社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

抽象構文木を利用したファイル間のコード移動検出

藤本　章良† 　肥後　芳樹† 　松本淳之介† 　楠本　真二†

† 大阪大学大学院情報科学研究科
〒 565-0871 大阪府吹田市山田丘 1-5
E-mail: †{a-fujimt,higo,j-matumt,kusumoto}@ist.osaka-u.ac.jp

あらまし　ソフトウェア開発において，開発者がソースコードの差分を理解することは重要である．ソースコードの差分を検出するツールとして GumTree がある．GumTree は編集前後のソースコードを入力として与えると，内部で抽象構文木を生成し，削除・挿入・移動・更新といった操作を抽象構文木のノード単位で検出する．しかし，GumTree は 1 つのファイルの差分しか計算できないため，ファイルをまたいだソースコードの移動を検出できないという問題点がある．そこで本研究では，プロジェクトに含まれる全てのソースファイルから 1 つの抽象構文木を構築し，ファイルをまたいだソースコードの移動を検出する手法を提案する．そしてオープンソースソフトウェアに対して提案手法を用いて実験を行った結果，全てのプロジェクトにおいてファイルをまたいだソースコードの移動を検出でき，最大で 2,406,459 個のファイルをまたいだ移動を確認できた．また，検出結果を確認したところ，ファイルをまたいだソースコードの移動が発生する際，いくつかの特徴が得られた．

キーワード　差分，GumTree，抽象構文木

1. はじめに

ソフトウェア開発において，バージョン管理システムの使用は必須といえる．バージョン管理システムを用いた開発において，バージョン間のソースコードの差分を正確に，理解しやすい形で開発者に示すことは重要である．開発者がソースコードの差分を理解することで，ソースコードの振る舞いの変更を把握しやすくなるからである．例えば，ある変更の後にバグが発生した場合，差分を確認することで，そのバグの原因となるソースコードの特定が容易となる．そのため，ソースコードの差分を表示するツールが開発されており，Git をはじめとする多くのバージョン管理ツールには，`diff` コマンドが組み込まれている．

しかし，`diff` コマンドには 2 つの問題点が存在する．1 つ目の問題点は，出力される差分の粒度が粗いという点である．`diff` コマンドは行単位でソースコードを比較するため，ある行の一部分のみが変更された場合でも，その行全体が変更されたとして出力する．また，いくつかの行を `if` 文や `for` 文で囲んだブロック内に出し入れした場合，そのブロック全体が変更されたと出力する．このため，実際に変更が行われた箇所の把握が困難になる．`diff` コマンドの出力結果に基づいて，より詳細に変更箇所をハイライトするツールも存在するが，後者の例ではハイライトが行われない．2 つ目の問題点は，編集操作が削除と挿入の 2 種類しか存在していない点である．すべての編集操作が削除または挿入の操作で出力されるため，それ以外の操作を行った場合，開発者が意図した差分が出力されない可能性がある．

これらの問題点を解決するためのツールとして，GumTree [1] が提案されている．GumTree は，バージョンが異なる 2 つのソースファイルを入力として与えると，抽象構文木（以下，AST）を生成し，それらを比較することにより，AST のノード単位の差分を出力するツールである．AST のノードはソースコードのトークンとほぼ一致する．したがって，AST を利用することで，`diff` コマンドに比べ，より適切な範囲で差分を出力可能になる．さらにノードの削除や挿入以外にも，更新や移動を検出できる．

GumTree 及び GumTree を改良したツールが出力した編集スクリプトは多くの研究で利用されている．例えば，Maven のビルドファイルの解析 [2] やバグ修正パターン検出の自動化 [3]，バージョン管理システムを利用する際のコミットメッセージの生成 [4]，API のコードのサジェスト [5] などに用いられている．

しかし，GumTree にも問題点が存在する．1 つのファイルの差分しか計算できないため，ファイルをまたいだソースコードの移動を検出できない点である．ファイルをまたいだソースコードの移動は，リファクタリングにおいて頻繁に行われる．このような「移動」と出力されるべき変更が，「削除と挿入」として出力されてしまい，その結果，開発者がソースコードの変更に対して誤った認識を持つ可能性がある．

そこで本研究は，1 つのファイルの差分しか計算できない GumTree を拡張して，プロジェクト全体の差分を計算し，ファイルをまたいだソースコードの移動を検出できるようにすることを目的とする．複数のソースファイルからそれぞれ AST を生成し，それらをまとめた 1 つの AST を構築する．変更前後のプロジェクト全体の AST を比較することで，ファイルをまたいだソースコードの移動を検出可能にする．

提案手法を 8 個のオープンソースソフトウェアに適用し，ソースファイルをまたいだ移動を検出できることを確認した．またそれらが誤検出でないかどうかの検証を行った．さらに，提案手法によって検出された，ファイルをまたいだソースコードの移動を目視で確認し，ソースコード，及びファイル名といったファイルの情報にどのような特徴があるかを分析した．

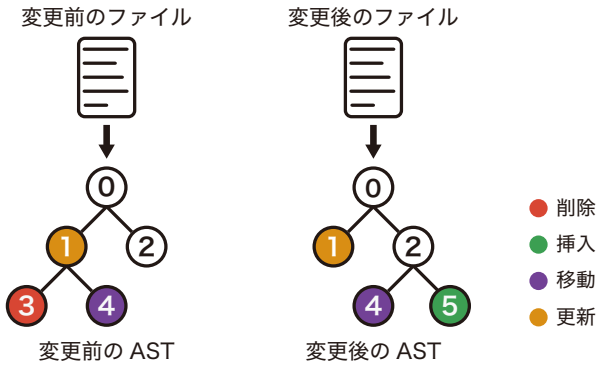

図 1　GumTree の差分検出

2. 準　　備

2.1 抽象構文木（AST）

AST は，ソースコードを構文解析して得られる木構造のデータである．AST の各ノードは以下の要素から構成されている．

親ノード：　AST の各ノードは，木構造上の親ノードへの参照を持つ．ただし，根ノードの親は存在しないので何も保持しない．

ラベル：　if 文や変数宣言といった文法上の型を表す．

値：　メソッド名や変数名などのラベル以外の情報である．

2.2 GumTree の差分検出

GumTree は入力として，変更前後のソースファイルを受け取り，AST のノード単位の編集スクリプトを出力する．編集スクリプトとは，変更後のソースコードを得るために変更前のソースコードに適用された編集操作の列である．GumTree は，削除・挿入・移動・更新のうち行われた操作と，その操作が行われた AST のノードの情報を，編集スクリプトとして出力する．

GumTree は入力された変更前後のソースファイルから，それぞれの AST を生成する．この 2 つの木の違いを差分として出力する．木構造の差分を計算するために，GumTree はマッチングを行う．マッチングとは，変更前後の AST のノードの対応付けを行う処理である．対応付けられたノードは，変更の前後で同じノードとして扱われる．マッチングの結果と AST を参照し，変更前の AST に対して，どのノードが削除・挿入されたか，どのノードの位置が変更されたか，どのノードの値が更新されたか，どのノードは変更が行われていないか，といった情報を得る．

図 1 は，GumTree の差分検出方法を模式的に表している．変更前後のファイルから生成した AST にマッチングを行い，ノードの対応付けがされている．図 1 では，マッチングの結果を数字で表している．この例では，3 番のノードは変更前にしかないため削除，5 番のノードは変更後にしかないため挿入，と編集スクリプトに出力される．4 番のノードは変更前後のどちらにも存在しているが，親のノードが変わったため移動と出力される．1 番のノードは，更新と出力されている．更新は，変更の前後に同じノードが存在し，その値が変更された場合に出力される．

3. 研 究 目 的

GumTree は，1 つのファイルの変更について，どのような操作が行われたかを検出できる．しかし，一度に複数のファイルに対して変更が行われることがあり，そのような場合，ファイルをまたいだソースコードの移動を検出することはできない．

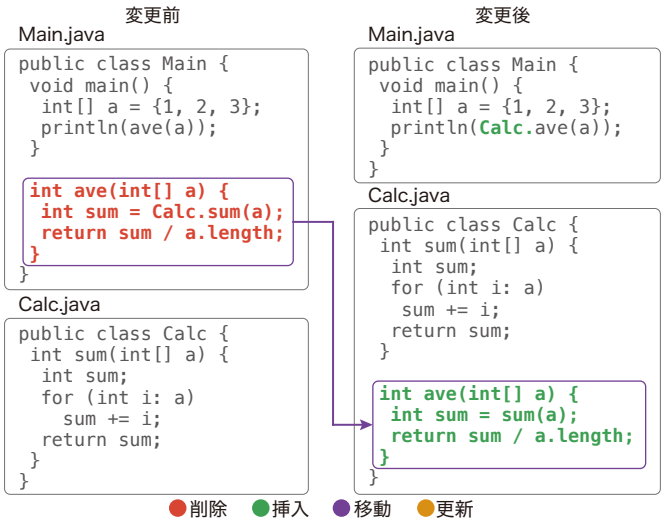

図 2　実際の編集操作と GumTree の出力が異なる例

図 2 の例では，ファイル `Main.java` と `Calc.java` が同時に変更されたと仮定する．この変更では，ファイル `Main.java` に存在したメソッド `ave()` が `Calc.java` に移動されている．この変更の前後で，GumTree を用いてそれぞれのファイルの差分を出力すると，`Main.java` の編集スクリプトはメソッド `ave()` の削除，`Calc.java` の編集スクリプトはメソッド `ave()` の挿入となる．しかし，実際に行われた操作を表現するには，メソッド `ave()` の移動の方が適切である．この違いにより，開発者がソースコードの変更について誤った認識を持つおそれがある．

開発者に差分を表示する際，複数のファイルの変更を考慮することで，より適切な差分を出力できると著者らは考えた．そこで，本研究では複数のファイルが変更された際に，ファイルをまたいだ移動を検出する手法を提案する．ファイルをまたいだ移動を検出することによって，開発者がソースコードの変更をより正しく理解できるようになると著者らは考えた．

4. 提 案 手 法

4.1 概　　要

提案手法の概要を図 3 に示す．提案手法の入力は，変更前後のプロジェクトに含まれる全てのソースファイルであり，出力は編集スクリプトである．まず，入力として与えられたソースファイルから，それぞれに対応する AST を生成する．それらの AST を用いて，プロジェクト全体の AST を構築する．このプロジェクト全体の AST は，変更前と変更後で 1 つずつ作られる．プロジェクト全体の AST の構築方法は，4.2 節で述べる．

次に，構築したプロジェクト全体の AST に対してマッチングを行い，差分を計算する．プロジェクト全体の AST の差分計算には，GumTree の差分を計算する処理を再利用した．そして，GumTree の手法を用いて得られた編集スクリプトを出力する．

しかし，単純にプロジェクト内の全てのソースファイルから 1 つの AST を構築し差分を計算すると，計算に多大な時間を要したり，移動を検出する精度が下がったりする可能性がある．そこで，4.3，4.4 節のような工夫を加え，これらの問題を回避する．

4.2 プロジェクト全体の AST の構築

プロジェクト全体の AST の構築を図 4 に示す．まず，プロジェクト全体の AST を構築するため，ノードを 1 つ作成する．

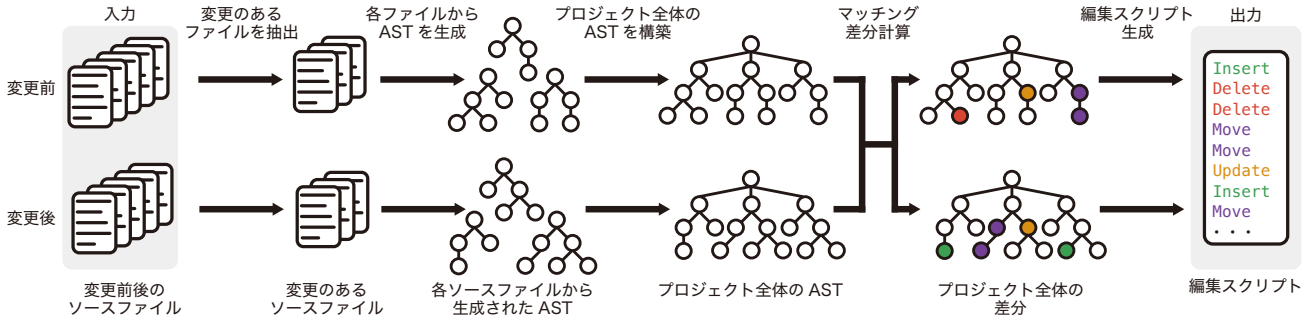

図 3　提案手法の概要

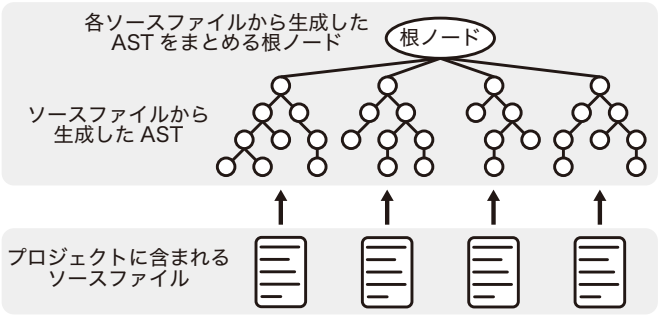

図 4　プロジェクト全体の AST の構築

このノードを根ノードとし，ソースファイルから生成した AST を子ノードとして加えていく．生成した AST を子ノードとして加える順番は，ソースファイル名のアルファベット順である．

4.3　変更のないファイルの除外

プロジェクトに含まれる全てのソースファイルからプロジェクト全体の AST を構築すると，プロジェクト全体の AST のノード数が増え，計算時間が長くなってしまう．また，変更されていないファイルから誤って変更があると検出される可能性がある．そこでプロジェクト全体の AST を構築する段階で，ソースファイルごとに変更の有無を確認する．変更があるソースファイルから生成した AST だけをプロジェクト全体の AST に加える．

ファイル内容の変更の有無は Myers のアルゴリズム [6] を用いて調べる．得られた結果が，

- 全く変更がない
- 行・スペース・タブの削除，挿入のみ

のどちらかに該当する場合，そのファイルから生成される AST は，プロジェクト全体の AST には加えない．

4.4　2 段階のマッチング

差分の検出対象を 1 つのファイルからプロジェクト全体に拡大した際，変更されていないにも関わらず，類似したソースコード間で移動と出力される場合がある．これは，AST がプロジェクト全体に拡大される事により，GumTree がノードを適切にマッチングできていないことが原因である．誤検出の例を図 5 に示す．この例では，変更の前後でメソッド `foo()`，`bar()` に含まれる `int num = 1;` は変更されていない．しかし `A.java` の `int num = 1;` が `B.java` の `int num = 1;` に不適切にマッチングされた結果，`A.java` から `B.java` に移動したと誤検出している．

適切にマッチングできない理由は，マッチング対象がプロジェクト全体に拡大されて候補が増えるため，適切なノードを探し出せないからである．そこで適切にマッチングするために，2 段階のマッチングを行う．2 段階のマッチングの方法を図 6 に示す．まず，プロジェクト全体の AST の中から，同じ名前を持つファイルの部分木を取り出して，マッチングを行う．その後，プロジェクト全体の AST についてマッチングを行う．

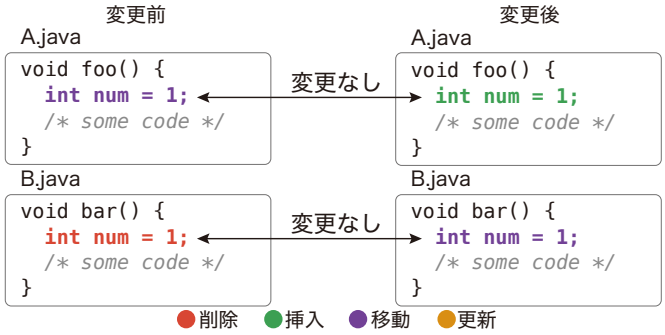

図 5　移動していないのに移動と誤って検出した例

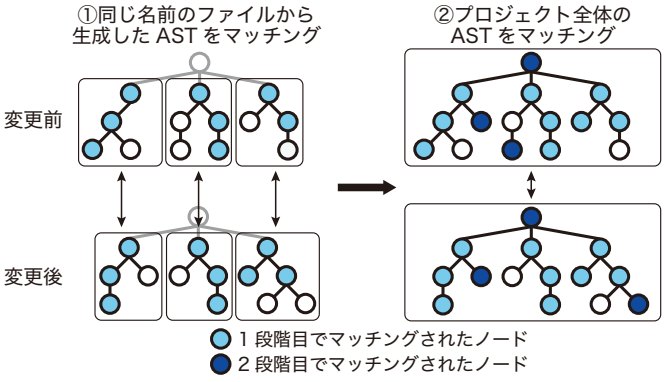

図 6　2 段階のマッチング

ファイルをまたいだ変更と比べて，ファイルをまたがない変更の方が多いと考えられる．そこで，同一ファイルの変更前後での AST のマッチングを先に行う．これは AST ノードのマッチング候補を同一ファイル内のみにすることで，マッチング候補の数を減らし，適切にマッチングを行うことが狙いである．さらに，2 段階目のプロジェクト全体のマッチングでは，1 段階目でまだマッチングされていないノードの中からマッチングを行い，マッチングの候補を限定する．これによって，類似したソースコードが移動と誤検出される可能性を下げることができる．

図 5 の変更に対し，2 段階のマッチングを適用すると，1 段階目の同一ファイルのマッチング時に，変更前後の `A.java` に含まれる `int num = 1;` がマッチングされる．同様に，変更前後の `B.java` に含まれる `int num = 1;` がマッチングされる．この結果，`int num = 1;` に対する変更は行われていないと出力される．

5.　実　　験

提案手法を用いて，オープンソースソフトウェア（以下 OSS）に対して実験を行った．実験の目的は次の 2 つである．

WorkbenchLayout.java（変更前）

```
private List getTrimContents(String areaId) {
  List trimContents = new ArrayList();
  Control[] children = layoutComposite.getChildren();
  for (int i = 0; i < children.length; i++) {
    // Skip any disposed or invisible widgets
    if (children[i].isDisposed() || !children[i].getVisible()) {
      continue;
    }

    // Only accept children that want to be layed out in a particular
    // trim area
    if (children[i].getLayoutData() instanceof TrimLayoutData) {
      TrimLayoutData tlData = (TrimLayoutData) children[i]
          .getLayoutData();
      if (tlData.areaId.equals(areaId)) {
        trimContents.add(children[i]);
      }
    }
  }
  return trimContents;
}
```

WorkbenchStatusDialogManager.java（変更後）

```
private boolean shouldAccept(StatusAdapter statusAdapter, int mask) {
  IStatus status = statusAdapter.getStatus();
  IStatus[] children = status.getChildren();
  if (children == null || children.length == 0) {
    return status.matches(mask) || (handleOKStatuses && status.isOK());
  }
  for (int i = 0; i < children.length; i++) {
    if (children[i].matches(mask)) {
      return true;
    }
  }
  if (handleOKStatuses && status.isOK()) {
    return true;
  }
  return false;
}
```

図 7　移動元と移動先のソースコードが大きく異なる例

1. ファイルをまたいだソースコードの移動を検出できるか
2. ファイルをまたいだ移動にはどのような特徴があるか

以下では，この実験の内容について説明する．

5.1 実験対象

GumTree [1] の評価実験に用いられた CVS-Vintage [7] に含まれている OSS のうち，Git に移行されているソフトウェアを実験対象とした．実験対象の OSS を表 1 に示す．これらのプロジェクトは全て Java で開発されている．

5.2 実験方法

本研究では 2 種類の実験を行った．1 つ目が，ファイルをまたいだ移動を検出できているかの確認，2 つ目が，ファイルをまたいだ移動にはどのような特徴があるかを確認するための実験である．以下では，この 2 つの実験方法について説明する．

5.2.1 ファイルをまたいだ移動の検出の確認

バージョン管理されている OSS に対し，全てのコミット前後でのソースファイルを，提案手法への入力として与える．検出した差分のうち，ファイルをまたいだ移動の数を計測する．

次に，ファイルをまたいだ移動であると正しく検出できているか評価するため，検出されたファイルをまたいだ移動の中からサンプリングして目視確認を行う．図 7 は Eclipse に対して提案手法を実行した際に，移動元と移動先のソースコードが大きく異なっているにも関わらず，ファイルをまたいだ移動であると出力された例である．原因として，検出対象を拡大したことにより，不適切なマッチングしてしまうことが考えられる．移動元と移動先のソースコードが異なっていた場合，提案手法がファイルをまたいだソースコードを適切に検出できるとはいえない．提案手法が有効であることを確認するため，移動先に同様のソースコードが存在した場合，正しく移動を検出したと判断する．

目視確認をする際には，移動するノードを限定して確認する．限定するノードの種類は，移動と検出された AST ノードの部分木に含まれるノード数（以下，部分木の大きさ）によって決定した．ファイルをまたいだ移動を分析すると，移動する AST ノードのラベルの種類とその平均の部分木の大きさには傾向があることが分かった．予備実験として，5.1 節に含まれる OSS のうち，ArgoUML と Log4j，Tomcat に対して提案手法を実行した際の，ファイルをまたいで移動したノードの種類ごとの平均の大きさを表 2 に示す．ただし，ファイルをまたいで移動したノードは種類が多いため，一部のノードを抜粋して掲載する．

目視確認では，平均的な部分木の大きさが大きいノードに関して確認を行う．移動する部分木の大きさが大きいと，移動するソースコードの量も多く，どのような機能が移動したかを開発者が理解しやすくなると考えられる．ゆえに，部分木の大きさが大きい移動を正しく検出できるかを評価することが重要となる．

移動する部分木の大きさが大きいノードを，有用な移動ノードと定義する．本研究では，有用な移動ノードとして MethodDeclaration，WhileStatement，ForStatement，IfStatement，Block を選択した．目視確認では，ファイルをまたいで移動したこれらのノードに対し，それぞれ最大 20 個をランダムに選び確認する．

5.2.2 ファイルをまたいだ移動の特徴の確認

ファイルをまたいだソースコードの移動の特徴を調査する．どのようなソースコードが移動しているか，変更の前後でのファイル内容の違い，移動前後のファイル名の特徴，などを調べる．

5.3 実験結果

5.3.1 ファイルをまたいだ移動の検出の確認

各 OSS とコミット数，ファイルをまたいだソースコードの移動の検出数，検出された全ての移動数を表 3 に示す．ファイルをまたいだ移動の数の横に示されている括弧は，検出された全ての移動に対するファイルをまたいだ移動の割合である．全てのプロジェクトで，ファイルをまたいだ移動が含まれることが分かる．

表 1　実験対象の OSS プロジェクト

OSS 名	コミット数	実験対象の最終コミット日
ArgoUML	16,144	2015 年 1 月 11 日
dnsjava	1,771	2019 年 10 月 27 日
Eclipse(注1)	29,811	2019 年 11 月 17 日
JHotDraw	763	2018 年 8 月 27 日
JUnit 4	2,418	2019 年 11 月 2 日
Apache Log4j 2	10,752	2019 年 11 月 1 日
Apache Struts	5,697	2019 年 11 月 4 日
Apache Tomcat	21,492	2019 年 11 月 6 日

(注1)：org.eclipse.ui.workbench のみ調査した．Eclipse はプロジェクトの規模が非常に大きく，CVS-Vintage では eclipse.ui.workbench と eclipse.jdt.core のみ調査している．eclipse.jdt.core に対しても実験を行ったが，実行時間が長く評価できなかったため，実験対象から除外した．

表 2　移動したノードのラベルごとの平均の大きさ（一部抜粋）

ノードのラベル	ArgoUML	Apache Log4j	Apache Tomcat
MethodDeclaration	85.20	67.41	57.82
WhileStatement	159.08	61.75	74.63
ForStatement	104.27	40.66	58.21
IfStatement	117.97	23.83	30.65
Block	53.25	50.17	39.86
ImportDeclaration	2.00	2.00	2.00
FieldDeclaration	10.23	9.73	10.28
Assignment	6.95	7.76	6.38
SimpleName	1.00	1.00	1.00
全体	10.26	8.55	7.69

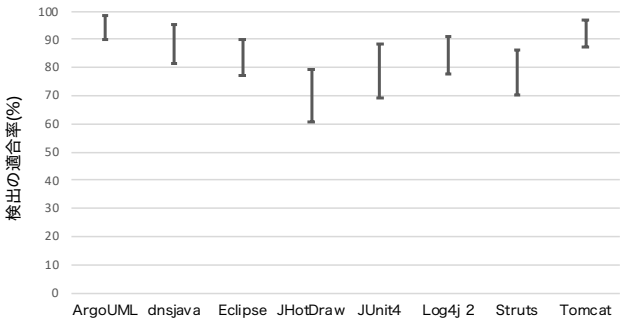

図 8　ファイルをまたいだ移動の検出の適合率

次に，ファイルをまたいだ有用な移動ノードがどの程度検出できたかを表 4 に示す．有用な移動ノードもファイルをまたいで移動していることが分かった．その数に注目すると，MethodDeclaration，Block といったノードが移動する回数が多く，IfStatement が続く．WhileStatement や ForStatement の移動の数は比較的少ない．また Eclipse を除いて，WhileStatement と ForStatement の移動の数に大きな差がないことが分かった．

プロジェクトごとに，ファイルをまたいだ移動の検出精度の適合率を二項分布に従うと仮定し，これを 95%信頼区間で表したのが図 8 である．その結果，半数以上のプロジェクトで 7 割以上の適合率で検出できていることが分かった．

5.3.2　ファイルをまたいだ移動の特徴

検出されたファイルをまたいだ移動の例を説明する．各ソースコードの例の紫色で示した部分が移動と検出された箇所である．

機能の分割・統合

大きいクラスが分割されて複数のクラスに分けられる時に，ファイルをまたいだメソッドの移動が検出された．その例を図 9 に示す．変更前は `Main` クラスに含まれていたメソッド `read()` と `write()` が `IO` クラスに移動している．この他にも，似た機能を持つクラスを統合するための移動もあった．

移動先のクラスが新たに作られる場合と，既存のクラスにソースコードが移動される場合を確認した．移動前後のファイル名に，Util，Helper といった名称が付く例が合計で 29 個あった．またそれら以外は，元の名前と似た名称になることが多かった．

表 3　OSS での実験結果

OSS 名	ファイルをまたいだ移動の数	全ての移動の数
ArgoUML	9,645(10.6%)	91,005
dnsjava	1,591(0.488%)	326,041
Eclipse	2,406,459(50.3%)	2,382,478
JHotDraw	5,477(0.639%)	847,494
JUnit 4	15,160(0.14%)	102,043
Apache Log4j 2	22,191(9.57%)	209,618
Apache Struts	20,027(11.3%)	176,458
Apache Tomcat	51,932(1.55%)	335,490

表 4　有用な移動ノードの検出数

OSS 名	Method Declaration	While Statement	ForSta-tement	IfState-ment	Block
ArgoUML	184	13	11	143	285
dnsjava	91	0	1	25	18
Eclipse	49,912	18	308	12,800	79,038
JHotDraw	184	1	6	54	149
JUnit 4	373	1	2	9	219
Log4j 2	740	8	9	96	583
Struts	459	8	6	187	527
Tomcat	1,040	16	14	1,113	2,448

変更前

Main.java

```
class Main {
  void read() {/* */}
  void write() {/* */}
  void prepare() {/* */}
  void execute() {/* */}
}
```

変更後

IO.java

```
class IO {
  void read() {/* */}
  void write() {/* */}
}
```

Main.java

```
class Main {
  void prepare() {/* */}
  void execute() {/* */}
}
```

図 9　機能の分割のソースコードの例

変更前

Main.java

```
void method() {
  while (!isDone()) {
    /* some codes */
  }
  doSomething();
}
```

変更後

Sub.java

```
void extraction() {
  while (!isDone()) {
    /* some codes */
  }
}
```

Main.java

```
void method() {
  extraction();
  doSomething();
}
```

図 10　メソッドへの切り出しのソースコードの例

変更前

C.java

```
class C {
  void methodA() {/* */}
  void methodB() {/* */}
}
```

変更後

A.java

```
abstract class A {
  void methodA() {/* */}
}
```

C.java

```
class C extends A {
  void methodB() {/* */}
}
```

図 11　継承関係の変化のソースコードの例

メソッドの移動以外にも，図 10 のような一部の処理のみを取り出してメソッド化するような変更があった．WhileStatement，ForStatement，IfStatement といったノードでこのような移動が見られた．メソッドの行数が長く，処理を別のメソッドに切り出すリファクタリングをした際に，このような移動が検出された．

継承関係の変化

クラス間の継承関係が変化するような場合に，ファイルをまたいだ移動を検出した．図 11 は，新たに作成した抽象クラスに一部のソースコードが移動される例である．このような，具象クラスから抽象クラスへのソースコードの移動が多く検出された．反対に，抽象クラスから具象クラスへの移動も存在した．

その他にも，継承関係のある具象クラス間で，実装箇所の変更によるソースコードの移動も検出した．

短いソースコードの移動

ファイルをまたいだ移動をするソースコードの中には，行数が短いソースコードも多く存在した．次のようなソースコードが挙げられる．

- getter/setter（MethodDeclaration）
- return 文（Block）
- null チェック（IfStatement）

getter や setter の移動は，抽象クラスを追加した際によく見

られた．特に Tomcat では具象クラスから抽象クラスへの 13 個の移動うち，6 個が getter または setter であった．

return 文はブロックの中に `return;` のように return 文のみが含まれるようなソースコードの移動である．このような移動はほとんどのプロジェクトで検出された．

IfStatement で null チェックを行うソースコードの移動も検出された．この `if` 文内では，新規インスタンスの作成のみや return 文など短いソースコードがほとんどである．また，この移動は Eclipse で非常に多く見られたが，他のプロジェクトでは確認を行ったサンプルの中にほとんど含まれなかった．

6. 考　　察

6.1 ファイルをまたいだ移動の検出

実験結果より，提案手法を用いてファイルをまたいだソースコードの移動を検出することが可能になることが分かった．検出した有用な移動ノードのうち，表 4 で示すように WhileStatement や ForStatement が移動した回数は少ない．表 5 は最新版の ArgoUML，Log4j，Tomcat から生成した AST から，IfStatement，WhileStatement，ForStatement の数を計測した結果である．WhileStatement や ForStatement は IfStatement に比べて出現数が少なく，ファイルをまたいだ移動の検出数も少ない．

また，プロジェクト全体に検出範囲を拡大したが，適合率は最も低いプロジェクトでも 60%，半数以上のプロジェクトで 70% 以上となっており，ファイルをまたいだソースコードの移動の検出に対して，提案手法は有効であるといえる．

6.2 ファイルをまたいだ移動の特徴

検出できたファイルをまたいだ移動のうち，機能の分割・統合，継承関係の変化の共通点として，これらの操作はリファクタリングによって多く引き起こされると考えられる．同様の機能を抽出したり，肥大化した機能を分割したりする操作はリファクタリングにおいて多く行われる [8]．この操作に伴う移動が，提案手法によって検出できることが確認できた．

また，表 4 からは，MethodDeclaration が WhileStatement に比べて多いことがわかる．メソッドが 1 つの機能を持っていることが多く，ある機能を持つソースコードを他のファイルに移す際は `while` 文や `for` 文単体で移動させるよりもメソッドの移動が中心になると考えられる．そのため，表 4 のように，MethodDeclaration の移動の数が多くなっていると考えられる．目視確認を行なった際には，`while` 文や `for` 文を含んだメソッドが，ファイルをまたいで移動していることが確認できた．

5.3.2 項で挙げた，短いソースコードの移動が多く検出された理由として，これらのソースコードはプログラム内に多く存在するということが考えられる．それらをマッチングした際，ファイルをまたいだノード同士のマッチングが多くなったと考えられる．さらに，ソースコードの長さも影響がある．GumTree は部分木の類似度を用いて，閾値が一定値を超えた場合にノードをマッチングする．短いソースコードであれば，ソースコードの差異が出にくく類似度が高くなりやすい．そのため，よりマッチングされやすくなり，検出される可能性が高くなると考えられる．

表 5　プロジェクトに含まれる `if`，`while`，`for` 文に対応するノードの数

ノードのラベル	ArgoUML	Log4j	Tomcat
IfStatement	12,426	5,080	23,521
WhileStatement	512	199	626
ForStatement	424	626	1,556

7. 妥当性の脅威

提案手法は特定のプログラミング言語に依存しないため，GumTree を適用可能なプログラミング言語すべてにおいて適用可能である．本研究では，Java で記述されたプロジェクトに対してのみ実験を行った．他の言語でもファイルをまたいだソースコードの移動を検出できると予想されるが，実際に検出できるかは分からない．

8. おわりに

本研究では，プロジェクトに含まれるソースファイルから，プロジェクト全体の AST を構築し差分を計算することで，ファイルをまたいだソースコードの移動を検出する手法を提案した．提案手法を用いて 8 個の OSS に対して実験を行ったところ，すべてのプロジェクトにおいてファイルをまたいだソースコードの移動を検出できた．今後の課題としては以下が考えられる．

他の言語で開発された OSS への適用：　Java 以外の言語で開発されている OSS に対して提案手法を適用し，ファイルをまたいだソースコードの移動を検出できるかを調査する．

検出した差分の可視化：　検出されたファイルをまたいだソースコードの移動を，より分かりやすい形で開発者に表示する．GumTree には，Web ブラウザ上で編集スクリプトを視覚的に確認できるツールがあり，これを改良することが考えられる．

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 (B)（課題番号：17H01725）の助成を得て行われた．

文　　献

[1] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” International Conference on Automated Software Engineering, Vasteras, Sweden - 19, 2014, pp.313–324, 2014.

[2] C. Macho, S. McIntosh, and M. Pinzger, “Extracting build changes with builddiff,” The 14th International Conference on Mining Software Repositories, pp.368–378, 2017.

[3] F. Madeiral, T. Durieux, V. Sobreira, and M. Maia, “Towards an automated approach for bug fix pattern detection,” 2018.

[4] M.S. Ahmed and A. Tabassum, “Automatic contextual commit message generation: A two-phase conversion approach”.

[5] A.T. Nguyen, M. Hilton, M. Codoban, H.A. Nguyen, L. Mast, E. Rademacher, T.N. Nguyen, and D. Dig, “API code recommendation using statistical learning from fine-grained changes,” International Symposium on Foundations of Software EngineeringACM, pp.511–522 2016.

[6] E.W. Myers, “Ano(nd) difference algorithm and its variations,” Algorithmica, vol.1, no.1, pp.251–266, Nov. 1986.

[7] M. Monperrus and M. Martinez, “Cvs-vintage: A dataset of 14 cvs repositories of java software”.

[8] M. Fowler, Refactoring: improving the design of existing code, Addison-Wesley Professional, 2018.