Extracting a Unified Directory Tree to Compare Similar Software Products

Yusuke Sakaguchi, Takashi Ishio, Tetsuya Kanda, Katsuro Inoue
Graduate School of Information Science and Technology, Osaka University, Japan
{s-yusuke, ishio, t-kanda, inoue}@ist.osaka-u.ac.jp

***Abstract*—Source code is often reused in software development. Although developers can avoid re-implementing features in existing products, doing so may result in a large number of similar software products. To understand the commonalities and variabilities of similar products, comparing their source code is critical. However, a product may change its own directory structure, even if the products share the same source code with other products. Hence, comparing source code among products in a systematic manner is difficult.**

In this paper, we propose a technique to extract and visualize a unified directory tree to compare the source code of similar products. This tree includes all directories of given products and merges corresponding directories into a single node. Since a node in a tree corresponds to multiple directories in products, developers can easily compare the contents of products. In our study, we implemented the visualization as a GUI tool. In addition, we conducted a case study using four Android products to demonstrate the tool's ability to assist developers in accessing the source code of multiple products.

I. Introduction

In software development, source code reuse is a common practice to reduce the cost of development. Developers often create a new product by copying and modifying an existing one or importing libraries [1], [2]. Furthermore, they often reuse the developed product to create yet another new product. This method of code reuse is called a "clone-and-own" approach.

Since the clone-and-own approach is common, the industry already maintains a large number of derived software products. Although Software Product Line Engineering is promising for managing such software products, the construction of a software product line from existing products requires that developers understand the commonalities and variabilities of them [3].

Source code comparison is an important step in understanding the features of products [4]. In general, since a directory of a product (e.g. a Java package) is assumed to represent a feature, comparing multiple products in units of directories is efficient [5], [6]. Duszynski et al. [7] proposed comparing corresponding directories among similar products. They visualized the numbers of the lines of both common and product-specific source code. However, developers must know the correspondence of directories before comparison; if directories have been moved or renamed among the products, determining correspondences may be difficult. Some existing techniques [8], [9] can extract corresponding directories between two similar products, but they cannot analyze more than two products at any one time.

In this paper, we propose a technique to extract and visualize automatically a unified directory tree representing corresponding directories among products. A unified directory tree includes all directories including source code of given products. Since a directory may be moved or renamed, we regard directories that include similar files as corresponding. We merge similar directories into a single node in a unified tree so that developers can easily compare their contents.

In our study, we implemented a tool that automatically extracts a unified directory tree from the given source code of products. The tool translates directory trees into a single connected graph and then extracts a spanning tree on the graph. The resulting tree is visualized using a tree view widget. Similar to existing file managers, selecting a directory node in the tree then our tool shows the details of the node. Since a node represents multiple directories of products, our tool provides a table of files summarizing the contents of directories. Developers can examine file similarities on the table, and compare pairs of files using an external source code differencing tool if necessary.

We conducted a case study with four open source archives of the Android operating system provided by two companies. The viewer enables us to compare multiple products in units of directories.

The reminder of the paper is organized as follows. Section II describes the background of our study. Section III illustrates the manner in which we visualize source code of similar products in a single unified directory tree. Section IV presents the case study. We conclude the paper in Section V.

II. Background

Clone-and-own reuse is a popular method of enhancing an existing product. Dubinsky et al. [10] reported that industrial developers tend to copy source code from existing products. Hemel et al. [11] analyzed industrial variants of Linux kernels using a code clone detection technique.

Duszynski et al. [7] proposed a visualization technique to distinguish common source code in products from product-specific code. Since their technique assumes input products have the same directory structure, developers must manually investigate directory structures to identify moved or renamed directories.

978-1-4673-7525-2/15/$31.00 © 2015 IEEE

Yoshimura et al. [8] extracted corresponding directories between two products by using code clone detection in order to merge two similar products. However, this approach has not been designed for comparing more than two products. Holten et al. [9] proposed a visualization that compares a pair of directory trees. It visualizes the manner in which directories in two products correspond to one another. Although it provides a global overview of structural differences, the technique does not aim to compare multiple products. A file-level rename tracking has been proposed by Lavoie et al. [12]. The technique identifies the most similar source file pairs between versions. We use file similarity at a directory level.

Beck et al. [13] proposed an asymmetric visualization to compare a pair of directory trees. It visualizes a primary directory tree in a large plot and embeds a secondary directory tree in the nodes of the primary tree. It enables developers to identify similarities between two directory trees. Our tool does not directly visualize similarities. Instead, our visualization facilitates file comparison among directory trees.

Lin et al. [14] proposed a source code comparison tool that is specialized for multiple instances of code clones. Although the tool visualizes commonalities among code clones on a code fragment level, it is not a tool used to investigate directory-level similarities.

III. Visualization

We visualize the source code of similar products in a single unified directory tree. We use as input a set of directories $R = \{r_1, r_2, \cdots, r_{|R|}\}$, where r_i is the root of a directory tree containing the source code of a product. We regard a directory tree as a directed graph. We obtain a unified directory tree from given directory trees through the following three steps.

1) Create a node for each set of directories that contain similar source files,
2) Connect nodes with weighted arcs representing subdirectory relationships in products, and
3) Extract a directed spanning tree from the resultant graph.

We implement a viewer for a unified directory tree, which provides several features to investigate and compare source files in the directory tree.

A. Directory Tree Extraction

To enable developers to compare similar source files across software products, we create a node for each set of directories that include similar files. We introduce a content similarity metric $sim(d_1, d_2)$ for two directories d_1 and d_2 in different products. The single node represents d_1 and d_2 if $sim(d_1, d_2)$ is equal to or greater than a predetermined threshold th. Although $sim(d_1, d_2) \geq th$ and $sim(d_2, d_3) \geq th$ does not always imply that $sim(d_1, d_3) \geq th$, we assign the same node to represent all of them so that developers can analyze the differences among the directories. We use $D(n)$ to represent a set of directories represented by a node n. The content similarity $sim(d_1, d_2)$ between two directories is

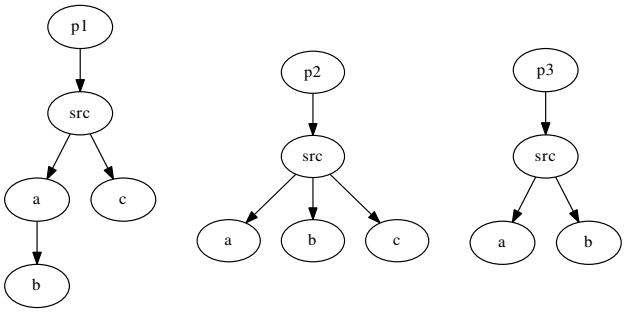

Fig. 1. Example directory trees of products

defined according to the following, using the Jaccard similarity coefficient:

$$sim(d_1, d_2) = \frac{|L(d_1) \cap L(d_2)|}{|L(d_1) \cup L(d_2)|}$$

where $L(d)$ is a set of lines extracted from all non-binary files directly contained in a directory d. Since developers may change the source code layout and line separator characters, we remove white space characters from lines.

For example, consider three products `p1`, `p2`, and `p3`, each consisting of three components `a`, `b`, and `c`. The components are stored in different directory trees as shown in Fig. 1. In this case, we introduce three nodes `na`, `nb`, and `nc` to represent the component directories as follows.

$$\begin{aligned} D(\texttt{na}) &= \{\texttt{p1/src/a}, \texttt{p2/src/a}, \texttt{p3/src/a}\} \\ D(\texttt{nb}) &= \{\texttt{p1/src/a/b}, \texttt{p2/src/b}, \texttt{p3/src/b}\} \\ D(\texttt{nc}) &= \{\texttt{p1/src/c}, \texttt{p2/src/c}\} \end{aligned}$$

The source code of a software product is usually organized into directories without files. For example, the `src` directory in a Java program contains only subdirectories that represent Java package names. We assign a single node to represent such directories if their subdirectories are represented by the same node. In other words, if two directories d_1 and d_2 are represented by a single node n ($d_1, d_2 \in D(n)$), their parent directories d_{p1} and d_{p2} are represented by a common parent node. In the case of the example directory trees, we merge `src` directories in the products into a single node, even if they possess no source files.

To ensure that the resultant graph is a connected graph, we introduce the root node r that represents all root directories of products, irrespective of product similarity. The root node represents the root of a unified directory tree to be extracted.

The created nodes are connected using weighted arcs. Given a pair of nodes n_1 and n_2, we compute their weight according to the following:

$$w(n_1, n_2) = |\{(d_1, d_2) : \quad d_1 \in D(n_1) \wedge d_2 \in D(n_2) \\ \wedge subdir(d_1, d_2)\}|$$

where a predicate $subdir(d_1, d_2)$ means that d_1 is a parent directory of d_2. If $w(n_1, n_2) > 0$, an arc connects two nodes with the weight, and if $w(n_1, n_2) = 0$, no arc exists because

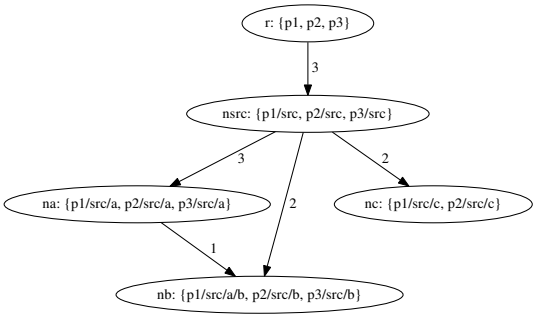

Fig. 2. A unified directory graph for the example directory trees

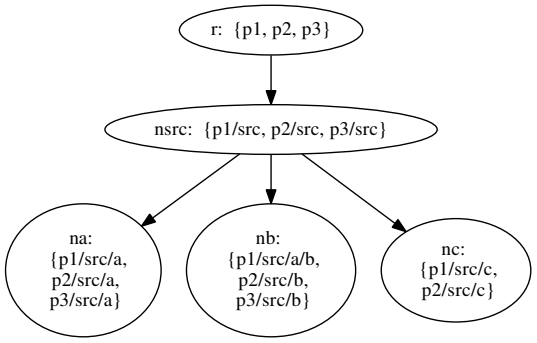

Fig. 3. A unified directory tree for the example directory trees

no parent-subdirectory relationships exist among nodes n_1 and n_2. Because a node representing a directory has arcs to nodes representing its subdirectories, all nodes are reachable from the root node r. As a result of this step, the example directory trees are translated into a directed graph as shown in Fig. 2. Since directory `b` is a subdirectory of `a` in `p1` but a subdirectory of `src` in `p2` and `p3`, the corresponding node `nb` has two incoming arcs that indicate the relationships.

We extract a directed spanning tree from the resultant graph. The extraction process is a simple greedy loop according to the following.

1) Initialize a spanning tree $V = \{r\}, E = \phi$, where V represents a set of vertices and E represents a set of selected arcs.
2) Select the arc (t, n) having the maximum weight $w(t, n)$ among $t \in V, n \notin V$. If the weights are identical, select that which is closest to r.
3) Update $V \leftarrow V \cup \{n\}$, $E \leftarrow E \cup \{(t, n)\}$
4) Repeat Steps 2 and 3 until V includes all nodes. Finally, E includes arcs of the spanning tree.

Given the graph shown in Fig. 2, the algorithm produces a tree as shown in Fig. 3. The tree includes arcs from `nsrc` to `na`, `nb`, and `nc` and ignores an edge from `na` to `nb`. Thus, the resultant unified tree is the same as the directory tree of `p2`. The directory `p1/src/a/b` is virtually regarded as a subdirectory of `src`, because two other products have similar directories at that location.

The most time-consuming aspect of our algorithm is calculating sim, and its complexity is $\mathcal{O}(|d|^2)$ where $|d|$ is the number of input directories. In general, $|d|$ is much smaller than the number of input file $|f|$. Therefore, our algorithm has less complexity than when performing file-to-file comparisons.

B. Aligning the tree with a specific product

Our algorithm as previously described *evenly* unifies directory trees. In other words, a unified directory tree may be different from all input products. If developers have expertise in a specific product, aligning directories of other products to a specific well-known product is reasonable. To support such developers, we introduce an extended weight for targeting a specific product. Given a target product P and an arc between nodes n_1 and n_2, we compute the weight according to the following:

$$w(n_1, n_2, P) = \begin{cases} \infty & \textbf{if } \exists d_1 \in D(n_1), d_2 \in D(n_2) : \\ & d_1 \in P \wedge d_2 \in P \wedge \\ & subdir(d_1, d_2) \\ w(n_1, n_2) & \textbf{otherwise}. \end{cases}$$

With this weight function, arcs representing a subdirectory relationship in the target product is always selected in the spanning tree extraction process. In addition, arcs connecting directories unique to other products are selected.

C. Directory Viewer

We implemented a visualization tool for a unified directory tree. The viewer enables developers to explore a unified directory tree and examine differences of directories in a node. Fig. 4 is a screenshot of the tool. The tool contains product selection buttons on the top to align a unified tree to a particular product. The remaining area consist of three views: a tree, a file list, and a file matrix.

1) Tree View: The tree view shows the unified directory tree. Our algorithm unifies the inputs into a single tree. Thus, this view shows only a single tree and users are free to compare multiple large trees. A node in the view represents a node in the tree. The name of each node is based on directory names in the node. If a node corresponds to several directories, the tree view uses the most popular name among them.

A node also reveals the number of directories x and number of variants y included as (y in x). The number of variants indicates the number of directories that include different file contents from one another. For example, (2 in 3) means this node has three directories but one of them has different content from the other directories, whereas (1 in 3) means this node has three directories with the same file content. The number of variants is based on file contents directly contained in the directories represented by the node. Thus, the (1 in 3) node may have a subdirectory with variants. If a node contains multiple variants, the node is colored in blue.

In addition, if a node contains a "moved" directory, that is, if a directory in the node has a different name or a different file path than the tree node, then the node is marked by "`*`".

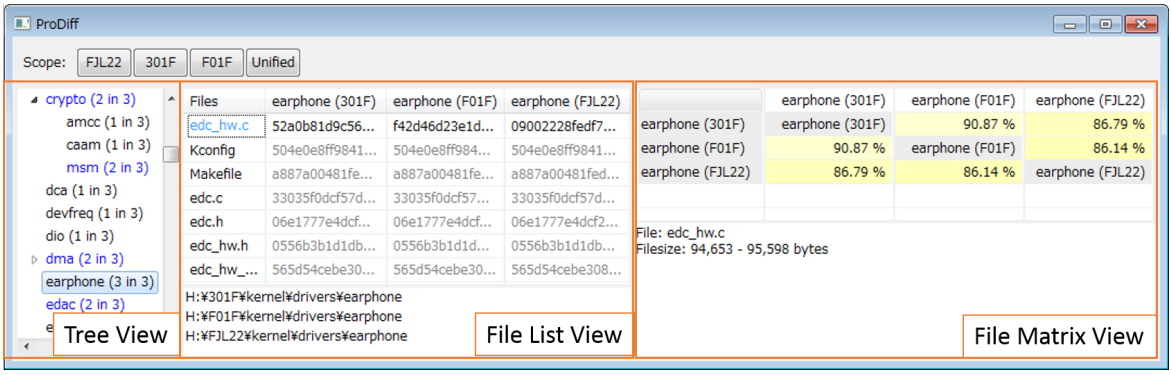

Fig. 4. An overview of the Directory Viewer

2) File List View: A file list view shows a table of files contained in a selected node. The first column denotes filenames. The second and subsequent columns correspond to directories contained in the node. Each cell shows the hash value of a file in a directory. A gray value indicates that the files of the same name have the same content. A black value indicates that some differences exist among the files.

The list can be sorted by either filename or colors of hash values so that developers can focus on files with multiple variants.

The full paths of the directories shown are listed at the bottom of the table. Clicking a path opens the directory using a default file manager (e.g., Explorer in Microsoft Windows).

3) File Matrix View: A file matrix view shows the similarity of files having the same file name selected in a file list view. Rows and columns correspond to directories (i.e., like the columns of the file list view). A cell (i, j) shows the $sim(f_i, f_j)$ between files in the selected directories. The cell is colored using color scale white ($sim = 1$) to yellow ($sim = 0.5$) to red ($sim = 0$). Developers can quickly see the amount of differences in files among the input products.

After one of the cells is selected, an external diff tool (e.g., WinMerge [1]) is executed in order to compare two files.

IV. Case Study

We demonstrate the proposed method using four variants of Android's source code released in the fourth quarter of 2013, including three products designed by Fujitsu and one by Sony. They all employ a Qualcomm MSM8974 CPU, Android 4.2 series OS. The list of inputs is shown in Table I.

Because these products use similar software versions, we expect that most parts of the source code are similar whereas some product specific parts are different. In addition, we expect that some directories and files in SO-01F are different from other Fujitsu products. In this case study, we show how to explore the code of these products using our tool.

We specified four directories as input for our tool. We ran the tool on a machine equipped with two Intel Xeon E5507 processors (2.27 GHz, 4 cores) and 24 GB RAM. Approximately 42 minutes were required to read files, compute similarity among directories, and extract a unified directory tree for the products. The tool used a predetermined similarity threshold $th = 0.8$.

The resultant unified directory tree comprises 9,037 nodes, and 245 nodes contain moved or renamed directories. In addition, 673 nodes contain different contents from other products.

Looking into the `/kernel` directory (Fig. 5), we observed that several files have different contents. For example, files `AndroidKernel.mk` and `Makefile` in SO-01F are different from other Fujitsu products. `Makefile` in Fujitsu products are only slightly customized and similarity to SO-01F is 95.01%. By contrast, the similarity of `AndroidKernel.mk` is 40.52% as shown in the file matrix view in Fig. 5. We found that developer-specific options for the kernel build is mainly written in this file. The file `MAINTAINERS` of F-01F is different from others. Although they all use the same kernel version, some files in F-01F lack some non-ASCII characters (e.g., umlaut). It seems that the characters are accidentally replaced by a developmental environment.

Because the `/vendor` directory in Fujitsu products contain completely different files from `/vendor` in SO-01F, the directories are shown as independent nodes in the unified tree. Similarly, the content of the `/external/chronium` directory in Fujitsu products is distinguished from the SO-01F version. For example, a file `Android.mk` is removed from the Fujitsu code. By contrast, common subdirectories of `/external/chronium` are nearly similar among the four products. Therefore, common subdirectories are merged into nodes in Fujitsu's `/external/chronium` node. Another `/external/chronium` node for SO-01F retains only subdirectories unique to the version.

The node `/external/llvm` contains six directories from Fujitsu products, three from `/external/llvm`, and three from `/external/llvm/projects/sample`. The `sample` directories are accidentally merged because the directories share similar files such as `configure` files.

In the case study, we explored the unified directory structure as if it is a single product, and we easily focused on the

[1] http://winmerge.org

TABLE I
LIST OF INPUT PRODUCTS. ALL OF THEM ARE VARIANTS OF ANDROID 4.2.

Product	Vendor	Mobile Network Operator	Release	#Dirs	#Files	#Lines
FJL22	Fujitsu	au	Nov. 2013	7,683	107,945	26,178,588
301F	Fujitsu	SoftBank	Dec. 2013	7,708	108,334	25,629,778
F-01F	Fujitsu	NTT DOCOMO	Oct. 2013	7,582	105,397	25,740,695
SO-01F	Sony	NTT DOCOMO	Oct. 2013	5,840	90,736	22,225,611

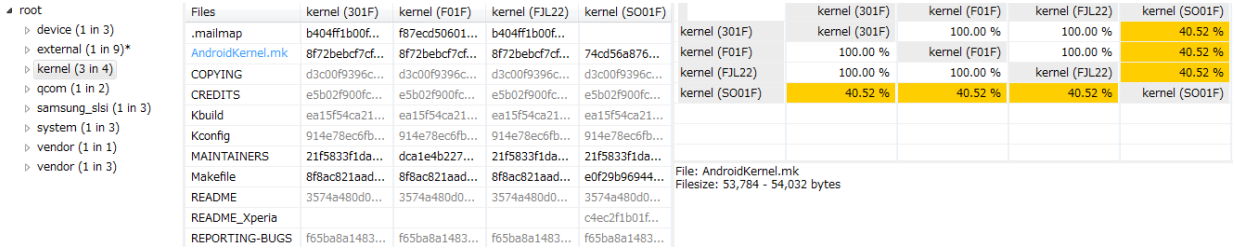

Fig. 5. Output for `/kernel` directory and similarity matrix for `AndroidKernel.mk`.

directories having differences. Although four Android source code archives included 28,813 total directories, the unified tree covered the entire structure in 9,037 nodes. Because directories are represented by independent nodes if they have different contents, we did not analyze such directories. In addition, since subdirectories are merged independently of their parent directories, we focused on the comparison of similar files in such subdirectories.

V. CONCLUSION

Source code comparison is crucial to understanding commonalities and variabilities among products. In this study, we presented a technique to extract a unified directory tree that includes all directories of multiple software products. We implemented a viewer of the tree and conducted a case study using four Android products. The tool enables developers to identify different files easily among products and compare them.

For a future work, we want to evaluate the quality of an extracted unified tree. Furthermore, we want to conduct a controlled experiment on the effectiveness of the tool for source code comparison tasks.

ACKNOWLEDGMENT

This work is supported by Japan Society for the Promotion of Science, Grant-in-Aid for Scientific Research (S) "Collecting, Analyzing, and Evaluating Software Assets for Effective Reuse"(No.25220003) and Osaka University Program for Promoting International Joint Research, "Software License Evolution Analysis."

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] N. Kawamitsu, T. Ishio, T. Kanda, R. G. Kula, C. De Roover, and K. Inoue, "Identifying source code reuse across repositories using lcs-based source code similarity," in *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 305–314.

[3] J. Bosch, "Maturity and evolution in software product lines: Approaches, artefacts and organization," in *Proceedings of the 2nd Conference Software Product Line Conference*, 2002, pp. 257–271.

[4] C. W. Krueger, "Easing the transition to software mass customization," in *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, 2002, pp. 282–293.

[5] M. de Jonge, "Build-level components," *IEEE Transactions on Software Engineering*, vol. 31, no. 7, pp. 588–600, 2005.

[6] ——, "Multi-level component composition," in *Proceedings of the 2nd Groningen Workshop Software Variability Modeling*, no. 2004-7, 2004.

[7] 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.

[8] K. Yoshimura, D. Ganesan, and D. Muthig, "Assessing merge potential of existing engine control systems into a product line," in *Proceedings of the International Workshop on Software Engineering for Automotive Systems*, 2006, pp. 61–67.

[9] D. Holten and J. J. van Wijk, "Visual comparison of hierarchically organized data," in *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*, 2008, pp. 759–766.

[10] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An exploratory study of cloning in industrial software product lines," in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 25–34.

[11] 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.

[12] T. Lavoie, F. Khomh, E. Merlo, and Y. Zou, "Inferring repository file structure modifications using nearest-neighbor clone detection," in *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 325–334.

[13] F. Beck, F.-J. Wiszniewsky, M. Burch, S. Diehl, and D. Weiskopf, "Asymmetric visual hierarchy comparison with nested icicle plots," in *Proceedings of the 1st International Workshop on Graph Visualization in Practice*, 2014.

[14] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao, "Detecting differences across multiple instances of code clones," in *Proceedings of the 36th International Conference on Software Engeneering*, 2014, pp. 164–174.