類似ソフトウェア比較のための統一されたディレクトリ構造の可視化ツール

坂口 雄亮　石尾 隆　神田 哲也　井上 克郎

ソフトウェア開発において，ソースコードの再利用が行われた結果，類似したソフトウェアが数多く生まれることがある．そのような類似ソフトウェアの共通性や独自性を理解するにはソースコードの比較が重要となるが，類似ソフトウェアといってもディレクトリ構造は異なる場合があり，対応したディレクトリを見つけることは困難である．本研究では，そのような類似ソフトウェアのソースコードからから，それらを統合した単一のディレクトリ構造を抽出し，可視化するツールを実装する．抽出されるディレクトリ構造は，与えられたソフトウェアのディレクトリを全て含み，ソフトウェア間で対応関係にあると推定されたディレクトリを 1 つのディレクトリとして表現する．そして，対応関係にあるディレクトリでは，それらのファイル内容の比較を容易に行うことができる．試作したツールのケーススタディとして，4 つの Android 製品に対して可視化を適用した例を示す．

Source code is often reused in software development. Such reuse activity may result in a large number of similar software products. To understand commonalities and variabilities among such similar products, comparison of their source code is important. However, each product may change its own directory structure; it is hard to identify corresponding directories among products. In this paper, we propose a technique to extract and visualize a unified directory tree. The extracted directory tree includes all the 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 files. As a case study of our protoype, we conducted a case study with four Android products.

1　はじめに

ソフトウェア開発において，開発コストを削減するためにソースコードの再利用は重要である．開発者は，しばしば，既存ソフトウェアのソースコードをコピーや編集を行ったり，ライブラリを取り込むことで新しいソフトウェアを開発する [8] [12]．さらに，新たなソフトウェアを開発するために，既存ソフトウェアの必要な部分のみに機能の変更や追加を行う．この開発方法は，“派生開発” と呼ばれる．

派生開発が進むことで，派生ソフトウェアプロダクトが数多く生まれる．このようなソフトウェアを管理する，ソフトウェアプロダクトラインエンジニアリングという概念があるが，既存のプロダクトにこの概念を適応するためには，機能の共通部分と固有部分を理解する必要がある [1]．

ソフトウェアの機能を理解するにあたって，ソースコードの比較は重要である [9]．一般的に，Java のパッケージのように，ソフトウェアのディレクトリは 1 つの機能を表すため，複数のソフトウェアをディレクトリ単位で比較することは効果的である [2] [3]．Duszynski ら [5] は，類似したソフトウェア間の対応しているディレクトリの比較を行う手法として，ソースコードの共通部分の行数と固有部分の行数を可視化する手法を提案した．しかし，この手法を適用するには，比較を行う前に対応したディレクトリを知ることが必要となるが，類似ソフトウェア間でディレクトリの移動やリネームが行われた場合，対応を取るこ

A Tool Visualizing a Unified Directory Tree to Compare Similar Software

Yusuke Sakaguchi, Takashi Ishio, Tetsuya Kanda, Katsuro Inoue, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻, Dept. of Computer Science, Graduate School of Information Science and Technology, Osaka University.

とは困難である．2 つのソフトウェア間でのディレクトリの対応関係をとる手法 [7] [13] はいくつかあるが，3 つ以上のソフトウェアを一度に比較する研究はなされていない．

本研究では，複数のソフトウェアのソースコードから，それらのディレクトリ構造を単一のディレクトリ構造に統一し，可視化するツールを実装する．抽出されるディレクトリ構造は，元のソースコードに含まれるディレクトリを全て含む．ディレクトリの移動やリネームが行われた場合でも，類似したファイルをもつディレクトリの対応関係を自動で判定することにより，1 つのディレクトリにマージされる．このため，抽出されたディレクトリ構造上で開発者はコンテンツの比較を容易に行うことができる．

ツールの入力はソフトウェアのソースコード，出力は単一のディレクトリ構造である．ディレクトリ構造は一般に木構造として知られているので，各ソフトウェアのディレクトリ構造を連結した有向グラフに変換し，そのグラフからスパニングツリーを抽出する．このスパニングツリーをディレクトリ構造とみなし，専用のビューアにより可視化する．このビューアは一般的なファイル管理ツールと同様に，ディレクトリ構造から 1 つのディレクトリを選択することで，そのディレクトリに所属するファイルの一覧を表示する．開発者は，一覧からファイルの類似度を調べることができ，必要に応じて外部ソースコード差分ツールを用いてファイルを比較することができる．

ケーススタディでは，2 社の Android スマートフォンの公開されたカーネルのソースコードを使用し，それらの比較作業を実施した．

以降，2 章では，研究の背景を述べる．3 章では，単一の統合されたディレクトリ構造を抽出し，可視化する手法を説明する．4 章でケーススタディを示し，5 章でまとめを述べる．

2 背景

派生開発は一般的に行われている．Dubinsky ら [4] は，企業のソフトウェア開発者は既存ソフトウェアのソースコードをコピーする傾向にある，と述べている．Homel ら [6] は，コードクローン検出技術を用いて，派生開発で作成された Linux カーネルを分析を実施した結果を報告している．

Duszynski ら [5] は，ソフトウェアのソースコードから共通部分を区別し，可視化する手法を提案した．この可視化は，入力されるソフトウェアのディレクトリ構造が同一であることを前提としているため，移動やリネームされたディレクトリを識別するために，ディレクトリ構造を手動で調べる必要がある．

Yoshimura ら [13] は，2 つの類似したソフトウェアをマージするために，コードクローン検出技術を用いて 2 つのソフトウェア間のディレクトリの対応付けを行った．この手法は，3 つ以上のソフトウェアを比較するために設計はされていない．Holten ら [7] は，2 つのソフトウェアのディレクトリが，相互にどのように対応しているかを可視化する手法を提案した．この手法は構造の違いの全体的な概観を示すことができるが，この手法も複数ソフトウェアの比較を目的とはしていない．リネームされたファイルの追跡は，Lavoie ら [10] によって提案されている．この技術は，ソフトウェアのバージョン間で最も類似しているソースファイルのペアを識別する．本研究では，ディレクトリ間でのファイル類似度を使用して，ディレクトリの対応付けを行う．

Lin ら [11] は，コードクローンの複数の特性に特化したソースコード比較ツールを提案した．ツールはソースコードの部分的なコードクローンの共通点の可視化をするが，ディレクトリレベルの類似性を見つけるツールではない．

3 統一ディレクトリ構造の抽出と可視化

本研究では，類似したソフトウェアのソースコードの構造を単一のディレクトリ構造に統一し，その可視化を行う．提案手法の入力は，ディレクトリの集合 $R = \{r_1, r_2, \cdots, r_{|R|}\}$ とする．ここで r_i は，1 つのソフトウェアのソースコードを含むディレクトリ構造のルートディレクトリを表す．このディレクトリ集合に対して，以下の 3 つのステップを実行することにより，統合されたディレクトリ構造を抽出する．

1. 類似したソースファイルを含むディレクトリの集合に対応するノードを生成する．

2. ソフトウェアのサブディレクトリ関係を表す重み付き辺でノードを接続する．
3. 得られたグラフから，有向スパニングツリーを抽出する．

得られた有向スパニングツリーが求めるディレクトリ構造であり，これを可視化するビューアを作成した．このビューアは，ディレクトリ構造内のソースファイルを比較する機能を備えている．

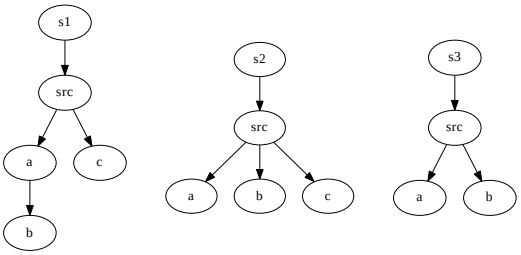

図 1　入力ソフトウェアのディレクトリ構造の例

3.1　ディレクトリ構造の統一

開発者が，複数ソフトウェア間の類似したソースファイルを比較できるようにするために，類似したファイルを含むディレクトリの集合を表現するノードを生成する．2 つのディレクトリの内容の類似度を表すメトリクスとして，$sim(d_1, d_2)$ を導入する．ここで，d_1 と d_2 は，それぞれ異なるソフトウェアのディレクトリである．$sim(d_1, d_2)$ が予め設定した閾値 th 以上である場合，d_1 と d_2 を単一のノードで表現する．$sim(d_1, d_2) \geq th$ かつ $sim(d_2, d_3) \geq th$ であっても $sim(d_1, d_3) \geq th$ となるとは限らないが，それでも開発者がディレクトリ間の差異を分析できるように，これら全てのディレクトリを同じノードに割り当てる．以降，ノード n によって表されるディレクトリの集合を $D(n)$ と表現する．

2 つのディレクトリの内容の類似度を表す $sim(d_1, d_2)$ は，以下のようにジャッカード類似度によって定義される．

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

ここで，$L(d)$ は，ディレクトリ d に含まれるテキストファイルの行の集合である（テキストかどうかはファイルの拡張子で判定する）．ソフトウェアによってソースコードのレイアウトが変更されたり改行文字が変更されたりする場合があるので，各行を取り出した時点で空白，改行文字を行中からすべて取り除いて使用する．

例として，3 つのコンポーネント `a`, `b`, `c` で構成されている 3 つのソフトウェア `s1`, `s2`, `s3` が入力された場合を考える．図 1 のように各コンポーネントが異なるディレクトリに格納されていたとすると，それぞれのコンポーネントに対応する 3 つのノード `na`, `nb`, `nc` が以下のようにディレクトリに対応付けられる．

$$D(\texttt{na}) = \{\texttt{s1/src/a}, \texttt{s2/src/a}, \texttt{s3/src/a}\}$$
$$D(\texttt{nb}) = \{\texttt{s1/src/a/b}, \texttt{s2/src/b}, \texttt{s3/src/b}\}$$
$$D(\texttt{nc}) = \{\texttt{s1/src/c}, \texttt{s2/src/c}\}$$

ディレクトリの類似度をそこに所属するファイルの類似度によって定義したが，ソフトウェアのソースコードの管理には，ファイルを持たないディレクトリも使用されている．例えば，多くの Java プログラムの `src` ディレクトリは，Java パッケージ名を表すサブディレクトリしか含まない．そこで，このようなファイルを持たないディレクトリについては，サブディレクトリが同一ノードで表される場合，それらのディレクトリを単一のノードに割り当てる．言い換えると，2 つのディレクトリ d_1, d_2 が，単一のノード n に割り当てられている時 $(d_1, d_2 \in D(n))$，それらの親ディレクトリ d_{p1}, d_{p2} は共通の親ノードに割り当てられる．図 1 の例のディレクトリ構造の場合，それぞれのソフトウェアのディレクトリ `src` が，たとえソースファイルを持っていなくても，単一のノードに割り当てられる．

得られるグラフが連結グラフであることを保証するために，ソフトウェアの類似性に関わらず，全てのルートディレクトリを表すルートノード r を設定する．このルートノードが，抽出される統一されたディレクトリ構造のルートディレクトリとなる．

ノードの生成が完了したら，それらのノードを重み付き辺で接続する．すべてのノードの組 n_1, n_2 に対

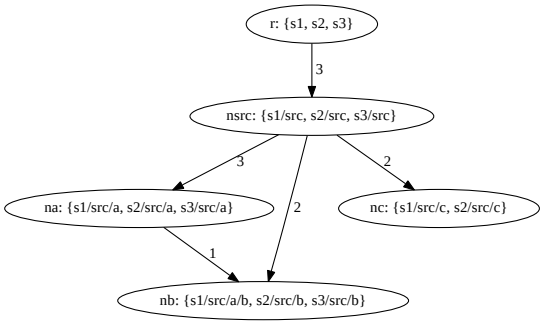

図 2　図 1 の例から得られるディレクトリグラフ

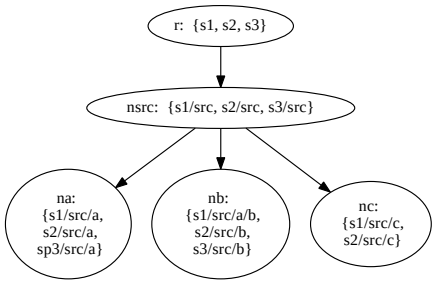

図 3　図 1 の例から得られる統一ディレクトリ構造

して，それらの重みを以下のように計算する．

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

ここで， $subdir(d_1, d_2)$ は，d_1 が d_2 の直接の親ディレクトリであることを意味する．$w(n_1, n_2) > 0$ である場合，その重みを持つノード間の有向辺を作成する．ディレクトリを表すノードは必ずそのサブディレクトリを表すノードへの辺を持つことになるため，ルートノード r からは全てのノードへ到達可能になる．この手順の結果，例のディレクトリ構造は，図 2 のような有向グラフへ変換される．ディレクトリ `b` は，`s1` では `a` のサブディレクトリであり，`s2`, `s3` では `src` のサブディレクトリであるため，対応するノード `nb` にはそれらの関係を表現した 2 本の辺が接続されている．

辺の接続によって得られた有向グラフからスパニングツリーを抽出する．この抽出は，以下のような貪欲法の一種である．

1. 初期状態として，スパニングツリー $V = \{r\}, E = \phi$ を生成する．V は，頂点の集合を表し，E は，選択された辺の集合を表す．
2. $t \in V, n \notin V$ を満たす最大の重み $w(t, n)$ を持つ辺 (t, n) を選択する．複数ある場合，r に最も近いものを選択する．
3. スパニングツリーに (t, n) を加える．すなわち $V \leftarrow V \cup \{n\}$, $E \leftarrow E \cup \{(t, n)\}$ とする．
4. 全てのノードが V に含まれるまで，ステップ 2, 3 を繰り返す．最終的に E がスパニングツリーの辺の集合となる．

このアルゴリズムによって，図 2 のグラフから図 3 のようなスパニングツリーを得ることができる．このツリーは，`nsrc` から `na`, `nb`, `nc` への辺を含み，`na` から `nb` への辺は含まない．従って，このスパニングツリーが表すディレクトリ構造は，`s2` のディレクトリ構造と同じになる．ディレクトリ `s1/src/a/b` は，他の 2 つのソフトウェアの類似したディレクトリが `src` のサブディレクトリに位置するため，`src` のサブディレクトリとみなされたことになる．

3.2　特定のソフトウェアの構造への統一

先に示したアルゴリズムは，入力されたすべてのソフトウェアのディレクトリ構造を公平に扱う．そのため，統一されたディレクトリ構造は，入力された全てのソフトウェアのディレクトリ構造のいずれとも異なる可能性がある．一方で，開発者が特定のソフトウェアについて既に情報を持っている場合には，他のソフトウェアのディレクトリ構造をそのソフトウェアのディレクトリ構造に揃えて閲覧することも可能としたい．そのために，特定のソフトウェア P が選択されたとき，そのディレクトリ構造に統一できるように，以下のように辺の重みを拡張する．

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

この関数を用いることで，目的のソフトウェアにお

けるサブディレクトリの関係を表す辺は，スパニングツリーを抽出する際に優先して選択される．そして，その後に，他のソフトウェアに固有のディレクトリを接続する辺が選択される．

3.3 ディレクトリビューア

統一されたディレクトリ構造を可視化する専用のビューアを設計，実装した．このビューアは，統一されたディレクトリ構造を扱うことができ，各ノードのディレクトリのファイルの内容の違いについて調べることができる．ツールのスクリーンショットを図 4 に示す．ツール上部に，ソフトウェアを指定しない状態での統一ディレクトリの表示と，特定のソフトウェアのディレクトリ構造に合わせた表示とを切り替える選択ボタンがある．残りの領域は，ディレクトリ構造を示す Tree View，選択ノード内のファイルを示す File List View，各ソフトウェアのファイルの類似度を示す File Matrix View の 3 つの領域からなる．

3.3.1 Tree View

Tree View は統一されたディレクトリ構造を示す．各ノードは，抽出されたスパニングツリーのノードに対応しており，元となったソースコードの 1 つ以上のディレクトリに対応する．それぞれのノードの名前は，ノードに表現されているディレクトリ名を元にしており，複数のディレクトリに対応するノードでは，最も多いディレクトリ名が選ばれる．

ノードの名前の横に，それぞれのノードに含まれるディレクトリ数 x と，その内の変種の数 y を，(y in x) として示す．変種の数とは，互いに異なるファイル内容を含むディレクトリ数のことである．例えば，(2 in 3) とあるノードは，3 つのディレクトリを含むが，そのうち 1 つが，他とは異なるファイル内容を持つことを表す．(1 in 3) とあるノードは，それらすべてのディレクトリに同一の内容が含まれていることを表す．変種の数は，ノードで表されるディレクトリに直接含まれるファイル内容を基にしていることに注意が必要である．たとえば (1 in 3) とあるノードの各ディレクトリのサブディレクトリには，変更が行われている場合がある．2 つ以上の変種を含む（ソフトウェア間に何らかの変更がある）ノードは青文字で表示される．

“移動した” ディレクトリ，すなわち，ディレクトリ名やルートからの経路が Tree View 上での見た目とは異なるディレクトリを含むノードには，ノードの名前の横にさらに “*” の印が付与される．

3.3.2 File List View

File List View は，選択したノード内に含まれるファイルの一覧を示す．複数のソフトウェアに存在するディレクトリの内容をまとめて表示するため，縦方向（1 列目）にファイル名を一覧表示し，横方向をノードに含まれるディレクトリに対応付けた表形式で表現する．それぞれのセルは，ディレクトリ内のファイルのハッシュ値を表す．その行に並ぶすべてのハッシュ値が等しいとき，その値は灰色で表現される．黒で描画された値は，ファイル間で異なる内容を持つことを表す．

異なる内容を持つファイルにすばやく注目できるように，この表は，ファイル名かハッシュ値の色によってソートが可能である．

表に含まれるディレクトリのフルパス名は，リストの下に表記されている．このフルパス名をクリックすると，既定のファイル管理ツール（Microsoft Windows 環境では Explorer）でそのディレクトリが開かれるので，開発者はすぐに必要なファイル操作を行うことができる．

3.3.3 File Matrix View

File Matrix View は，File List View から選択した同名のファイルの類似度を示す．行と列は，File List View のディレクトリの順に対応している．セル (i, j) は，選択したディレクトリのファイル間の類似度 $sim(f_i, f_j)$ を示す．各セルは，類似度 sim の値に応じて白 (sim=1) から黄 (sim=0.5)，赤 (sim=0) のカラースケールで色づけされる．そのため，入力したソフトウェア間のファイルがどのくらい異なっているのかを確認することができる．

詳細を確認したい場合は，セルを 1 つ選択すると，外部の比較ツール（本ツールの現在の実装では WinMerge †1）を起動して対応するファイルの比較

†1 http://winmerge.org

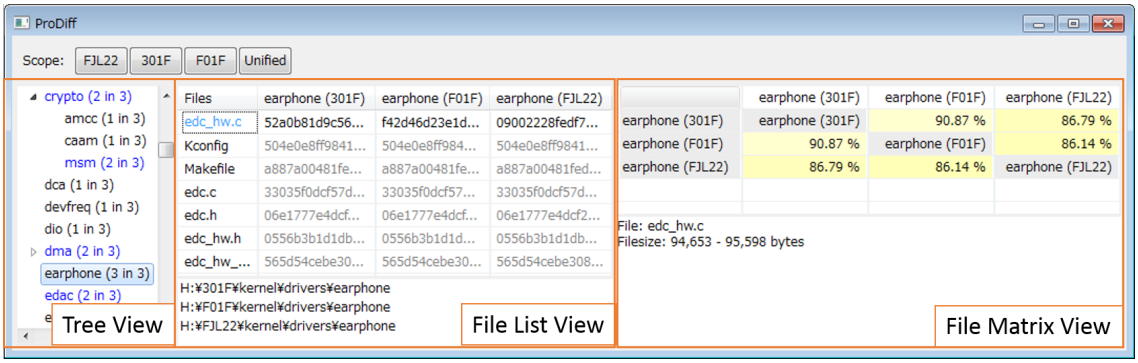

図 4　実装したビューアのスクリーンショット

結果を可視化する．

4　ケーススタディ

ツールの機能を確認するために，ケーススタディを実装した．ツールの入力には，2013 年の第 4 四半期にリリースされた Android スマートフォン 4 機種の公開されているソースコードを使用した．この 4 機種は，Fujitsu の 3 機種と Sony の 1 機種からなり，いずれも CPU は Qualcomm MSM8974，Android のバージョンも 4.2 と共通している．各機種の発売時期や規模の情報を表 1 に示す．

この事前情報だけからでも，類似したソフトウェアのバージョンを入力としたため，ほとんどのソースコードは共通であり，いくつか特定の部分が互いに異なっていると予測できる．さらに，ベンダーが異なる SO-01F のいくつかのディレクトリとファイルが，他の機種のものとは異なるとも予測できる．ケーススタディでは，実装したツールを使用して，これらの製品のコードを実際に探索する．

まず，4 つのソフトウェアをツールに入力すると，解析が実行された．ファイルの読み込み，ディレクトリ間の類似度比較，統一されたディレクトリ構造の抽出に約 42 分を要した．類似度の閾値は，$th = 0.8$ としている．

得られた統一されたディレクトリ構造は，9,037 ノードで構成されていた．そのうち 245 ノードは移動ないし名前が変更されたディレクトリを含んでいた．また，673 ノードが互いに異なるファイルを含むディレクトリ群を表現していた．

ディレクトリ `/kernel`（図 5）を見ると，内容が異なるファイルがいくつかあることがわかった．例えば，SO-01F のファイル `AndroidKernel.mk` と `Makefile` は，他の機種のものとは異なっている．Fujitsu 製品の `Makefile` は，互いに少しの変更があり，SO-01F との類似度も 95.01%であった．一方，図 5 の File Matrix View に示すように，`AndroidKernel.mk` の類似度は 40.52%であった．開発者は，カーネルビルドのための特別な設定を，主にこのファイルで行っているということがわかる．F-01F の `MAINTAINERS` は，他の製品とは異なっている．すべての製品は，同じカーネルバージョンを使用しているが，F-01F のいくつかのファイルには非アスキー文字（たとえばウムラウトの付いた文字）が含まれていない．これは，そのような文字が開発途中に何らかの理由で置換されたと思われる．

Fujitsu の 3 機種のディレクトリ `/vendor` に含まれるファイルは，SO-01F の `/vendor` に含まれるものとは全て異なっているので，それらのディレクトリは，統一されたディレクトリ構造で独立したノードとして表された．同様に，`/external/chronium` ディレクトリも，SO-01F とは区別された．このディレクトリでは，例えば `Android.mk` というファイルが Fujitsu のソースコードからはなくなっていた．一方，`/external/chronium` の共通のサブディレクトリは 4 機種間でほとんど類似している．その結果，共通のサブディレクトリは，Fujistu 側の機種に

表 1　ケーススタディの入力一覧．すべて **Android4.2** を搭載

製品名	ベンダー	通信事業者	発売時期	#Dirs	#Files	#Lines
FJL22	Fujitsu	au	2013/11	7,683	107,945	26,178,588
301F	Fujitsu	SoftBank	2013/12	7,708	108,334	25,629,778
F-01F	Fujitsu	NTT DOCOMO	2013/10	7,582	105,397	25,740,695
SO-01F	Sony	NTT DOCOMO	2013/10	5,840	90,736	22,225,611

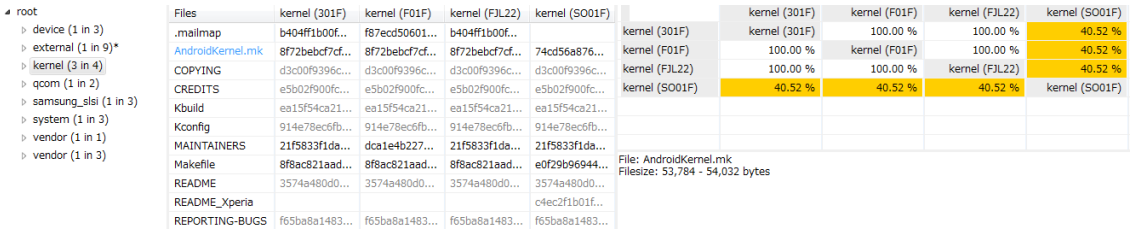

図 5　ツールで `/kernel` ディレクトリ内部の `AndroidKernel.mk` ファイルを選択した状態の画面

対応する `/external/chronium` ノードのサブディレクトリとして表示されていた．SO-01F に対応する `/external/chronium` は，SO-01F に固有のサブディレクトリだけを保有した状態となっていた．

ノード `/external/llvm` は，Fujitsu 製品の 6 つのディレクトリから成り，そのうち 3 ディレクトリは，`/external/llvm`，残りの 3 ディレクトリは `/external/llvm/projects/sample` であった．ディレクトリ `sample` が偶然にマージされてしまったのは，ファイル `configure` のような類似したファイルをもつためである．

ケーススタディでは，4 つの個別に公開されているソフトウェアが 1 つの製品であるかのように統一されたディレクトリ構造扱うことができ，そして，違いを持つディレクトリに簡単に焦点を当てることができた．4 つの Android 製品のソースコードは，合計で 28,813 ディレクトリであったが，統一されたディレクトリ構造は，その全てを 9,037 ノードで表した．ディレクトリが著しく異なるファイル内容を持つときは，それらは独立したノードとして表されるため，そのようなディレクトリについては詳細な分析を行わずに済ませることができた．さらに，サブディレクトリは，親ディレクトリとは独立に対応関係が計算されるため，ディレクトリごとに個別に類似ファイルを比較することもできた．

5　まとめ

ソースコード比較は，ソフトウェア間の共通部分や固有部分を理解するために重要である．本研究では，複数ソフトウェアの全てのディレクトリを含む，統一されたディレクトリ構造を抽出する手法を提案した．統一されたディレクトリ構造を可視化するためのビューアを設計し，4 つの Android 製品に対してケーススタディを実施した．その結果，ツールがソフトウェア間の異なるファイルの発見，比較を容易にすることを定性的に確認した．

今後の課題としては，抽出された統一ディレクトリ構造の質について評価を行うことが挙げられる．また，実装したツールがソースコード比較のために有効であるかどうかを，対照実験によって評価したい．

謝辞 本研究は JSPS 科研費（課題番号 25220003, 26280021）の助成を受けたものです．

参 考 文 献

[1] Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization, *Proceedings of the 2nd Conference Software Product Line Conference*, 2002, pp. 257–271.

[2] de Jonge, M.: Multi-level component composition, *Proceedings of the 2nd Groningen Workshop*

Software Variability Modeling, No. 2004-7, 2004.

[3] de Jonge, M.: Build-level components, *IEEE Transactions on Software Engineering*, Vol. 31, No. 7(2005), pp. 588–600.

[4] Dubinsky, Y., Rubin, J., Berger, T., Duszynski, S., Becker, M., and Czarnecki, K.: An Exploratory Study of Cloning in Industrial Software Product Lines, *Proceedings of the 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 25–34.

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

[6] Hemel, A. and Koschke, R.: Reverse Engineering Variability in Source Code Using Clone Detection: A Case Study for Linux Variants of Consumer Electronic Devices, *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 357–366.

[7] Holten, D. and van Wijk, J. J.: Visual Comparison of Hierarchically Organized Data, *Proceedings of the 10th Joint Eurographics / IEEE - VGTC Conference on Visualization*, 2008, pp. 759–766.

[8] Kawamitsu, N., Ishio, T., Kanda, T., Kula, R. G., De Roover, C., and Inoue, K.: Identifying Source Code Reuse across Repositories using LCS-based Source Code Similarity, *Proceedings of the 14th International Working Conference on Source Code Analysis and Manipulation*, 2014, pp. 305–314.

[9] Krueger, C. W.: Easing the Transition to Software Mass Customization, *Revised Papers from the 4th International Workshop on Software Product-Family Engineering*, 2002, pp. 282–293.

[10] Lavoie, T., Khomh, F., Merlo, E., and Zou, Y.: Inferring Repository File Structure Modifications Using Nearest-Neighbor Clone Detection, *Proceedings of the 19th Working Conference on Reverse Engineering*, 2012, pp. 325–334.

[11] Lin, Y., Xing, Z., Xue, Y., Liu, Y., Peng, X., Sun, J., and Zhao, W.: Detecting Differences across Multiple Instances of Code Clones, *Proceedings of the 36th International Conference on Software Engeneering*, 2014, pp. 164–174.

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

[13] Yoshimura, K., Ganesan, D., and Muthig, D.: Assessing Merge Potential of Existing Engine Control Systems into a Product Line, *Proceedings of the International Workshop on Software Engineering for Automotive Systems*, 2006, pp. 61–67.