRb}, D_{LbRb}, I_{LbRb})$ where $M_{LbRb} = \{(1,1),(5,3)\}$, where nodes are indicated by indexes (by post-order traversal). Similarly, for (v_{La}, v_{Ra}), (v_{La}, v_{Lb}), and (v_{Ra}, v_{Rb}), we let $M_{LaRa} = \{(1,1),(3,3)\}$, $M_{LaLb} = \{(1,2),(3,5)\}$, and $M_{RaRb} = \{(3,3)\}$. Among the pairs contained in M_{LbRb}, only $(5,3) = (A_{Lb5}, A_{Rb3})$ is able to form a commutative diagram consisting of dashed arrows in Fig. 4. Similarly in M_{LaRa}, only (A_{La3}, A_{Rb3}) form a diagram. We apply Δ to (A_{Lb5}, A_{Rb3}) and (A_{La3}, A_{Ra3}) to obtain CCSs for them. Let $M_b = M_{Lb5Rb3}$ and $M_a = M_{La3Ra3}$ be node maps obtained from the applications, respectively. By definition, $\mathrm{CCS}_{M_b}(A_{Lb5}, A_{Rb3}) = (A_{Lb5}|_{\mathrm{dom}(M_b)}, A_{Rb3}|_{\mathrm{cod}(M_b)})$ and similarly we have $\mathrm{CCS}_{M_a}(A_{La3}, A_{Ra3}) = (A_{La3}|_{\mathrm{dom}(M_a)}, A_{Ra3}|_{\mathrm{cod}(M_a)})$. We let $C_{Lb5Rb3} = A_{Lb5}|_{\mathrm{dom}(M_b)}$ and $C_{La3Ra3} = A_{La3}|_{\mathrm{dom}(M_a)}$. Finally, we apply Δ to C_{La3Rb3} and C_{Lb5Rb3}. Let $(M, D, I) = \Delta(C_{La3Rb3}, C_{Lb5Rb3})$. I corresponds to $\mathrm{Xn}(A_{Lb5}, A_{Rb3})$ and D "degenerated" homologues between A_{Lb5} and A_{Rb3}, denoted by $\mathrm{Dg}(A_{Lb5}, A_{Rb3})$. Note that we

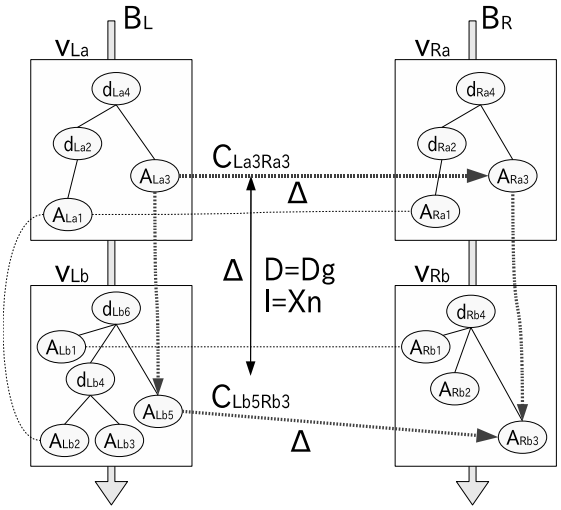

Fig. 4. Differencing Common Code Structures

can choose $A_{Rb3}|_{\mathrm{cod}(M_b)}$ for C_{Lb5Rb3} or $A_{Ra3}|_{\mathrm{cod}(M_a)}$ for C_{La3Ra3} since we ignore the difference of node labels in the node maps.

As mentioned in the beginning of Sect. 3, it is impossible to determine the origin of xenologues in general. For example, in Fig. 3, suppose that there exists some $a \in \mathrm{Xn}_{L3R4}$ and it also exists v_{L3} through v_{L4}, and v_{R1} through v_{R6}. We can not decide whether a is introduced by simultaneous patch application to v_{L3} and v_{R1} or by copying some part from a revision between v_{R1} and v_{R6} to v_{L3}.

We use existing tools for identifying paralogues (code clones) in the initial version of the given software versions. For the versions descending from the initial versions, we use a code tracking method described in the next section for detecting cloning activities.

3.2 Tracking Code Fragments

Once the occurrence of a homologue is discovered, we look into a *code continuum* to inspect developments in the subsequent versions. A code continuum is a data structure created by composing differencing results across versions to record entire lifetime of a source code entity. A code continuum can be illustrated by a set of *node continua*, that is, threads representing the lifetime of AST nodes over time as in Fig. 5, where the beginning and the ending of a thread indicate the introduction and the removal of the AST node, respectively.

A node continuum is constructed for each AST node. For the same reason as we perform directory tree differencing, we construct a *file continuum* for each source code file. Each node/file continuum stores trace information of an AST node and a source file, respectively. A continuum for versions $v_0, \ldots, v_n$ is represented by a sequence of names $\langle N_0, \ldots, N_n \rangle$, where $N_i (0 \leq i \leq n)$ is a name of the node/file in version v_i. The node name is given by its index in post-order traversal and the file name by its path name. Non-existence of

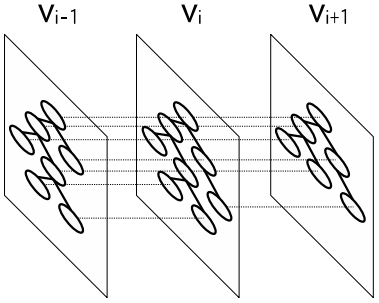

Fig. 5. Code Continuum

Alg.1. Continuum Construction

```
1:  procedure ConstCTM(⟨T_0, T_1, ... T_n⟩, K)
2:      K ← ∅
3:      for 0 ≤ i ≤ n − 1 do
4:          M_i = M(T_i, T_{i+1})
5:          Mapped ← ∅
6:          for k = ⟨N_0, ..., N_l⟩ ∈ K do
7:              k ← ⟨N_0, ..., N_l, M_i(N_l)⟩
8:              Mapped ← Mapped ∪ {N_l}
9:          end for
10:         for x ∈ dom(M_i) do
11:             if x ∉ Mapped then
12:                 K ← K ∪ {⟨ϵ, ..., ϵ, x, M_i(x)⟩}
                                 (ϵ: i times)
13:             end if
14:         end for
15:     end for
16: end procedure
```

the node/file is represented by an empty name ϵ for convenience. An algorithm for constructing continua is shown in Alg. 1. In the description of continuum construction algorithm, $\mathcal{M}$ denotes a wrapper function of Diff/TS. For a pair of trees T_1 and T_2, $\mathcal{M}(T_1, T_2)$ computes a tree map between T_1 and T_2. The result is stored in K.

Since our AST nodes contain location information such as file names, line numbers, column positions and file offsets, continua make various analysis tasks easy including tracking corresponding source code entities on texts and reconstructing change sequences for a given code segment according to the results of source code tree differencing.

3.3 Detecting Cloning Activities

While traditional clone detection tools can discover code clones among given sets of software versions, they do not support tracking discovered code clones in the subsequent versions of software. To cope with the problem, we rely on code continua for identifying cloning activities by looking for the original code of cloning in the previous version by way of token based sequence matching algorithm.

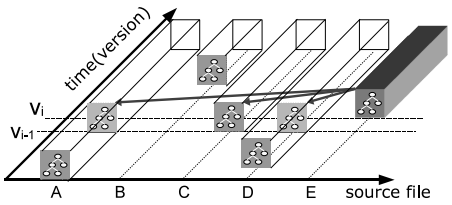

Fig. 6. Cloning Activity Detection

Suppose that we analyze a sequence $v_0, v_1, v_2, \ldots, v_n$ of software versions. Clones that exist already in the initial version v_0 are marked using existing tools such as SimScan[3] for Java code and CCFinder [10] for C code. Cloning activities taking place between v_{i-1} and v_i ($i \geq 1$) are identified by the following procedure:

1. collect node continua starting from v_i to form a tree T included in the AST of v_i,

[3] `http://www.blue-edge.bg/simscan/simscan_help_r1.htm`

2. convert T into a sequence p of tokes by pre-order traversal,
3. compare p with token sequences obtained from ASTs in v_{i-1} by pre-order traversal using an O(ND) algorithm [11],
4. compute a score for each match by (number of matched tokens)/(number of tokens in p),
5. select the maximum score s and if it exceeds the pre-defined threshold, concludes that T is cloned from v_{i-1}.

The procedure is illustrated in Fig. 6. Note that code continua is collected to align pre-defined boundaries such as functions, methods, and classes to form subtrees.

3.4 Constructing and Comparing Change Histories

A change history for a code fragment is a sequence of change types obtained by accumulating change types whose locations are contained in the location of the fragment. We compare change histories in order to detect inconsistent changes. We regard a pair of code fragments as inconsistently modified if the similarity score between non-empty change histories is less than a specified threshold. The similarity score between two change histories is defined based on the Levenshtein distance by $2m/t$, where m is the number of matches and t is the total number of change types in both histories.

4 Experiments

In this section, we present the results of code homology analysis on several open source software to demonstrate the capability of the method. Examples include BIND9 daemon (for xenologue detection), a couple of Java projects, jEdit[4] and Ant[5] (for paralogue detection and tracking), and Linux kernel drivers (for change history construction). The experiments are conducted in the following manner:

1. Fill the local software repository with revisions of target software.
2. Xenologues are calculated by double-differencing (BIND9 only) and paralogues are calculated by backward code matching. Code clones detected in the first version are also treated as paralogues.
3. Paralogues are checked for inconsistent changes descending from the origin of the paralogues, which often account for latent bugs. Checking is done at the level of change type sequences described in Sect. 2 (except BIND9).
4. Discovered homologues and change histories go through human inspection for validation.

We used a PC with a pair of quad-core Intel Xeon CPU (3.0GHz) with 16GB RAM running under Linux kernel 2.6.24 for these experiments.

[4] `http://www.jedit.org/`
[5] `http://ant.apache.org/`

Table 1. Sample Projects

	lang.	# of ver.	versions	# of src files	kSLOC
BIND9	C	26	9.0.0 - 9.5.0-P2	767 - 1,186	153 - 260
jEdit	Java	56	3.2.2 - 4.3.0pre17	279 - 532	55 - 108
Ant	Java	41	1.1 - 1.7.1	87 - 1,220	9 - 125
Linux	C	32	2.6.0 - 2.6.31	12,424 - 23,752	3,626 - 7,337
(drivers)				(3,285 - 8,116)	(1,778 - 4,106)

Programming languages in which analyzed source code is written, the numbers of analyzed versions, and versions, files, and kSLOC (comments and blank lines are ignored) of the initial and the latest versions of the target systems are shown in Table 1. For Linux, data for `drivers` subsystem are shown in parentheses.

4.1 BIND9

The main purpose of this case is to ascertain that our analysis can actually detect xenologues corresponding to merged or patched code fragments. ISC BIND (Berkeley Internet Name Domain) is an implementation of the Domain Name System (DNS) protocols. As of October, 2008, three release branches of BIND9, namely 9.2.x, 9.3.x, and 9.4.x are actively maintained. We selected 26 versions to be analyzed. Our system detected 215 orthologues. We found that about 30% of them are degenerating, that is, decreasing in size. This indicates that 30% of commonly inherited code was modified in one or more branches and 70% of it is stable for generations. The system also detected 8,948 xenologues most of which (98.27%) are relatively small in size (< 64 nodes).

In the case of BIND9, it is likely that xenologues are introduced by patch applications such as security patches since multiple releases have been maintained in parallel. For example, a modification "Query id generation was cryptographically weak." (RT#16915) is commonly included in `CHANGES` files contained in releases 9.2.8-P1, 9.3.4-P1, and 9.4.1-P1. As we expected, several xenologues in `dispatch.c`, out of 34 xenologues among 9.2.8-P1, 9.3.4-P1, and 9.4.1-P1, appear to be strongly related to that modification.

4.2 jEdit and Ant

We collected 56 versions of jEdit from release 3.2.2 to 4.3pre17, and 41 versions of Ant from release 1.1 to 1.7.1. First, we compare the paralogue (code clone) tracking ability of our method with that of Duala-Ekoko and Robillard [6]. They implemented a system called CloneTracker which is capable of tracking code clones detected by clone detection tools. It identifies clone regions at the granularity of code blocks using heuristics based on the structural properties, lexical layout, and similarities of the clone region. They provided case studies of jEdit and Ant. A clone detection tool called SimScan was used to detect code clones in the initial versions. Then, they selected five clone groups, which were tracked across the subsequent versions. They also manually inspected changes made in the tracked clone groups.

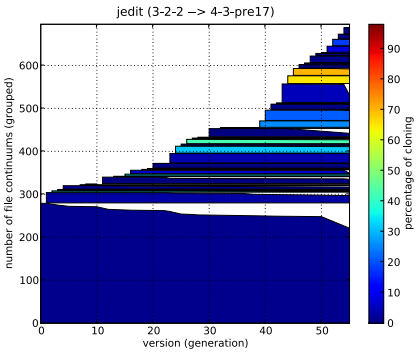

Fig. 7. File Continua of jEdit

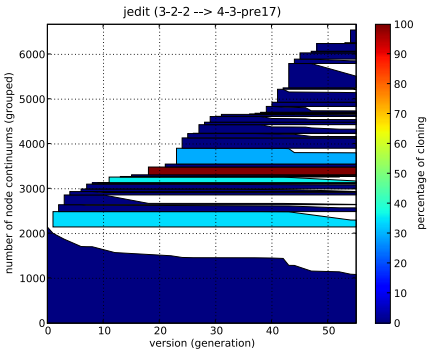

Fig. 8. Node Continua of jEdit (MiscUtilities.java)

We have tracked the clone groups detected by SimScan including the five clone groups above and reconstructed change histories for them by our method. We ran SimScan with the same settings as that of Duala-Ekoko and Robillard's experiment, namely volume=medium, similarity=fairly similar, and speed=fast.

Our tracking results were consistent with their results except for a clone region that they could not track. Our system was able to track the clone region up to the most recent version. All reported changes collected by their manual inspection were automatically reconstructed by our change history construction method briefly described in Sect. 3.4.

We also pairwise compared reconstructed change histories of code fragments in tracked clone groups. We set the similarity threshold to 0.5. For lack of space, we only show the results of Ant. 537 out of 1,078 initial clone pairs detected by SimScan were inconsistently modified without disappearing before the latest version, while 340 pairs disappeared before the latest version. Our system also detected 1,247 additional clone pairs after the initial version. Among them, 272 pairs were inconsistently modified (excluding 369 disappeared clone pairs). Note that detected inconsistent changes do not immediately account for bugs. There is a possibility that clones are intentionally modified differently [12]. It took our system 60 and 70 minutes to complete the whole analysis of jEdit and Ant, respectively.

Our system can also visualize cloning activities overlaid on continua. Figures 7 and 8 show file and code (MiscUtilities.java) continua for jEdit. Each horizontal line represents a file (node) continuum and each polygon a group of file (node) continua that begin at the same version. In each polygon, continua are sorted by terminating versions and colors represent the percentage of continua generated by cloning activities in the corresponding group. We can see that a couple of versions introduced numerous clones.

4.3 Linux Device Drivers

In this case study, we manually investigated an inconsistent change detected by our system and made a certain contribution to an open source community. We

collected and analyzed 32 versions of Linux 2.6 kernel source code from 2.6.0 to 2.6.31. We first detected and tracked the paralogues in the whole kernel source code. Then we constructed fine-grained change histories for the paralogues in the `drivers` subsystems and manually inspected the histories to detect inconsistent changes. The `drivers` subsystem is large enough as they can occupy more than 70% of modern operating systems in volume and account for the vast majority of bugs as reported by Chou and others [13].

In order to detect initial clones in the initial version 2.6.0, we used CCFinder with the default setting. CCFinder detected 2,851 clone pairs. We set the similarity threshold between change histories to 0.9999, and then ran the system. It took three days to detect and to track the whole paralogues and two days to construct change histories for paralogues in `drivers` subsystems. Our system detected 814 additional clone pairs (cloning activities) after version 2.6.0 and 1,441 and 385 inconsistently modified pairs out of the initial and the additional clone pairs, respectively. We manually inspected inconsistent changes detected by our system. In this experiment, only change history pairs with similarity score more than or equal to 0.9 are inspected. It should be noted that overlooking changes such as **argument deleted** and **parameter type changed** leads to compiler errors, and hence immediate regression faults. Overlooking protocol changes such as **function call inserted**, however, often causes latent errors difficult to detect. Thus we focused on insertion of statements.

We were able to find approximately 10 inconsistent changes involving insertions of function calls to `lock_kernel` that remained in the latest version 2.6.31. An inconsistent change involving `lock_kernel` detected by the system is shown in Fig. 9, where a clone pair in `synclinkmp.c` and `synclink_cs.c` was inconsistently modified. Their change type sequences were mostly the same, which means that they almost consistently co-evolved, but differ in only one change. In this experiment, we observed a number of clone pairs that evolved almost consistently. Among them, a pair of change histories that shares 178 change types in common with a similarity score of 0.98 was discovered.

By further inspection, we found that the inconsistency relates to similar inconsistencies observed in Linux serial drivers violates a development policy concerning kernel locking known as BKL (big kernel lock) pushdown. The BKL was introduced to make the kernel work on multi-processor systems. The role of the BKL, however, has diminished over years since fine-grained locking has been implemented throughout the kernel for better throughput. Although some attempts to entirely remove the BKL have been made, progress in that direction has been slow in recent years.

It was not long before the BKL accounted for a performance regression. At last, some of the developers decided to go a step further in versions 2.6.26 and 2.6.27. They began with serial drivers. In order to remove upper-level acquisition of the BKL in the control flows, they attempted to push the acquisition down to the device specific code level, where they expected BKL removal to be achieved. Indeed, numerous `lock_kernel` calls were (almost blindly for safety) inserted into serial driver code including `synclinkmp.c` at version 2.6.26, and then the upper-

level call (in `fs/char_dev.c`) was removed at version 2.6.27. However, during the pushdown, `synclink_cs.c` was left unchanged, which led to an inconsistent change detected by our system.

Although the inadvertency itself does not cause errors, we could promote the BKL pushdown policy. In response to our report on the inconsistent change, the author of `synclinkmp.c` pointed out that the `lock_kernel` calls in the driver can be safely removed. We submitted a patch removing the BKL related calls from `synclinkmp.c` and its variations. It was acknowledged by the author of the driver.

```
*** detected pair [1482] (historical similarity: 0.972973) ***
ORIGIN: "linux-2.6.0/drivers/char/synclinkmp.c":1129-1172 --
        "linux-2.6.0/drivers/char/pcmcia/synclink_cs.c":2616-2659
LATEST: "linux-2.6.31/drivers/char/synclinkmp.c":1052-1096 --
        "linux-2.6.31/drivers/char/pcmcia/synclink_cs.c":2439-2481
total significance of history1: 32 (max=4)
total significance of history2: 30 (max=4)

--- 1 changes found only in "linux-2.6.0/drivers/char/synclinkmp.c":1129-1172:

@"linux-2.6.25/drivers/char/synclinkmp.c":1109-1151
@"linux-2.6.26/drivers/char/synclinkmp.c":1111-1155
[statement inserted] (significance=2)
[Statement.Expression(Expression.Call()){lock_kernel}[(<call>(lock_kernel,<args>))]]
@Definition(wait_until_sent)(1108L,0C-1167L,0C(30773-32316))
@Definition(wait_until_sent)(1124L,1C-14C(31088-31101))
```

Fig. 9. An Example of Inconsistent Change

5 Related Work

There is a large body of studies on code clones and related detection tools [4]. Code clone detection and source code differencing are two complementary techniques for analyzing relationship among software products. Although code clone detection tools may well be able to discover homologous code, they may not be suitable for precisely tracking life cycles of such code in the long evolution of the software as it requires consistently identifying changed and unchanged parts. It usually takes awkward steps of taking difference by way of clone detection. Since code clone detection methods can analyze clones across different and unrelated software products, it would be an interesting topic to combine these two methods. One idea is to use clone detection tools first to reduce the size and the scope of the problem, and then to apply tree differencing tools for detailed analysis.

Kim and others [7] proposed a method for inferring high-level description of structural changes based on the first-order logic in order to determine method level matches between versions of Java programs. While their method is specialized for Java programs, their idea of aggregating low-level description of changes seems applicable to our system. Kim and Notkin apply clone detection techniques to understand evolution of code clones [5] in Java software. They rely on location overlapping relationship to track code snippets across multiple versions of a program. While their analysis is simple and fast, it may not be able to extract how such code snippets changes over time from the source code. Duala-Ekoko and Robillard [6] proposes a code tracking method tailored for Java. Although

the method is driven by heuristics and suitable for interactive use, the lack of precision in syntactic analysis may limit the ability of the tool. Godfrey and Zou [14] developed a set of techniques for detecting merging/splitting of functions and files in software systems. They presented a set of merge/split patterns and employed call relationships to aid in detecting their occurrence, which are also useful for our analysis. Aversano and others [15] proposed a method of investigating how clones are maintained over time. In order to derive evolution patterns of clones, they rely on fast but coarse line-by-line differencing to track clones.

In Sect. 3, we implicitly assumed that the genealogies of target software systems are given. However, without any development history or any explicit record of tagging or version copying operations, it may be difficult to determine the origin of branching, notably the version from which a development branch is duplicated. In such situations, we can reconstruct the genealogies by utilizing tree differencing and tools for phylogeny [9].

6 Conclusion

We have proposed an automated method for analyzing code homology in genealogy of evolving software based on fine-grained tree differencing. Homologues can be introduced through various activities: branching/forking in software projects (orthologues), code exchange between neighboring branches such as code import/merging and common bug-fix patches (xenologues), and code duplication within branches (paralogues or code clones). As the development proceeds, homologues can incur additional maintenance efforts. We have developed a method for detecting and tracking such distinctive pieces of code by exploiting fine-grained tree differencing. Detecting and tracking homologues along evolution branches enable us to reconstruct and to compare change histories of homologues, which leads us to detect inconsistent changes. Results of experiments conducted on several large-scale software including BIND9 DNS servers, a couple of Java software jEdit and Ant, and Linux device drivers have been reported to show the capability of the method. Having scalable and precise tree differencing engines helped us to analyze a large-scale software project such as the Linux kernel consisting of several millions of SLOC.

Future work includes the following:

1. to improve processing speed by further exploiting parallelism and by eliminating redundant computation,
2. to build a database for efficiently storing and retrieving various (intermediate) results computed by the system, together with comprehensive graphical user interface, and
3. to apply our analysis to:
   (a) change pattern mining and future modification prediction,
   (b) language-aware merging, and
   (c) the concrete problem of generating generic patches [16] that cover wide range of Linux device drivers.

References

1. Fitch, W.: Homology a personal view on some of the problems. Trends in Genetics **16**(5) (May 2000) 227–231
2. Fluri, B., Gall, H.C.: Classifying change types for qualifying change couplings. In: ICPC '06: Proceedings of the 14th IEEE International Conference on Program Comprehension. (2006) 35–45
3. Kim, M., Notkin, D.: Program element matching for multi-version program analyses. In: MSR '06: Proceedings of the 2006 international workshop on Mining software repositories. (2006) 58–64
4. Roy, C.K., Cordy, J.R., Koschke, R.: Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. Sci. Comput. Program. **74**(7) (2009) 470–495
5. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: An empirical study of code clone genealogies. In: ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering. (2005) 187–196
6. Duala-Ekoko, E., Robillard, M.P.: Tracking code clones in evolving software. In: ICSE '07: Proceedings of the 29th International Conference on Software Engineering. (2007) 158–167
7. Kim, M., Notkin, D., Grossman, D.: Automatic inference of structural changes for matching across program versions. In: ICSE '07: Proceedings of the 29th international conference on Software Engineering. (2007) 333–343
8. Reiss, S.P.: Tracking source locations. In: ICSE '08: Proceedings of the 30th international conference on Software engineering. (2008) 11–20
9. Hashimoto, M., Mori, A.: Diff/TS: A tool for fine-grained structural change analysis. In: WCRE '08: Proceedings of the 15th Working Conference on Reverse Engineering. (2008) 279–288
10. Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering **28**(7) (2002) 654–670
11. Myers, E.W.: An O(ND) difference algorithm and its variations. Algorithmica **1**(2) (1986) 251–266
12. Kapser, C., Godfrey, M.W.: "cloning considered harmful" considered harmful. In: WCRE '06: Proceedings of the 13th Working Conference on Reverse Engineering. (2006) 19–28
13. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles. (2001) 73–88
14. Godfrey, M.W., Zou, L.: Using origin analysis to detect merging and splitting of source code entities. IEEE Transactions on Software Engineering **31**(2) (2005) 166–181
15. Aversano, L., Cerulo, L., Di Penta, M.: How clones are maintained: An empirical study. In: CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. (2007) 81–90
16. Padioleau, Y., Lawall, J.L., Muller, G.: Understanding collateral evolution in linux device drivers. In: EuroSys '06: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. (2006) 59–71