ソースファイルの派生関係の自動抽出

神田 哲也[1,a)]　石尾 隆[1,b)]　井上 克郎[1,c)]

概要：ソフトウェアの進化において，1つのソースコードから複数の派生ソースコードが作られることがある．それら派生ソースコードの中から最新のバグ修正が施されたものを選択したい場合や，開発が分岐した複数のソフトウェアから機能を集約したい場合には，ソースコードの派生関係を知ることが重要になる．本研究では，類似するファイルの差分に注目することで，ソースコード集合のバージョン履歴がない状態から，ソースファイルの派生関係を自動抽出する手法を提案する．具体的には，差分の行数をもとに最小全域木を構築することで，開発者が比較を行う際に読解する差分の量を最小化する．例としてオープンソースのソースコードに手法を適用し，結果をもとに今後の課題について考察した．

キーワード：ソフトウェア進化，差分

Towards Automatic Extraction of the Derivative Relationships of Source Files

Kanda Tetsuya[1,a)]　Ishio Takashi[1,b)]　Inoue Katsuro[1,c)]

Abstract: In software evolution, branching of development occurs in some cases. If a developer wants to select a source file to which the latest bug fixes was performed or to collect functions out of branched software, it is important to know the derivative relationships of source files. In this paper, we focus on the differences of similar files and propose an approach for automatic extraction of the derivative relationships of source files. We construct a minimum spanning tree based on the number of lines of difference. The tree minimizes the amount of difference developers read. We performed a case study with open source software and discussed for the future work.

Keywords: Software evolution, difference

1. はじめに

ソフトウェアは，機能を追加，削除あるいは修正されることにより進化する．ソフトウェアの進化は必ずしも一本道ではなく，バージョン管理システムを用いて機能追加・テスト用のブランチをリリース用のブランチと別に作り，並行して開発を行うことも多い．ときには元のソフトウェアから派生して，以後別系統の進化をたどるソフトウェアも生まれる．この場合には，開発元とは別のチームがソフトウェアを改良し，進化させていくケースも考えられる．開発チームが別個になる場合，ソースコードの管理も別個に行われるため，元のソフトウェアとの関連性が明らかでなくなる．

ソフトウェアの派生関係を知ることは，ソフトウェアの再利用において重要となる．新たな派生ソフトウェアを作るにあたって，できるだけ新しい，バグ修正の行われたソースファイルを選択して再利用したい場合 [1] や，開発が分岐した複数のソフトウェアの機能を1つのプロダクトに集約したい場合に，複数の既存ソフトウェアの派生関係を調査する必要がある．

バージョン違いや分岐して進化したソフトウェア間のソースコードを比較すると，元が同一であるため非常に類

[1] 大阪大学
Osaka University, Suita, Osaka, 565–0871, Japan
a) t-kanda@ist.osaka-u.ac.jp
b) ishio@ist.osaka-u.ac.jp
c) inoue@ist.osaka-u.ac.jp

© 2012 Information Processing Society of Japan

似度の高いものとなる．ソースコードの類似性に関する研究として，コードクローン検出技術が挙げられる．コードクローンとは，ソースコード中に存在する互いに一致・類似したコード片のことであり，大規模なソフトウェア集合においてもソフトウェア間でのコードクローンが多く存在することが知られている．コードクローン検出技術では，ソースコード中の類似する部分に着目しているが，元が同一のソフトウェアのソースコードはその大部分がクローンとなってしまうため，そこからソースコード間の関係性を理解することは難しい．

本研究では，類似するファイルの差分に注目する．開発者が類似ファイルの比較をすべての組み合わせで行わずに済むよう，差分の量が最小になる比較順序を求めることで，ソースコード集合のバージョン履歴がない状態からソースファイルの派生関係を自動抽出する．

第 2 章では，本研究の背景となる事項について述べる．その後，第 3 章で提案する解析手法について説明し．第 4 章でいくつかのソースコードに対し提案手法を簡単に適用した結果について述べる．第 5 章で関連研究を．第 6 章で，まとめと今後の課題を記述する．

2. 背景

2.1 ソフトウェアの派生

本稿では，あるソフトウェアのバージョン違いや，分岐によって派生したソフトウェアをまとめて「派生ソフトウェア」と呼ぶ．ソフトウェアの分岐の要因として，開発ブランチの作成とプロジェクト自体の分岐が挙げられる．

ソフトウェアの進化を明らかにする試みは多く行われている．Prinzger らはメトリクスと可視化を組み合わせてソフトウェアの進化を示した [2]．Jaafar は，同時期に修正されるクラスに注目してソフトウェアのバージョン間の関係を抽出することを提案した [3]．

これらの研究はソフトウェアのバージョン間の具体的な差異については考慮しておらず，ソフトウェアの差異に関する開発者の理解を支援することが目的ではない．

開発ブランチ

ソフトウェアの開発において，リリース用のリリースブランチと機能追加・テスト用の開発ブランチとにソースコードを分岐させることが行われている [4]．

図 1 の例では 2 種類の開発ブランチで機能追加をし，安定したらリリースブランチに統合している．この場合，統合後の版はリリースブランチに属するが，リリースブランチの統合前の版よりも開発ブランチの直前の版のほうが，類似度が高い場合も考えられる．全てのバージョンを時系列順に並べると図 1 下のようになるが，上と比較して，ソフトウェアの派生関係を的確には表していないことがわかる．

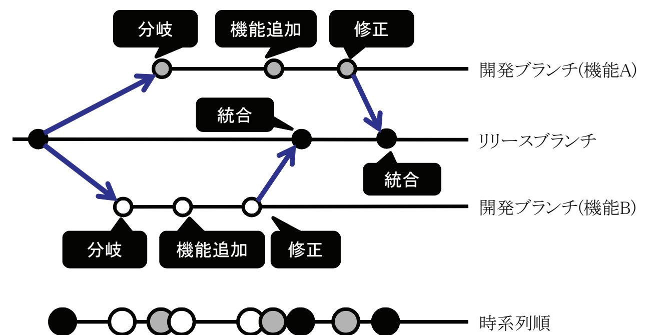


図 1 開発ブランチを用いたソフトウェアの進化
Fig. 1 Software evolution with development branches

プロジェクトの分岐

進化の過程において，元のソフトウェアに変更を加えたものが，以後別系統のソフトウェアとして進化していくことがある．ソフトウェアの開発プロジェクト自体が，別の開発チームに移動し，プロジェクトが分岐することも考えられる．オープンソースでの実例として有名なものは，FreeBSD や NetBSD，OpenBSD などの BSD 系のオペレーティングシステムがある．

プロジェクト自体が分岐した場合，ソフトウェアリポジトリも別個のものとなる．しかし，これらの分岐したプロジェクトは以後無関係というわけでもなく，あるソフトウェアの派生ソフトウェアがバージョンアップに際し，元のソフトウェアの機能追加やバグ修正を取り込む場合もある．このようなケースにおいて開発の分岐が発生した場合，元のソフトウェアと派生ソフトウェアとがどのように関連しているのかが不明瞭になる．

2.2 コードの類似関係

派生関係にあるソフトウェア間では元が同一であるために，対応するファイル同士は高い類似度を持つと考えられる．ソフトウェアの派生関係を明らかにするためには，あるバージョンのソースファイルが別のバージョンのどのソースファイルに派生したかという対応関係を求め，内容を比較することになる．この対応関係を求めるためには，ソースコードの類似度が有効である．

Yoshimura らは産業向けシステムのソースコードについて，類似度の高いファイルの存在を可視化した [5]．Yoshimura らの手法では，どのファイル同士が類似しているかをグラフの形で図示することができるが，類似しているファイル同士をグループ化するだけであり，それらのファイル間の実際の関係については，ファイルの内容を手作業で調査する必要がある．

Inoue らが開発した Ichi Tracker[1] では，コード断片をソースコード検索エンジンで検索し，検索結果のソースファイルをコード断片との類似度でクラスタリングする．その上で，ソースファイルを時系列順に並べることにより

© 2012 Information Processing Society of Japan

ソースコードの再利用の経緯を可視化する．ただし，このツールでは開発者が注目している 1 ファイルとの類似度だけを用いてファイルの可視化を行っており，互いに異なるファイルであっても元のファイルからの類似度が等しければ，同一のグループに所属しているかのように可視化されてしまう．

本研究ではソースコードの内容を用いて，ソフトウェアの派生関係の抽出を目指す．また，ファイル間の差異に重点を置くことで，開発者の理解を支援する．

2.3 ソースファイルの比較

実際にソフトウェアの進化を理解するためには，「このファイル同士が似ている」という情報だけではなく，「ファイル間でどの部分がどのように変更されたのか」という情報が必要になる．

ソースファイルを比較する手段としては，2 ファイル間の差分を行の追加と削除の 2 種類で表現する UNIX diff[6] が一般的である．本稿では差分とは Unix diff の出力を指すものとする．バージョン管理システムでは，あるファイルと，別々の変更を行った 2 ファイルとを比較，統合する 3-way マージ機能を備えているものもあるが，任意の 3 ファイルを比較するものではない．任意の 3 ファイル以上を比較するツールはいくつか存在するが，ほとんどは 2 ファイルの比較を並べたものであるため，ファイルの比較は 2 ファイル間で行うことが一般的であると言える．

類似するソースファイル間のすべての差分を調査するコストは，ファイル数の増加に従い大きくなる．N 個のソースファイルに対してファイル間比較を行うには，単純計算で $N(N-1)/2$ 回の比較が必要となり，ファイル数の 2 乗の速度で比較回数が増加する．これは計算機にとっては問題ないが，差分を実際に読み解く開発者の負担が非常に大きくなる可能性がある．

適当に比較順序を定めることで，比較回数は $N-1$ 回に減らすことが可能である．しかし，比較順序の設定方法によっては，2 つの差分間において重複する部分が存在してしまう．例えば図 2(a) の順序で 3 つのファイルを比較する場合，出力される 2 つの差分の中に methodC(); が 2 回出現する．この例では差分は 1 行のメソッド呼び出しであるが，差分が複数行にわたる処理になるケースでも同様の重複は起こりうる．重複する部分を何度も調査することは非効率的であるので，差分の重複が少なくなるような比較順序を考える必要がある．先の例では，図 2(b) の順に比較することで，差分の重複を解消することができる．差分の重複が少ない比較順序としてはソフトウェアが進化していった順序をたどるのが妥当であるが，ソフトウェアの進化順序が明確でない場合も考えられる．ソフトウェアが単一の組織で枝分かれなく進化しているならば，ソースファイルを更新日時の順に並び替えればよい．また，ソースコードがバージョン管理システムによって管理されていれば，コミットログからソフトウェアの派生履歴を追跡可能である．しかし，バージョン管理システムが用いられていない場合，あるいはリポジトリが複数に分かれてしまっている場合もあり，これらは常に利用可能な情報とは言えない．

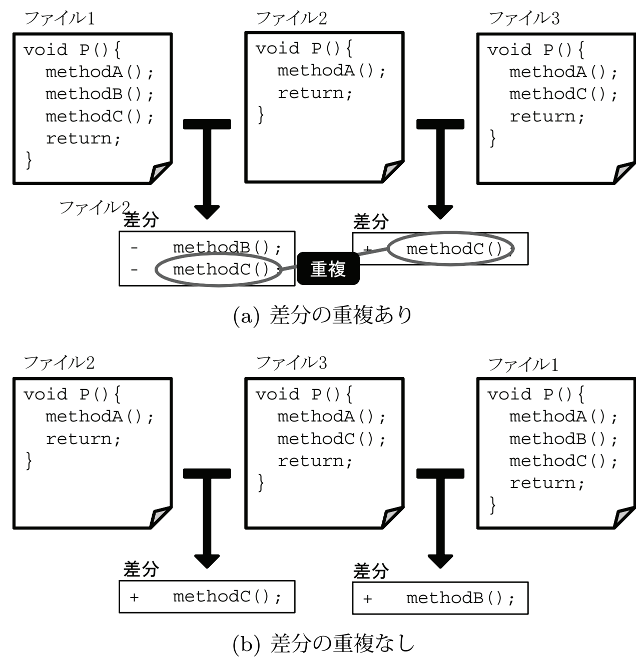


図 2 比較順序による差分の重複の有無
Fig. 2 Overlaps in the difference

3. 提案手法

提案手法では，ソースファイルの近似した派生関係を定義し，それに基づいて Java で書かれた類似ソフトウェア集合の解析を以下の手順で行い，ソースファイルの派生関係を自動抽出する．

- 類似関係にあるファイルのグループ化
- 類似関係にあるファイル間の派生関係の抽出

近似した派生関係の定義について述べ，各手順について，詳細に説明していく．

3.1 派生関係の近似

あるソースコード A1 を変更してソースコード A2 を作成したとき，A1 と A2 の間には派生関係があると言える．本稿ではこれを「本来の派生関係」と呼ぶ．本来の派生関係には，コードの進化の向き（この例では A1 → A2）が存在する．

ソースコードの解析のみで本来の派生関係を得ることは難しいため，本研究ではソースコードから得られる情報によって派生関係を近似する．関連研究 [1], [5] では，類似するファイルを検出するためにファイル間の類似度を利用していた．実際にファイルを比較する際には，ソースコードを人間が読解するか，あるいはツールで処理することに

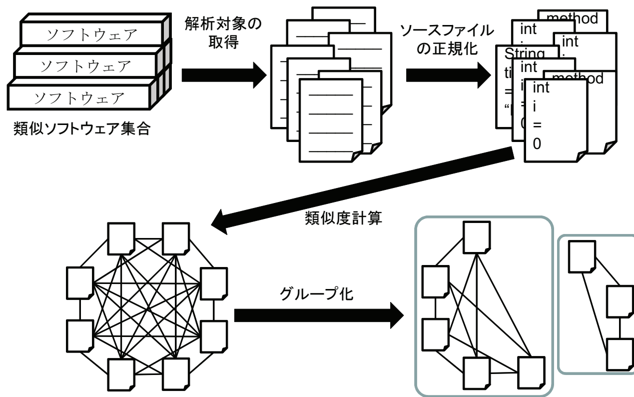


図 3　グループ化手順の概要
Fig. 3　An overview of grouping

なる．その際，ソースファイルがどれほど類似しているかの割合よりも，実際の行数のほうが差分の理解に必要なコストと関係があると考えた．そこで本研究では，派生関係がないファイル間よりも，派生関係があるファイル間の方が差分の行数が小さいと仮定する．つまり，あるソースファイル A1 と，A1 から見て最も差分の行数が小さいソースファイル A2 との間には派生関係があるとみなす．類似ファイル集合の中で，最も差分の行数が小さいソースファイル間から順にすべてのファイルを連結することで，ファイル集合の中での近似した派生関係を得る．以後本稿では「派生関係」とは，この近似した派生関係を意味する．どちらのファイルが元になっているかはわからないため，この定義では派生関係の向きは定めない．本来の派生関係では進化の向きが存在するため，進化の始点となるファイルを決定可能である．この場合ソースファイルの比較の際には，始点となるファイルから順に本来の派生関係をたどればよい．一方，本研究での派生関係は向きを定めていないために，進化の始点がわからないために，始点となるファイルを手作業で調査する必要がある．そこで，すべての類似するソースファイルが派生関係で連結されたのち，差分を読む総行数が少なくなるようなファイルを計算することで，比較の際の始点を提示する．

3.2　類似関係にあるファイルのグループ化

この手順は Yoshimura らの手法 [5] に基づき，類似ソフトウェアの集合から，類似関係にあるファイルを抽出しグループ化する．図 3 にグループ化までの手順の概要を示す．

解析対象の取得

入力された類似ソフトウェアのファイル集合から，解析対象とするものを取得する．今回の解析では Java で記述されたものを対象としているため，入力されたファイル集合から拡張子が.java のものを特定する．

ソースファイルの正規化

コーディングスタイルの違いを吸収するためにソースコードを正規化する．解析対象の各ファイルに対し字句解析を行い，ソースコードを 1 行 1 トークンに切り分ける．その後コメントおよび空白を除去する．これにより各ファイルが正規化された状態となる．今回は識別子名の正規化は行っていない．

類似度計算

正規化後のファイル同士の差分をとり，追加，削除された行数と共通している行数を数え，類似度計算を行う．ファイル間差分の計算には，java-diff[7] を用いた．正規化後の 2 ファイル A,B の類似度 Simirality(A,B) は式 1 であらわされる．使用する差分計算ツールの出力により式 2 に変形して利用可能である．

$$Similarity(A,B)$$
$$= \frac{\text{A,B 間で共通する行数}}{\text{A,B 間で共通する行数} + \text{A,B 間の差分の行数}} \quad (1)$$
$$= \frac{\text{A の行数} + \text{B の行数} - \text{A,B 間の差分の行数}}{\text{A の行数} + \text{B の行数} + \text{A,B 間の差分の行数}} \quad (2)$$

A,B は正規化後のファイルであるので，「A の行数」とは「A の元ファイルをトークン化しコメントと空白を除いた長さ」である．

また，ファイルサイズが大きく違うものは類似度が低いため，今回はファイルサイズに 2 倍以上の差がある場合には類似度の計算を省略した．

類似度計算の後，ソースファイルを頂点，ファイル間の類似度を辺の重みとした無向重みつきグラフを生成する．

グループ化

次に，無向重みつきグラフの各辺について，類似度がしきい値以下の辺を削除する．これにより類似度の高いファイル同士のみが連結されたグラフが生成される．辺を削除したことにより，グラフは非連結な複数のグラフに分割される．分割された各グラフは，大量のソースファイル集合から類似関係にあるファイルをグループ化したものとなる．

3.3　類似関係にあるファイル間の派生関係の抽出

類似関係にあるファイルをグループ化した後，グループごとにファイルの派生関係を抽出する．また，この手順においては派生関係をどのファイルからたどり始めるとよいか，その始点となるファイルを提示する．

差分の計算

グループ内の正規化後のファイルについて，ソースファイルを頂点，差分の行数を辺の重みとした無向重みつきグラフを生成する．今回はグループ化の際に取得した差分の行数をそのまま利用する．

最小全域木の構築

生成したグラフに対し，最小全域木を求める．全域木とは，あるグラフの部分グラフのうち，すべての頂点が連結されており，かつ閉路が無いグラフである．最小全域木とは，グラフを構成する辺の重みの総和が最小となる全域木である．最小全域木を求めることで，すべてのファイルを

© 2012 Information Processing Society of Japan

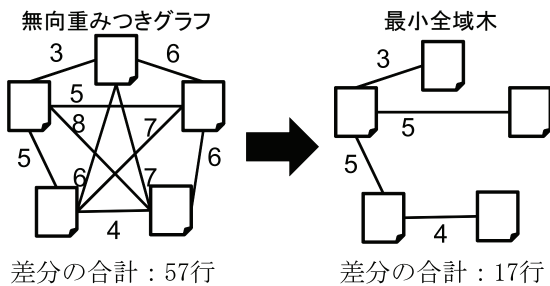


図 4　最小全域木の構築
Fig. 4　Structureing a minimum spanning tree

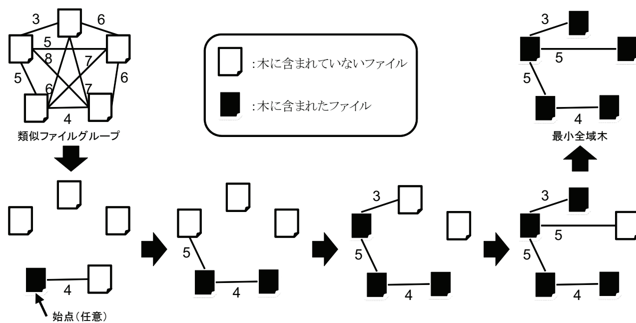


図 5　Prim 法による最小全域木の構築
Fig. 5　Finding a minimum spanning tree using Prim's algorithm

含み，かつファイル間差分の行数の合計が最小となるような木を得ることができる（図 4）．この木が，自動抽出したファイル間の派生関係である．最小全域木を求めるアルゴリズムとして，Prim 法 [8] を用いた．アルゴリズムの適用例を図 5 に示す．Prim 法では，始めに元となるグラフから任意の頂点を木に追加する．木に含まれる頂点から木に含まれない頂点への辺のうち，重みが最も小さいものを順次木に追加していく．重みが最も小さい片が複数ある場合は，そのうちどれか 1 つを選択する．選択する辺によって最小全域木の形は変わるものの，最終的な重みの合計は変化しないことがアルゴリズムで保証されている．この作業をすべての頂点が木に含まれるまで行った時，作成した木は最小全域木となっている．

始点の選定

「他のファイルへの差分」の合計が最小になるファイルを，比較の始点として提示する．最小全域木内のすべての頂点について，他の頂点への経路の重みの合計を計算する．全域木は閉路を持たず任意の頂点間の経路が一意に定まるので，この値も一意である．経路の重みの合計が最小になる頂点を，比較の始点とする（図 6）．

提示した始点からの比較順序は開発者が読む差分の量を削減することを目的としているため，本来の派生関係の順序（ソースファイルの新旧）とは一致しないことに注意する．最小全域木と，始点を示すことで，開発者が読むべき差分の量を減らしたファイルの派生関係を提示できる．

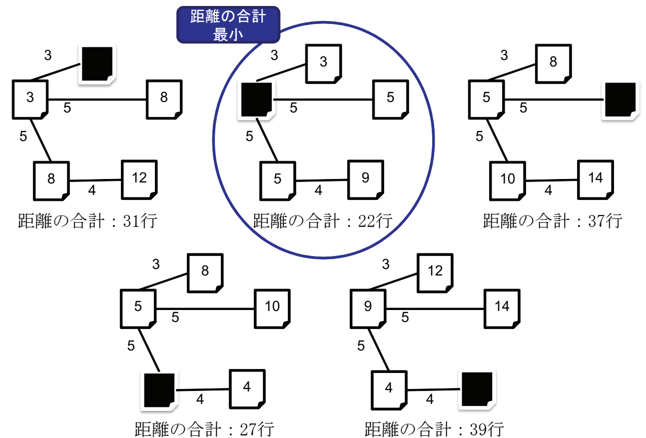


図 6　他ファイルへの差分の合計を計算
Fig. 6　Caluculating the sum of the defferences

4. ケーススタディ

手法を簡単に実装し，いくつかのソースコードに適用して結果の調査を行った．同一ソフトウェアのバージョン違いのほかに，ソフトウェアの単一のバージョン内のファイルに対しても解析を行い，類似ファイル同士の関係を可視化した．グループ化の際の類似度のしきい値は 0.8 とした．

4.1 結果

単一バージョン内の類似ファイルの比較

対象として携帯電話端末 F-05D[9] に含まれる OpenHealthManager のソースコードを用いた．抽出された 2 ファイル以上を含む類似ファイル集合は 13 あった．その中から 1 つを抜き出したのが図 7 である．派生関係の始点となる頂点は二重線で囲って示した．

図 7(a) からわかるように，ファイル間の差分は 24 行と 48 行の 2 種類しかない．そのため図 7(b) はここから得られる唯一の最小全域木ではない．

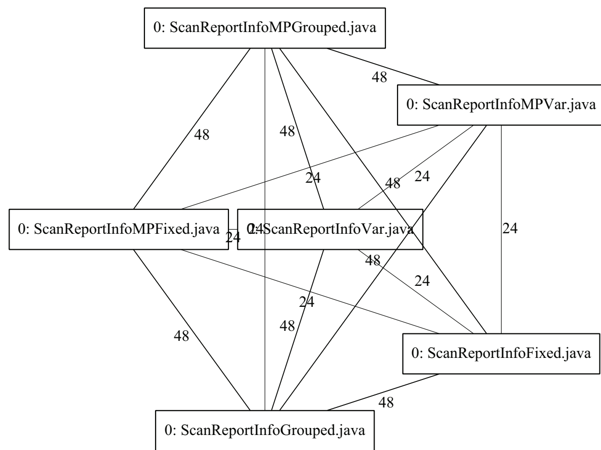
図 7(b) の差分を読んだ結果，2 種類に分類することができた．24 行の差分は，クラス名とリテラル名が変更されているほか，呼び出すクラス名が異なっていた．48 行の差分は，24 行の差分の内容のほかに，メソッド名とその返り値の型も変更されていた．

同一ソフトウェアのバージョン間の比較

対象として AlgoUML[10] の 8 つのバージョンを比較した．抽出された 2 ファイル以上を含む類似ファイル集合は 2247 あった．その中から抜き出したのが図 8 と図 9 である．各頂点のラベルの最初に付した番号がバージョン番号である．なお，全く変更のないファイル同士は派生関係では同一の頂点にまとめてある．派生関係の始点となる頂点には二重線で囲って示した．

図 8(b) を見てわかるとおり，ファイル CrUML.java はバージョン 0 から 7 まで 4 回の変更を受けており，その順

© 2012 Information Processing Society of Japan


(a) 類似ファイル集合の関係

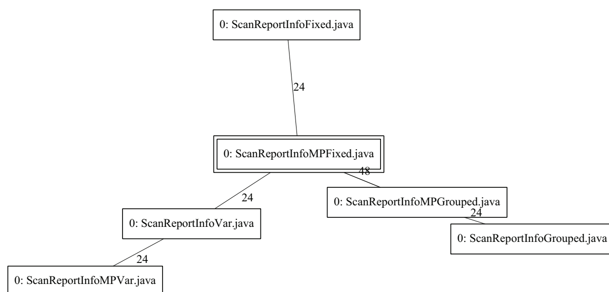

(b) 抽出した派生関係

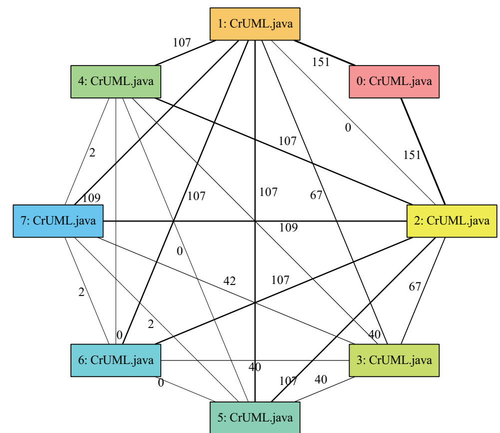
図 7 OpenHealthManager 内の類似ファイル集合
Fig. 7 Set of similar files in OpenHealthManager

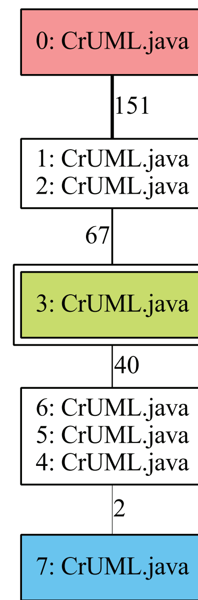
番通りに派生関係が抽出できている．

一方で，図 9(b) の例では，バージョン番号と順序が一致しない個所が見られた．バージョン 2 に向かう辺がバージョン 1 ではなくバージョン 0 から，バージョン 3 に向かう辺がバージョン 2 ではなくバージョン 0 から出ていることが確認できる．このファイル集合を確認したところ，これらのファイルはインターフェイスであった．差分を調査すると，バージョン 0 に「getAssocationRole」というメソッド宣言が存在し，バージョン 1 で「getAssociationRole」というメソッド宣言が追加されていた．その後バージョン 2 では「getAssocationRole」が削除された．引数の部分は 2 つのメソッド宣言で差がなかったことから，バージョン 1 からバージョン 2 の「メソッド宣言の削除」よりもバージョン 0 からバージョン 2 の「メソッド名の変更」のほうが差分が少ないという結果になった．バージョン 2 とバージョン 3 に関しても，一旦追加されたメソッド宣言が削除されたためにバージョン 0 からの差分の方が小さなものとなっていた．

4.2 考察

F-05D のソースコードにおいては 1 つのファイルに同様の変更が施されていたため差分の大きさが同じ辺が複数生じていた．重みの同じ片が存在すると最小全域木は複数の形を取りうる．同一カ所への複数パターンの変更は，まとめておくなどの工夫があると比較が容易になると考えられる．


(a) 類似ファイル集合の関係



(b) 抽出した派生関係

図 8 AlgoUML:履歴復元に成功した例
Fig. 8 AlgoUML: an example that maches the version history

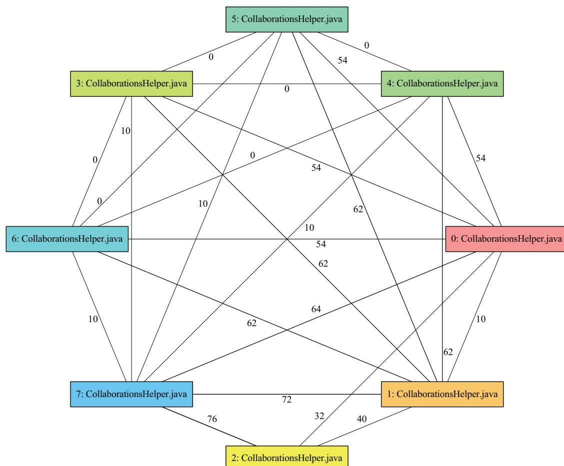
AlgoUML のソースコードにおいて，バージョンの順番と抽出した派生関係の形が一致しない例が見られた．あるバージョンで追加された要素が次のバージョンで削除される「手戻り」は最小全域木の形に影響を与えることがわかる．変更箇所が手戻りだけならば，バージョン履歴とは一致せずともソースコードの変化としては自然と考えられる．しかし，同時に他の場所で機能追加が行われていた場合など，バージョン間で複数の変更があった際には手戻りによる差分の行数への影響は，ソースファイルの派生関係が正確に抽出できなくなる要因となる恐れがある．

5. 関連研究

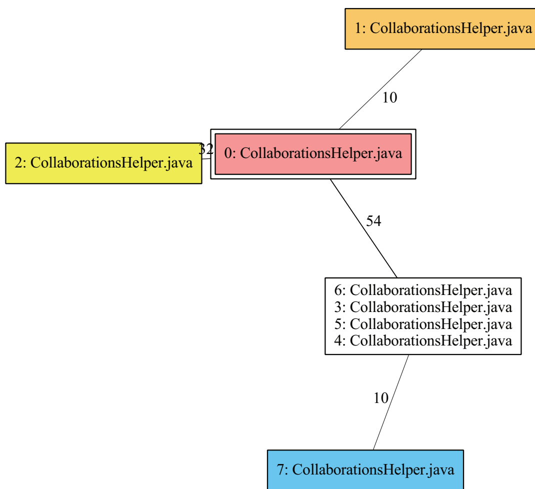
5.1 コードクローン検出技術

ソースコードの類似性に関する技術の一つに，コードク

© 2012 Information Processing Society of Japan



(a) 類似ファイル集合の関係



(b) 抽出した派生関係

図 9 AlgoUML:履歴復元に失敗した例
Fig. 9 AlgoUML: an example that unmaches the version history

ローン検出技術がある．コードクローンとは，ソースコード中に存在する互いに一致・類似したコード片のことであり，これまでに様々なコードクローン検出手法が研究されている [11][12]．

コードクローン検出に関する既存研究では，複数のソフトウェア集合の解析を想定した手法も提案されており，その中にはファイル単位やメソッド単位など粒度の大きなクローンも含まれていることが既存研究で報告されている．佐々木らはファイル単位でのクローン検出を行い，FreeBSD Ports Collection の中に約 68%のファイルクローンを検出した [13]．また，石原らは大規模ソフトウェア群からメソッド単位のコードクローンを検出し，ファイルクローンでないファイル間でもメソッドのクローンが存在することを確認した [14]．これらの研究により，派生関係にあるソフトウェア間ではファイルやメソッドが多く流用されていることが明らかになっている．つまり，コードクローン検出技術を用いてソフトウェア間で類似するファイルを特定したといえる．

コードクローン検出技術ではソースコード中の類似する部分に着目しているが，本研究ではソフトウェアがどのように進化したかを把握するためにコードクローン検出技術で得られる“どの部分が一致・類似しているのか”という情報ではなく，“どの部分が変更されたのか”という情報に注目している．

5.2 リポジトリマイニング

バージョン管理システムは，ソフトウェア開発においてソースファイルの変更履歴を保存し管理するために利用される．大規模なソフトウェアリポジトリに対して，バージョン管理システムにより得られる情報を活用した解析は多く行われている．Śliwerski らは，CVS のリポジトリを利用してバグを含む変更はどのような時に多いかを調査した [15]．Cubranic らはバージョン管理システムのほかにバグトラッキングデータベース，メーリングリストを組み合わせ，開発に関する情報を抽出した [16]．リポジトリに対するこれらの調査は，バージョン管理システムから様々な情報を得て解析を行っている．本研究では，これらの情報なしに，ソースコードのみを用いて解析を行っている．

6. 今後の課題

今回の手法の実装は簡易的なものであり，また対象としたソフトウェア集合の規模も大きくない．より大きな，分岐を含むソフトウェア集合に対して実験を行うことが主な課題となる．その他の手法の改善点を挙げる．

6.1 ディレクトリ・アプリケーション単位での比較

今回はファイル単位での比較を行ったが，実際のアプリケーションを比較する際には，ディレクトリ単位やアプリケーション単位など，粒度を大きくしてソフトウェア自体の派生関係を把握出来るようにしたい．その際，ファイル単位の派生関係とソフトウェアの派生関係で，出力されるグラフが異なる形になることも考えられる．個々のソースファイルの進化と，ソフトウェアの進化の間にどのような関係があるかを調査することが今後の課題となる．

6.2 識別子名の正規化

Bellon らは，コードクローンの差異の程度に基づき，コードクローンを 3 つに分類している [17]．そのうち，タイプ 2 では，変数名や関数名，変数の型などが変更されているコードクローンを定義している．

今回の正規化では識別子名や型名はそのままの文字列として扱ったが，識別子名を適当な文字列に置換して正規化することで，識別子名のみの変更を無視し，処理の追加・削除に注目した解析が行える．

© 2012 Information Processing Society of Japan

6.3 類似度計算の高速化

テキストベースの差分比較手法は，コストが大きいことが指摘されている．そのため，ソースコードを直接比較せず，メトリクスや統計的手法などを用いる研究も提案されている [18]．本手法では，類似度計算はすべてのファイル間で行うが，おおよそ類似していると思われるファイル同士をグループ化するための手段であるので，類似度計算を精度は劣るが高速な別の手法で行うことで処理の高速化が期待できる．

一方，類似ファイルをグループ化した後の比較は，実際にソースコードが異なるかが重要であると考えている．この部分に関しては，提案手法のようにテキストベースでの差分抽出を用いて，実際の差分に基づいた派生関係を抽出し，また開発者が読解するコードの絶対量を減らすことが適していると考えている．

6.4 派生関係の順序

今回の手法では差分の行数のみに着目して比較の始点を定めたため，提示する比較順序は本来の派生関係の順序とは無関係であった．ファイルが作成された日付がわかっているならば，古いファイルから新しいファイルへ進化する形のグラフにすることで，より実態に近い派生関係が得られると期待できる．

また，ソフトウェアの変更は機能削除よりも機能追加・修正の方が主であると考えられる．このことから，ソースコードに対する変更は削除よりも追加のほうが多いと仮定し，行の追加が大きく，削除が小さくなるような方向を求めることで，ソースファイルの派生関係の方向とすることも検討できる．

6.5 他言語への適用

今回は Java で記述されたプログラムを対象としたが，この手法は他の言語でも適用できると考えている．ソースコードの正規化部分に関しては，言語に応じてトークン化を行わなければならないが，トークン化が難しい場合などはコメントや空白空行のみを除去する簡易的な正規化を行うことで，少ないコストで多くの言語に応用できる．また，類似度計算や差分の計算は，テキストベースの手法であるので，ソースコードが行単位で記述されているか，行単位に分割可能であれば Java 以外の言語でも実行可能である．

謝辞

本研究は，日本学術振興会科学研究費補助金基盤研究 (A)（課題番号:21240002）および若手研究 (A)（課題番号:23680001）の助成を得た．

参考文献

[1] Inoue, K., Sasaki, Y., Xia, P. and Manabe, Y.: Where does this code come from and where does it go? – Integrated code history tracker for open source systems –, in *Proc. 34th International Conference on Software Engineering*, pp. 331–341 (2012).

[2] Pinzger, M., Gall, H., Fischer, M. and Lanza, M.: Visualizing multiple evolution metrics, in *Proc. ACM Symposium on Software Visualization 2005*, pp. 67–75 (2005).

[3] Jaafar, F.: On the analysis of evolution of software artefacts and programs, in *Proc. 34th International Conference on Software Engineering*, pp. 1563–1566 (2012).

[4] Tarvo, A., Zimmermann, T. and Czerwonka, J.: An integration resolution algorithm for mining multiple branches in version control systems, in *Proc. 27th International Conference on Software Maintenance*, pp. 402–411 (2011).

[5] Yoshimura, K. and Mibe, R.: Visualizing code clone outbreak: An industrial case study, in *Proc. 6th International Workshop on Software Clone*, pp. 96–97 (2012).

[6] Myers, E. W.: An O(ND) difference algorithm and its variations, *Algorithmica*, Vol. 1, No. 1, pp. 251–266 (1986).

[7] incava.org java-diff, `http://www.incava.org/projects/577828189`.

[8] Prim, C. R.: Shortest connection networks and some generalizations, *Bell Syst.Tech.J.*, Vol. 36, pp. 1389–1041 (1957).

[9] 携帯電話 (F-05D オープンソースソフトウェア) - FM-WORLD.NET（個人）：富士通:, `http://spf.fmworld.net/oss/oss/f-05d/`.

[10] argouml.tigris.org, `http://argouml.tigris.org/`.

[11] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: a multilinguistic token-based code clone detection system for large scale source code, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7, pp. 654–670 (2002).

[12] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Trans. Softw. Eng.*, Vol. 32, pp. 176–192 (2006).

[13] 佐々木裕介, 山本哲男, 早瀬康裕, 井上克郎：大規模ソフトウェアシステムを対象としたファイルクローンの検出, 電子情報通信学会論文誌. D, 情報・システム, Vol. 94, No. 8, pp. 1423–1433 (2011).

[14] 石原知也, 堀田圭佑, 肥後芳樹：大規模ソフトウェア群に対するメソッド単位のコードクローン検出, 電子情報通信学会技術研究報告：信学技報, Vol. 111, No. 481, pp. 31–36 (2012).

[15] Śliwerski, J., Zimmermann, T. and Zeller, A.: When do changes induce fixes?, in *Proc. 2nd International Workshop on Mining Software Repositories*, pp. 1–5 (2005).

[16] Cubranic, D. and Murphy, G.: Hipikat: recommending pertinent software development artifacts, in *Proc. 25th International Conference on Software Engineering*, pp. 408–418 (2003).

[17] Bellon, S., Koschke, R., Antoniol, G., Krinke, J. and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Softw. Eng.*, Vol. 33, pp. 577–591 (2007).

[18] Kontogiannis, K., Demori, R., Merlo, E., Galler, M. and Bernstein, M.: Pattern Matching for Clone and Concept Detection, *Autom. Softw. Eng.*, Vol. 3, No. 1/2, pp. 77–108 (1996).

© 2012 Information Processing Society of Japan