情報処理学会論文誌 Vol.58 No.4 833–844 (Apr. 2017)

編集スクリプトへのコピーアンドペースト操作の導入によるコード差分の理解向上の試み

肥後 芳樹[1,a)] 大谷 明央[1,b)] 楠本 真二[1,c)]

受付日 2016年8月2日, 採録日 2017年1月10日

概要：ソフトウェアの変更内容を理解することはコードレビュー等において重要であり，これまでに数多くの変更内容の理解支援を行う手法が提案されている．そのなかでも抽象構文木を用いた手法はプログラムの構造を考慮して変更内容を表現するため，Unix の diff のような行単位の表現に比べて人間が理解しやすいとされている．本研究では，既存の抽象構文木を用いた手法を改良することにより，より理解しやすく変更を表現する手法を提案する．提案手法では，ソースコードの実装において開発者がしばしば行うコピーアンドペーストの操作を取り入れて，変更を表現する．提案手法を実装して，複数のオープンソースソフトウェアに対して適用し，その有効性が確認できた．

キーワード：コードの変更理解支援，抽象構文木の解析，コードリポジトリマイニング

Improving Understandability of Source Code Change via Introducing Copy-and-Paste Operation into Edit Script

Yoshiki Higo[1,a)] Akio Ohtani[1,b)] Shinji Kusumoto[1,c)]

Received: August 2, 2016, Accepted: January 10, 2017

Abstract: Understanding source code changes is important in some activities such as code review. A variety of methodologies and tools have been proposed to support understanding changes. In such methodologies, Abstract Syntax Tree based (in short, AST) ones are considered more helpful for understanding than text based ones such as UNIX diff. In this paper, the authors propose a new AST-based methodology, which is an enhanced version of existing methodology. More concretely, the authors introduce a notion of *copy-and-paste* operations to express differences between two versions of source code. The authors implement a prototype based on the proposed methodology and apply it to 14 open source systems. As a result, the authors confirm that the proposed methodology generates shorter (more understandable) difference expressions than an existing methodology.

Keywords: supporting code change understanding, analyzing abstract syntax trees, mining code repositories

1. はじめに

ソースコードのレビューやバージョン管理システムにおける変更の競合が発生した場合のように，ソースコードに対して行われた変更を詳しく理解する必要がある状況が存在する．そのような場面において，ソフトウェア開発プロジェクトに携わる人間を支援するために，変更の内容を分かりやすく表現する研究が数多く行われている．

ソースファイルの変更内容を表現する手法のうち広く使われるのは，テキストに基づいて変更内容を表示する手法である [3], [12], [14]．たとえば，Unix の diff では，入力としてソースファイルを 2 つ与えると，Myers のアルゴリズム [14] を適用し，行の追加や削除を示して変更内容を表現する．しかし，テキストに基づく手法はプログラムの構造を考慮できていないという弱点がある．そのため，表現された変更内容が人間にとって理解しやすいとは限らない．

[1] 大阪大学
Osaka University, Suita, Osaka 565–0810, Japan
[a)] higo@ist.osaka-u.ac.jp
[b)] a-ohtani@ist.osaka-u.ac.jp
[c)] kusumoto@ist.osaka-u.ac.jp

テキストに基づく手法の弱点を克服した手法として，抽象構文木を用いた手法がある [6], [9], [10], [15]．抽象構文木に基づく手法は，プログラムの構造を考慮して変更内容を表現するため，人間はその変更内容をより理解しやすい．抽象構文木に基づく手法は編集スクリプトを生成することによりプログラムの構造を考慮して変更内容を表現する．編集スクリプトとは，2 つの抽象構文木が与えられたとき，一方の抽象構文木をもう一方の抽象構文木に変更するための操作の手順である [6]．編集スクリプトにおいて，変更内容は抽象構文木の頂点への操作の連なりとして表現される．変更において行われる操作が多いほど，人間が変更内容を理解する負担が大きくなるため，編集スクリプトの長さを用いることで人間が変更内容を理解する際の労力を表すことができる [8]．生成される編集スクリプトが短いほど人間が変更内容を理解しやすい．

抽象構文木に基づく手法は，非常に多くの頂点を含む抽象構文木を対象にした場合や頂点の移動を考慮した場合に，編集スクリプトの生成に長い時間を要するという課題点がある．この課題点を改善するために，Falleri らは編集スクリプトの生成の過程で経験則を用いることで，効率的に移動を考慮した編集スクリプトを生成する手法を考案した [8]．また，Falleri らはオープンソースソフトウェアのリポジトリを対象に実験を行い，Falleri らの手法が実用的な速度で動作することを示した．また，生成した編集スクリプトを人間が理解しやすいことも示した．

本研究では，既存の手法よりもさらに人間にとって理解しやすく変更内容を表現することを目的として，Falleri らの手法を改良した．Falleri らの手法では挿入，削除，移動，更新を組み合わせて変更内容を表現する．しかしながら，ソースコードの実装においては，上記の操作に加えてコピーアンドペースト（ソースコードの一部をコピーして，そのコピーしたものを別の場所にペーストする操作）も頻繁に行われる [1], [11]．そのため，コード片のコピーアンドペーストが行われた場合，Falleri の手法を用いると，新しい頂点が多数挿入されたかたちで変更内容が表現される．その結果，編集スクリプトが長くなってしまう．

本研究では，コピーアンドペーストを導入した編集スクリプトの生成手法を提案する．コピーアンドペーストを導入することにより，より短い編集スクリプトを生成できるようになるため，人間にとってより理解しやすい表現になると著者らは考えた．また，本研究では，提案手法を実装し，手法の有効性を評価するための実験を行う．以下に，実験によって得られた結果をまとめる．

- 18%の変更に対して，提案手法が生成する編集スクリプトは Falleri らの手法が生成するものよりも短い．
- 残りの 82%の変更では，両者の長さは同じであった．
- 提案手法の実行時間は Falleri らの手法と比較して長くなるが，96%の変更において Falleri らの手法の 1.5 倍以内の時間で実行できる．
- 96%の変更において提案手法は 2 秒以内に編集スクリプトを生成できる．

2. 準備

2.1 抽象構文木

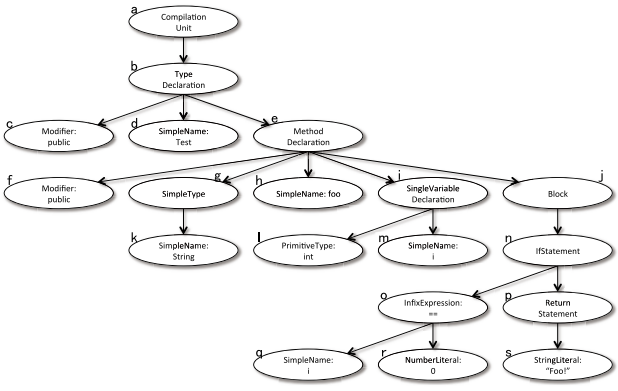
抽象構文木（Abstract Syntax Tree，以下 **AST**）とは，ソースコードの構文情報を表現した木構造である．AST は順序木であり，子頂点の数に制限はない．例として，簡単な Java ソースコードとそれに対応する AST を**図 1** に示す．この AST はプログラムの構造に対応する 19 の頂点を持つ．AST の頂点は，ID とソースコードの要素に対応するラベルとソースコード内の実際の字句に対応する値を持つ．たとえば，図 1 (b) において「NumberLiteral:0」は NumberLiteral がラベル，0 が値を表す．AST 上の頂点は構文上の 1 つの要素を表し，枝で直接結ばれた子頂点はその詳細情報を表す．図 1 (b) において，IfStatement の構文情報はその子頂点により表されており，InfixExpression および ReturnStatement であることが分かる．

2.2 コピーアンドペースト

ソフトウェア開発において，開発者はソースコードを頻繁にコピーアンドペーストする [11]．また，Ahmed らの研究により，開発者が行うコピーアンドペーストのうち，63.52%においてコピー元のファイルとコピー先のファイルが同一であることが分かっている [1]．コピーアンドペーストによって生成されたソースコードは，コピー元のソースコードとコードクローン関係になる．

```
public class Test{
 public String foo(int i){
   if(i == 0) return "Foo!";
 }
}
```

(a) ソースコード



(b) AST

図 1 AST の例
Fig. 1 An example of AST.

2.3 コードクローン

コードクローン（以下，クローン）とはソースコード中に存在する同一，あるいは類似するコード片のことである．クローンは既存ソースコードのコピーアンドペースによる再利用等，様々な理由で発生する [16], [18], [19], [20]．クローンは，その類似度に基づいて以下の 3 種類に分類されることが多い．

TYPE-1 空白，タブ，改行文字，コメント等のプログラムの振舞いに影響を与えないソースコード中の要素を除いて完全に同一のクローン．

TYPE-2 変数名や関数名等のユーザ定義名の違いやリテラルの違いのような字句単位での差異を含むクローン．

TYPE-3 字句単位よりも大きな違いを含むクローン．

これまでにさまざまな検出手法が提案されているが，クローンの定義は手法ごとに異なる．そのため，異なる手法を利用して同一ソースコードからクローンを検出した場合，その検出結果は異なる．

本研究では，類似する AST の部分木をクローンと呼ぶ．本研究における TYPE-1 クローンとは構造が同じであり頂点の値も同じである部分木，TYPE-2 クローンとは構造が同じであり頂点の値が異なる部分木，TYPE-3 クローンとは構造が類似して入るが同じではない部分木を指す．

3. 編集スクリプト

編集スクリプトは，2 つの AST が与えられたときに一方の AST をもう一方の AST に変更するために必要な操作の列である [6], [8], [9]．既存研究では，編集スクリプトは以下の種類の操作から構成される．

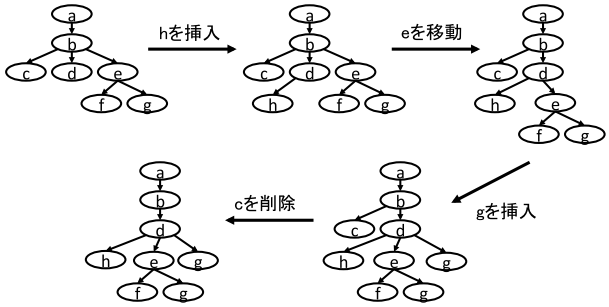
挿入 $insert(t, t_p, i, l, v)$
頂点の追加を表す．挿入する頂点の ID として t，頂点のラベルとして l，頂点の値として v，挿入後に親頂点とする頂点の ID として t_p，何番目の子にするかを表す数値として i を引数に持つ．

削除 $delete(t)$
頂点の削除を表す．削除する頂点の ID として t を引数に持つ．

更新 $update(t, v)$
頂点の値の更新を表す．更新の対象となる頂点の ID として t，新しい値として v を引数に持つ．

移動 $move(t, t_p, i)$
部分木の移動を表す．移動する部分木の根頂点の ID として t，移動先の親頂点の ID として t_p，何番目の子にするかを表す数値として i を引数に持つ．

図 2 に AST への編集の例を示す．図 2 (a) では，AST において，頂点 h を頂点 d の 1 番目の子として挿入し，頂点 e を頂点 d の 2 番目の子として移動，頂点 g を頂点 d の 3 番目の子として挿入，頂点 c を削除，の順で操作を行っている．このように，AST の編集は対象の AST に操作を適用することで行う．また，この場合の編集スクリプトは図 2 (b) である．ただし，ここでは挿入された頂点のラベルと値は省略している．



(a) AST の操作

```
insert(h, d, 1, …, …)
move(e, d, 2)
insert(g, d, 3, …, …)
delete(c)
```

(b) 対応する編集スクリプト

図 2 編集スクリプトの例
Fig. 2 An example of edit script.

編集スクリプトを構成する操作の数を**編集スクリプトの長さ**と呼ぶ．図 2 (b) の編集スクリプトの長さは 4 である．編集スクリプトが長いほど，多くの操作が含まれる．多くの操作が行われる変更は人間にとって理解に要する労力が大きい [8]．よって，編集スクリプトの長さは人間が変更内容を理解するために要する労力を表す．そのため，編集スクリプトはその長さが短いほど，人間にとって理解しやすい編集スクリプトであるといえる．

編集スクリプトは，一方の AST をもう一方の AST に変換するための 1 つの手順を表すが，それが必ずしも人間が行ったソースコードの変更の手順を再現しているわけではない．人間が行ったソースコード変更の手順を再現するためには，文献 [17] で紹介されている手法やツールを利用して人間が行った変更そのものを記録する必要がある．

3.1 既存の編集スクリプト生成手法

これまでに，編集スクリプトを生成する多くの手法が提案されている [5], [6], [9], [10], [15]．それらの中で高速に編集スクリプトを生成する手法は RTED [15] である．それでも計算量は少なくなく，大きなソースファイルに対しては短い時間で編集スクリプトを生成できない．また，これらの手法はソースコードの変更時に頻繁に現れるコードの移動を考慮しておらず，移動元の頂点をすべて削除し移動後の頂点をすべて追加するという方法で変更を表現する．その結果，編集スクリプトは長くなり，理解しにくくなる．

頂点の移動を考慮した短い編集スクリプトの生成は NP 困難である．そのため，経験則を用いることにより，移動を考慮したうえで比較的短い編集スクリプトを生成す

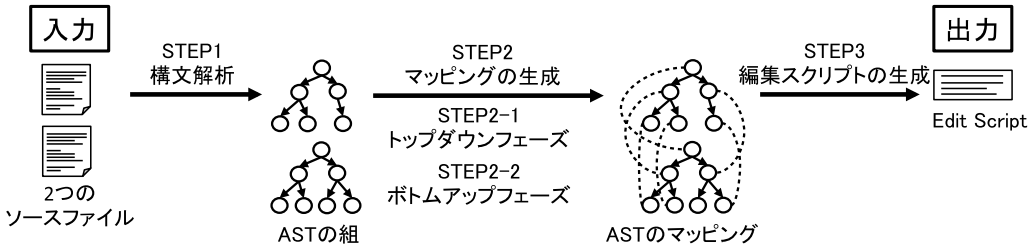


図 3 GumTree の概要
Fig. 3 An overview of GumTree.

る手法が研究されている．そのうちで最も有名なものが Chawathe らのアルゴリズム [6] である．しかし，Chawathe らのアルゴリズムには制約があり，ソースファイルを表現した細粒度な AST にそのままでは適用できない．

また，XML 文書を対象として編集スクリプトを生成するアルゴリズムが提案されている [2], [7]．これらのアルゴリズムは Chawathe らのアルゴリズムと異なり制約を持たない．しかし，これらのアルゴリズムは速度を最も重視しており，人間にとって変更内容を理解しやすい編集スクリプトの生成は重視していない．

AST において編集スクリプトを生成するアルゴリズムのうち，最も有名なアルゴリズムが ChangeDistiller [9] である．ChangeDistiller は Chawathe らのアルゴリズムに影響を受けており，AST においてより効果的に動作するように調整されている．しかし，ChangeDistiller は簡略化した AST を対象としており，1 つの文に多くの要素を持ちうる言語において，ChangeDistiller は細粒度な編集スクリプトを生成できない．

3.2 GumTree

Falleri らは，細粒度な AST において，実用的な時間の範囲内で移動を考慮した短い編集スクリプトを生成する GumTree を開発した [8]．GumTree の概要を**図 3** に示す．GumTree は図 1 (a) および**図 4** (a) のようなソースファイルの組を入力として与えると，図 4 (b) のような形式で編集スクリプトを出力する．

GumTree は入力として与えられた 2 つのソースファイルに対して以下の処理を行い編集スクリプトを出力する．

STEP1（構文解析） 与えられた 2 つのソースファイルそれぞれについて構文解析を行い，AST を生成する．

STEP2（マッピング） STEP1 で生成した 2 つの AST について，STEP2-1 および STEP2-2 の処理を行うことにより，2 つの AST における頂点や部分木のマッピングを生成する．

STEP2-1（トップダウンフェーズ） 2 つの AST に共通して存在する部分木を発見する．

STEP2-2（ボトムアップフェーズ） STEP2-1 で発見した共通の部分木を基準として，それらに含まれていない部分木のうち類似する部分木を発見する．

STEP3（編集スクリプト生成） STEP1 で生成した 2 つの AST と，STEP2 で生成したマッピングを用いて編集スクリプトを生成する．編集スクリプトの生成には Chawathe らのアルゴリズム [6] を用いる．

```
public class Test{
 private String foo(int i){
   if(i == 0) return “Bar";
   else if(i == -1) return “Foo!”;
 }
}
```

(a) 変更後のソースファイル

```
insert(t1, n, 2, ReturnStatement, ε)
insert(t2, t1, 1, StringLiteral, Bar)
insert(t3, n, 3, IfStatement, ε)
insert(t4, t3, 1, InfixExpression, ==)
insert(t5, t4, 1, SimpleName, i)
insert(t6, t4, 2, PrefixExpression, -)
insert(t7, t6, 1, NumberLiteral, 1)
move(p, t3, 2)
update(c, private)
```

(b) 出力される編集スクリプト

図 4 GumTree が出力する編集スクリプトの例（図 1 のソースコードに対して変更を加えたもの）
Fig. 4 An example of GumTree's edit script for revised source code of Fig. 1.

GumTree は生成した編集スクリプトに基づき変更内容をソースコード上に表すことができる．図 4 (b) の編集スクリプトに基づく変更内容の表示例を**図 5** に示す．緑色が挿入と識別されたコードである．この例では 2 カ所へ挿入が行われている．黄色のコードが更新を表している．青色のコードが移動を表しており，return 文が if 文の直後から if 文の else 節に移動している．

3.3 研究動機

ソースコードの実装において，コピーアンドペーストが頻繁に行われる [11]．しかしながら GumTree は，挿入，削除，更新，移動の 4 種類の操作で編集スクリプトを表現するため，開発者のコピーアンドペーストという 1 つの操作が複数の挿入の操作で表現されることになる．

図 6 は，新しいメソッド newFoo が追加された変更を表す．メソッド newFoo はメソッド foo と TYPE-2 クロー

```
public class Test{
 public String foo(int i){
   if(i == 0) return "Foo!";
 }
}
```

(a) 変更前のソースファイル

```
public class Test{
 private String foo(int i){
   if(i == 0) return “Bar";
   else if(i == -1) return “Foo!”;
 }
}
```

更新 挿入 移動

(b) 変更後のソースファイル

図 5 GumTree が出力した編集スクリプトに基づいた変更内容の強調表示

Fig. 5 Highlighting changed program elements based on GumTree's edit script.

ンの関係にある．図 6 (b) は変更後のソースコードの AST を表す．また，説明のため AST の各頂点の ID としてアルファベットを付加している．この変更に対して，GumTree が生成した編集スクリプトは図 6 (c) である．この編集スクリプトより，この変更では 15 の頂点が挿入されたことが分かる．

しかし，実際に挿入された頂点群（部分木）は，変更前の AST に存在していた AST の頂点群（部分木）と同様のものである．そのため，既存の頂点を利用することでこの編集スクリプトをより簡潔に表すことができる．また，この例において，挿入された部分木は既存の部分木と同じ形である．よって，部分木の根頂点のみへの操作として表すことで（図 6 (d)），編集スクリプトを大幅に短縮できる．

4. 提案手法

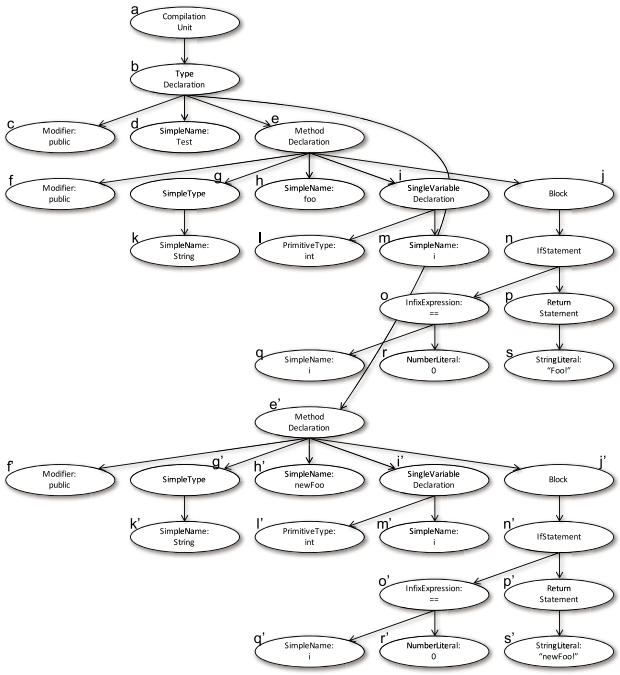
4.1 概要

3.3 節で説明したように，追加された部分木を頂点の挿入の繰返しではなく 1 つのコピーアンドペーストとして表すことで，人間にとってより理解しやすい編集スクリプトを生成できる．提案手法では編集スクリプトを構成する操作として，挿入，削除，更新，移動に加えてコピーアンドペーストを追加することで，人間にとってより理解しやすい編集スクリプトを生成する．図 6 の場合だと編集スクリプトを大幅に短くすることができる．この例において，提案手法が生成する編集スクリプトは GumTree が生成する編集スクリプトよりも短いだけでなく，コピーアンドペーストや更新といった具体的な変更内容をイメージしやすい．そのため，提案手法が生成する編集スクリプトは GumTree が生成する編集スクリプトよりも人間にとって理解しやすいといえる．

本研究では GumTree を拡張することで提案手法を設計している [8]．提案手法の概要を図 7 に示す．提案手法は

```
public class Test{
 public String foo(int i){
   if(i == 0) return "Foo!";
 }
 public String newFoo(int i){
   if(i == 0) return "newFoo!";
 }
}
```

(a) 変更後のソースファイル



(b) 変更後の AST

```
insert(e’, b,  4, MethodDeclaration, ε)
insert(f’, e’, 1, Modifier, public)
insert(g’, e’, 2, SimpleType, String)
insert(k’, g’, 1, SimpleName, String)
insert(h’, e’, 3, SimpleName, newFoo)
insert(i’, e’, 4, SingleVariableDeclaration, ε)
insert(l’, i’, 1, PrimitiveType, int)
insert(m’, i’, 2, SimpleName, i)
insert(j’, e’, 5, Block, ε)
insert(n’, j’, 1, IfStatement, ε)
insert(o’, n’, 1, InfixExpression, ==)
insert(q’, o’, 1, SimpleName, ε)
insert(r’, o’, 2, NumberLiteral, 0)
insert(p’, n’, 2, ReturnStatement, ε)
insert(s’, p’, 1, StringLiteral, newFoo!)
```

(c) Gum Tree が出力する編集スクリプト

```
c&p(e, b, 4)
update(h’, newFoo)
update(s’, “newFoo!”)
```

(d) 提案手法が出力する編集スクリプト

図 6 コピーアンドペーストが行われた変更の例（図 1 のソースコードに対して変更を加えたもの）

Fig. 6 An example of revised code with copy & paste for source code of Fig. 1.

2 つのソースファイルを入力とし，GumTree と同様に 3 つの STEP を経て編集スクリプトを生成する．提案手法において，編集スクリプトを構成する操作は挿入，削除，更新，

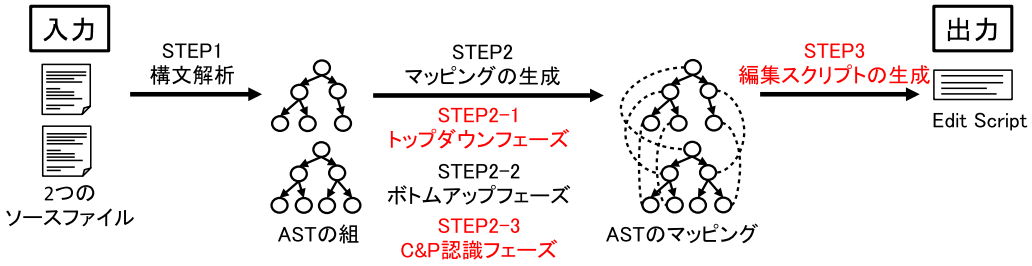


図 7 提案手法の概要
Fig. 7 An overview of the proposed technique.

移動，コピーアンドペーストの 5 種類である．このうち，挿入，削除，更新，および移動の定義は 3 章で述べた従来の編集スクリプトの定義と同じである．コピーアンドペーストの定義を以下に示す．

コピーアンドペースト $c\&p(t, t_p, i)$
部分木のコピーアンドペーストを表す．コピーアンドペーストする部分木の根頂点の ID として t，コピーアンドペースト先の親頂点の ID として t_p，何番目の子にするかを表す数値として i を引数に持つ．

4.2 手順

提案手法は GumTree のアルゴリズムを拡張している．図 7 では，提案手法において拡張あるいは追加した STEP を赤で示している．それらの STEP の概要を以下に示す．

STEP2-1（トップダウンフェーズ） 2 つの AST に共通して存在する部分木を発見する．さらに，並行してコピーアンドペーストの対象となりうる部分木（コピーアンドペーストの候補）を抽出する．

STEP2-3（C&P 識別フェーズ） STEP2-1 において抽出したコピーアンドペーストの候補から，STEP2-2 においてマッピングされていないものを発見する．それらがコピーアンドペーストによって生成された部分木となる．

STEP3（編集スクリプト生成） STEP1 で生成した 2 つの AST と，STEP2 で生成したマッピングを用いて編集スクリプトを生成する．STEP2-3 において識別した部分木をコピーアンドペーストによって作成するものとして処理する．

以降，上記の各 STEP について詳細に説明する．

4.3 STEP2-1（トップダウンフェーズ）

STEP2-1 では，GumTree が行う処理に加えて，コピーアンドペースト操作の対象となりうる部分木の候補を抽出する．STEP2-1 は以下の手順で実行される．なお，手順 (1) および (2) は GumTree の処理，手順 (3) は提案手法で追加した処理である．

(1) 2 つの AST から類似する部分木を発見し記録する．ここでは，変更前 AST における n 個の部分木が，変更後 AST における m 個の部分木と類似しているとする．n 個の部分木を頂点とする集合と m 個の部分木を頂点とする集合があり，類似度が閾値以上の頂点間に辺が存在する状態（2 部グラフの構造）である．

(2) 発見した類似部分木のうち，以下の手順でマッピングを作成する．n 対 m の類似部分木の中から最も類似度が高い組合せを 1 つ取得し，それをマッピングに追加する．そして，マッピングされた部分木の組（2 つの部分木を表す頂点とそれらの間の辺）を集合から取り除く．残った集合（$(n-1)$ 対 $(m-1)$ の類似部分木）に対して，同様の処理を行う．この処理を，辺がなくなるまで繰り返す．

(3) (2) が終了した時点で，集合に残っている類似部分木は，マッピングされなかったものである．これらには類似する部分木が存在しているが，それらにはより類似する他の部分木が存在していることを意味する．集合に残っているマッピングされなかった類似部分木のうち，変更後 AST の部分木とそれと最大類似度を持つ（すでにマッピングされている）変更前 AST の部分木をコピーアンドペーストの候補として記録する．

4.4 STEP2-3（C&P 識別フェーズ）

STEP2-3 では，コピーアンドペーストとみなす変更を決定する．変更後の AST を先行順に探索し，下記の条件をともに満たす頂点を探す．

- STEP2-2 の終了時点でマッピングに追加されていない．
- コピーアンドペーストの候補としてトップダウンフェーズで記録されている．

条件を満たす頂点を発見すると，変更前の AST において対応する頂点（トップダウンフェーズでクローンとして検出された部分木）とともに，コピーアンドペーストされた頂点としてマッピングに追加する．そして，コピーアンドペーストされたとしてマッピングに追加された頂点を根とする部分木に含まれるすべての子頂点をコピーアンドペーストの候補から除外する．

コピーアンドペーストの判定において制約を設けた．提案手法では，共通する部分木を発見する際に識別子名やリテラル等は正規化される．そのため，偶然同じ形になった

部分木もコピーアンドペーストとして識別される可能性がある．たとえば，変数宣言において，宣言する変数の変数名と，代入する値の双方が異なっていても，部分木は同じ形となるため，コピーアンドペーストと識別される可能性がある．このような誤検出を防ぐため，提案手法では，部分木の形が同じであっても，その部分木を構成する頂点が持つすべての値が異なる場合はコピーアンドペーストとして識別しない．

4.5 STEP3（編集スクリプトの生成）

編集スクリプトの生成において，挿入，削除，移動，更新の検出は GumTree と同様に Chawathe らのアルゴリズム [6] を用いる．ただし，提案手法では Chawathe らのアルゴリズムを拡張しており，検出する操作としてコピーアンドペーストを加えている．この STEP では各 AST を一度ずつ探索し，編集スクリプトを生成する．

(1) 変更後の AST を幅優先探索により挿入，移動，更新に加えてコピーアンドペーストを検出する．
(2) 変更前の AST を後行順探索により削除を検出する．

(1) の処理において，C&P 識別フェーズにおいてコピーアンドペーストとしてマッピングされた頂点を発見すると，編集スクリプトにコピーアンドペーストとして追加される．

4.6 提案手法による編集スクリプト生成の例

図 8 は提案手法による AST のマッピングの例を表している．この例では変更前 AST として図 1 (b)，変更後 AST として図 6 (b) を用いた．

STEP2-1（トップダウンフェーズ）では，変更前 AST の部分木 e と変更後 AST の部分木 e が 1 対 1 対応であるためにマッピングされる．さらに，変更前 AST の部分木 e と変更後 AST の部分木 e' が類似部分木の組であり，部分木 e' は変更前 AST のどの部分木ともマッピングされていないため，コピーアンドペーストの候補として記録される．

STEP2-2（ボトムアップフェーズ）では，まずコンテナマッピングという処理が行われる．コンテナマッピングは，多くの子頂点がマッピングされているがその頂点自身はマッピングされていない場合に，その頂点もマッピングに加える処理である．この処理により，変更前 AST の頂点 b と変更後 AST の頂点 b および変更前 AST の頂点 a と変更後 AST の頂点 a がマッピングされる．次にリカバリマッピングという処理が行われる．この処理では，コンテナマッピングによりマッピングされた頂点から下にたどることにより新しくマッピングできる頂点を探す．この例では，リカバリマッピングにより，変更前 AST の頂点 c と変更後 AST の頂点 c および変更前 AST の頂点 d と変更後 AST の頂点 d がマッピングされる．

STEP2-3（C&P 認識フェーズ）では，STEP2-1 で記録されたコピーアンドペーストの候補のうち，変更後 AST の部分木が STEP2-2 においてもマッピングされていないものをコピーアンドペーストとして識別する．この例では，変更後 AST の部分木 e' はマッピングされていないため，変更前 AST の部分木 e からのコピーアンドペーストにより生成されたと識別する．

STEP3 では，マッピングの情報およびコピーアンドペーストの情報をもとに編集スクリプトを生成する．

- マッピングされた部分木については，それが移動であ

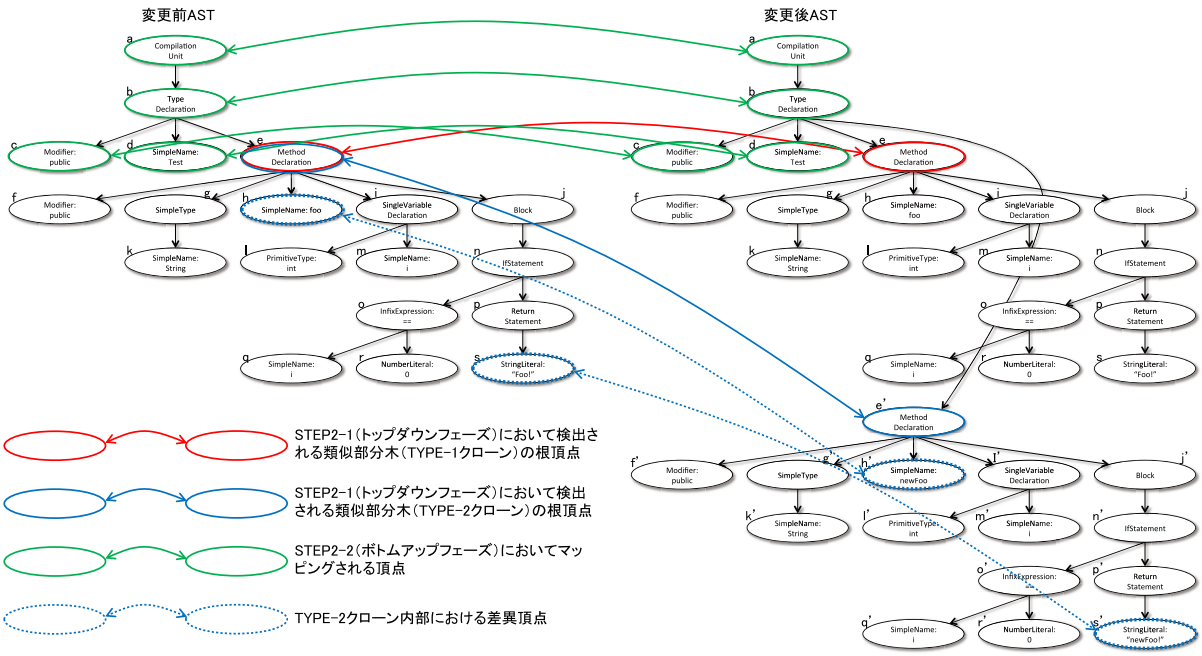


図 8 AST のマッピング例（図 1 (b) と図 6 (b) のマッピング）
Fig. 8 An example of AST mapping between Figs. 1 (b) and 6 (b).

る場合は move 操作を編集スクリプトに加える．マッピングされた頂点のうち，値が異なるものについては update 操作を編集スクリプトに加える．それ以外の場合はなにもしない．この例では移動とみなされた部分木も値が異なる頂点もないため，マッピングされた部分木からはまったく操作は生成されない．

- 変更前 AST に存在する部分木のうち，変更後 AST とマッピングされておらず，かつ，コピーアンドペーストにも含まれないものについては，delete 操作を編集スクリプトに加える．この例では delete 操作の対象となる部分木は存在しない．
- 変更後 AST に存在する部分木のうち，変更前 AST とマッピングされておらず，かつ，コピーアンドペーストにも含まれないものについては，insert 操作を編集スクリプトに加える．この例では insert 操作の対象となる部分木は存在しない．
- コピーアンドペーストと識別された部分木については c&p 操作を編集スクリプトに加える．また，その部分木に存在する頂点のうち，変更後 AST と変更前 AST で値が異なるものについては，update 操作を編集スクリプトに加える．この例では，コピーアンドペーストの操作として `c&p(e,b,4)` が編集スクリプトに加えられる．また，値の異なる頂点に対する操作として，`update(h', newFoo)` および `update(s',"newFoo!")` が編集スクリプトに加えられる．

その結果，図 6 (d) に示す編集スクリプトが出力される．

5. 実験

3.1 節で述べたように，編集スクリプトを生成する手法として GumTree 以外の手法も存在する．しかし，Falleri らの研究において，GumTree がそれらの手法より，変更内容を人間にとって理解しやすく表現できることが示されている [8]．そのため，提案手法と GumTree を比較し，提案手法の有効性を評価する．

5.1 準備

この実験では，提案手法と GumTree は下記のしきい値を用いて実行した．

- トップダウンフェーズにおいてマッピングする部分木の高さの最小値を 2 とする．
- ボトムアップフェーズにおいてマッピングする部分木の類似度の最小値を 0.5 とする．
- 計算時間短縮のため，ボトムアップフェーズにおいてマッピングする部分木の頂点数の最大値を 100 とする．

これらのしきい値は Falleri らの実験 [8] と同じである．

実験対象として 14 のソフトウェアで構成される CVS-Vintage dataset [13] を用いた．CVS-Vintage dataset には 43,250 のソースファイル，ソースファイルへの変更が行われた 352,182 のリビジョンが含まれている．

5.2 手順

2 つのツールを用いて編集スクリプトを生成し，それらを比較する．本実験では，1 つのコミット前後の各ソースファイルの対を 1 つの変更と呼ぶ．つまり，あるコミットにおいて改版されたソースファイルの数がそのコミットにおける変更の数となる．2 つのツールへの入力として与える変更は CVS-Vintage dataset に含まれる 14 のソフトウェアのリポジトリから取得する．各ソフトウェアから 1,000 の変更を取得する．リポジトリに含まれる全変更数が 1,000 に満たない場合は全変更を使用する．実験に用いる変更の取得は下記の手順で行う．

(1) リポジトリから，ソースファイルが変更されたリビジョンを特定し，そのリビジョンにおいて変更されたソースファイルを取得する．
(2) 取得したソースファイルのそれぞれについて，直前のリビジョンから対応するソースファイルを取得する．
(3) 対応する変更前のソースファイルと変更後のソースファイルを 1 つの変更として保存する．
(4) 保存した変更のうち，提案手法に入力として与えた場合に長さ 1 以上の編集スクリプトを生成する変更のみを残す．フォーマットの変換やコメントの追加のみからなる変更等が取り除かれる．
(5) 残った変更からランダムに 1,000 選択する．

上記の手順により，13,699 の変更を取得した．取得した変更を提案手法と GumTree へ入力として与え，編集スクリプトの長さとツールの実行時間を取得する．

5.3 評価基準

各ツールが生成する編集スクリプトの長さと実行時間を評価基準とした．編集スクリプトが短いほど人間が変更の内容を理解するための負担が少なくなる [8]．そのため，より短い編集スクリプトを生成するツールほど優れている．

5.4 結果

18%の変更において提案手法の方が GumTree よりも短い編集スクリプトを生成していた．82%の変更については，提案手法と GumTree の編集スクリプトの長さは同じであった．提案手法が GumTree よりも長い編集スクリプトを生成している変更はまったくなかった．

表 1 は，提案手法の方が短い編集スクリプトを生成した変更に対して，2 つのツールの変更の長さを表している．また，**図 9** は，各変更において提案手法を用いることにより編集スクリプトがどの程度変化するかを箱ひげ図を用いて表している．縦軸は GumTree が生成する編集スクリプトの長さを 1.0 としたときの提案手法が生成する編集スクリプトの長さを表す．横軸が対応するソフトウェアを表

表 1 ツールによって編集スクリプトの長さが異なる場合における編集スクリプトの長さ
Table 1 A summary of length of edit scripts where the proposed technique generated different edit scripts from GumTree.

ソフトウェア	最大値		第 3 四分位数		中央値		第 1 四分位数		最小値	
	GumTree	提案手法	GumTree	提案手法	GumTree	提案手法	GumTree	提案手法	GumTree	提案手法
argouml	3,049	2,708	231	198	69	49	30	22	4	2
carol	1,586	1,581	231	214	116	107	54	32	7	3
columba	892	872	128	100	47	39	18	11	5	1
dnsjava	1,632	1,544	192	167	75	71	33	18	4	2
jboss	2,876	2,457	224	209	76	62	32	21	5	2
jedit	2,858	2,853	202	178	79	67	29	25	4	2
jhotdraw	1,691	1,686	173	144	72	64	29	19	4	2
junit	751	728	142	133	75	64	33	26	7	2
log4j	3,029	2,827	195	165	75	60	33	24	4	2
jdtcore	13,269	12,628	195	169	84	65	36	23	4	2
workbench	3,651	3,351	195	170	72	55	22	12	4	2
scarab	3,016	2,928	190	176	83	70	31	17	5	2
struts	1,486	1,313	159	129	59	43	33	16	5	1
tomcat	2,152	2,064	162	151	54	43	25	18	4	2

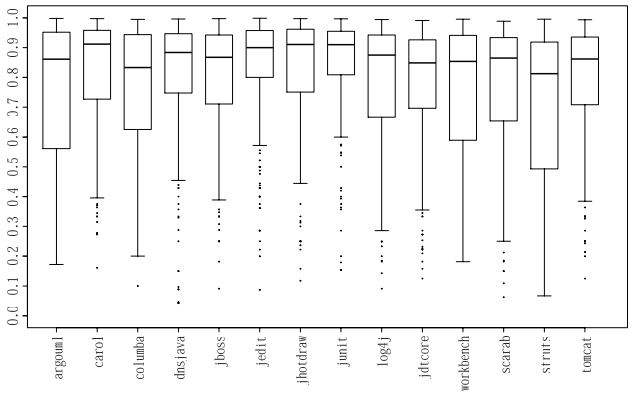


図 9 編集スクリプトの長さが異なる変更の長さの比較
Fig. 9 Comparing length of edit scripts where the proposed technique generated different edit scripts from GumTree.

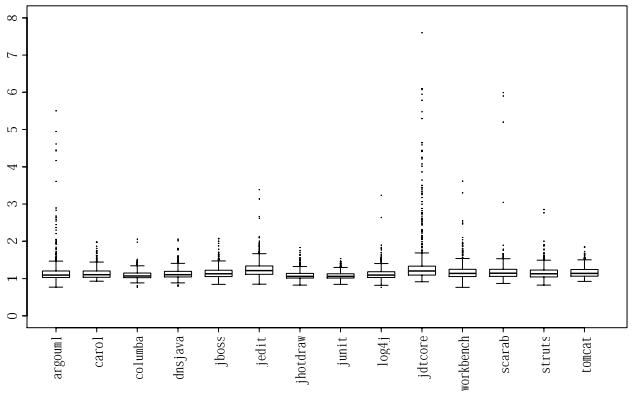


図 10 すべての変更に対する実行時間の比較
Fig. 10 Comparing run-time for all the target changes.

す．すべてのソフトウェアにおいて，中央値がおよそ 0.8 から 0.9 の間にある．つまり，半数の変更において，提案手法を使うことにより 10%から 20%程度は編集スクリプトが短くなっている．具体的な数値をあげると，58.5%の変更において編集スクリプトが 10%以上短縮されていた．

次に実行時間について述べる．実行時間は編集スクリプトの長さが異なる変更のみではなく，全変更について調査を行った．**図 10** は，提案手法を用いることにより実行時間がどの程度変化するかを箱ひげ図を用いて表している．縦軸は，GumTree を用いた場合の実行時間を 1 としたときの提案手法の実行時間を表す．横軸は対応するソフトウェアを表す．GumTree と比較して提案手法は実行時間が長くなっている．すべての変更における提案手法と GumTree の実行時間を，ウィルコクソンの符号順位検定を用いて有意水準 $\alpha = 0.05$ で検定したところ，統計的に有意な差があった．しかしながら，提案手法が編集スクリプトの生成に要する時間はそれほど長くはない．96%の変更において提案手法は GumTree の 1.5 倍以下の実行時間で編集スクリプトを生成していた．また，75%の変更において提案手法は 1 秒以内に編集スクリプトを生成し，96%の変更において 2 秒以内に編集スクリプトを生成していた．

5.5 考察

5.5.1 提案手法が編集スクリプトを大幅に短縮できる場合

実験において，編集スクリプトを大幅に短縮できた変更の具体例を説明する．**図 11** に ArgoUML において行われた変更を示す．図上部のソースコードが変更前のソースコード，図下部が変更後のソースコードである．この変更では，setReception というメソッドが追加されている．追加された setReception と変更前から存在していた setReceiver メソッドは，参照している変数名や呼び出すメソッドは異なるがメソッド内部の構造は非常によく似ている．

(a) メソッドが追加された変更　(b) GumTree の出力　(c) 提案手法の出力

図 11　提案手法によって編集スクリプトを短縮できた例

Fig. 11　An example of code change where the proposed technique generated a shorter edit script than GumTree.

図 11 (b) は GumTree による変更の表現である．変更後のソースコードを見ることにより，緑色のコード片が追加されたことが分かる．図 11 (c) は提案手法による表現である．変更前のソースコードにおいて，紫色でハイライトされたコード片はコピー元と判定されたコード片である．また，変更後のソースコードを見ることによりこのコード片が別の箇所にペーストされたと判定されていることが分かる．黄色でハイライトされたコード片はコピー元とコピー先で異なる部分を表している．これらは，ペースト後に更新されたと判定されたことを意味する．

GumTree による表現では，変更により setReception というメソッドが追加されたことが分かる．しかし，追加されたメソッドは既存のメソッド setReceiver と似ていること，およびそのメソッドをコピーアンドペーストにより再利用した可能性があることは分からない．

提案手法による表現では，追加されたメソッドは既存メソッドと似ており，コピーアンドペーストおよび軽微な修正により追加された可能性があることを示している．この編集スクリプトの利用者がコピー元のメソッドについて知識がある場合は，追加されたメソッドを理解する助けになることも期待できる．

この例における編集スクリプトの長さは，GumTree は 20，提案手法は 5 である．提案手法を用いることにより編集スクリプトは 75%短縮されている．

5.5.2　編集スクリプトの長さと変更の理解しやすさの関係

本実験では GumTree と提案手法による編集スクリプトの長さを比較した．また，長さが異なった編集スクリプトを例示し，提案手法の有用性を述べた．しかしながら開発者が編集スクリプトそのものを見ることはあまりなく，図 11 で示すような可視化した変更を見て理解を試みることが多いだろう．そのような場合は，開発者は編集スクリプトそのものは見ないため，その長さによって必ずしも変更の理解しやすさが決まるわけではない．GumTree が生成した長い編集スクリプトによる可視化が，提案手法が生成した短い編集スクリプトによる可視化に比べて，理解性が高い場合もあるだろう．たとえば，提案手法が広い範囲のコードをコピーアンドペーストとして認識した場合は，既存手法による連続したコードの追加よりも理解しやすい変更の可視化になっているかもしれない．また，複数箇所の小さい範囲のコードが変更されていた場合は，コピーアンドペーストと更新ではなく，コードの追加とした方が理解性が高いかもしれない．提案手法の有用性をより厳密に評価するためには，さまざまな変更に対して理解しやすい編集スクリプト（とその可視化）になっているのかを実際に人間が評価する必要がある．

5.5.3　コピーアンドペーストとして識別する部分木

現在の実装では，まったく同じ部分木（木の構造が同じであり，かつ，頂点の値も同じ）と構造は同じであるが

頂点の値が異なる部分木をコピーアンドペースト処理が行われたと認識する．前者は TYPE-1 のクローン，後者は TYPE-2 のクローンである．実験では 82%の変更において，提案手法と GumTree の変更の長さが同じであった（コピーアンドペーストを含まない変更であった）．既存の AST を利用したコードクローン検出技術 [4] を利用することで，TYPE-3 のクローンもコピーアンドペーストとして認識することができる．もし TYPE-3 のクローンも検出できるように実装を拡張した場合，より多くの変更がコピーアンドペーストを含むと識別される．また，実験では高さが 2 以上の部分木をコピーアンドペーストの検出対象とした．もし，より大きな部分木のみを対象すれば，コピーアンドペーストを含むと認識される変更の数は少なくなる．

5.6 実験の妥当性について

実験では Java にて記述されたソースファイルのみを使用した．提案手法や GumTree のアルゴリズムは Java の特徴とは無関係だが，他の言語を対象とした場合において異なる結果が得られる可能性がある．

実験において提案手法と GumTree の双方において，しきい値は Falleri らの研究において用いられた値と同じ値を使用している．しきい値が異なれば，各ツールは異なる振舞いをする．実験結果について，しきい値の影響を評価するにはより多くの実験が必要とされる．

本実験では，Falleri らの実験と同様に実験対象として CVS-Vintage dataset を使用した．他のデータセットを実験対象にした場合に異なる結果が得られる可能性がある．

6. おわりに

本論文では，ソースファイルの変更内容を人間が理解しやすく表現するために，コピーアンドペーストを考慮することにより既存手法の GumTree を改良した．また，提案手法の有効性を評価するために評価実験を行った．実験では，提案手法と GumTree を比較し，提案手法を評価した．実験において，CVS-Vintage dataset を用いて 14 のソフトウェアの開発履歴から 13,699 の変更を実験対象として取得した．取得した変更に対して提案手法と GumTree が生成する編集スクリプトの長さと，編集スクリプトの生成に要する実行時間を比較した．その結果，提案手法は GumTree より短い編集スクリプトを生成できることを確認した．また，ほとんどの変更に対して提案手法は数秒以内で編集スクリプトを生成できることを確認した．

今後の課題は以下のとおりである．

- より多くのソフトウェアに対して実験を行う．
- 被験者実験を行い，コピーアンドペーストを考慮した表現により，変更内容がどの程度理解しやすくなったか調査する．
- TYPE-3 クローンをコピーアンドペーストとして識別できるよう手法を改良する．

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（S）（課題番号：JP25220003）の助成を得て行われた．

参考文献

[1] Ahmed, T.M., Shang, W. and Hassan, A.E.: An Empirical Study of the Copy and Paste Behavior During Development, *Proc. 12th Working Conference on Mining Software Repositories*, pp.99–110 (2015).

[2] Al-Ekram, R., Adma, A. and Baysal, O.: diffX: An Algorithm to Detect Changes in Multi-version XML Documents, *Proc. 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pp.1–11 (2005).

[3] Asaduzzaman, M., Roy, C.K., Schneider, K.A. and Penta, M.D.: LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines, *Proc. 2013 IEEE International Conference on Software Maintenance*, pp.230–239 (2013).

[4] Baxter, I., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone Detection Using Abstract Syntax Trees, *Proc. 14th International Conference on Software Maintenance*, pp.368–377 (1998).

[5] Bille, P.: A Survey on Tree Edit Distance and Related Problems, *Theoretical Computer Science*, Vol.337, No.1-3, pp.217–239 (2005).

[6] Chawathe, S.S., Rajaraman, A., Garcia-Molina, H. and Widom, J.: Change Detection in Hierarchically Structured Information, *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, pp.493–504 (1996).

[7] Cobena, G., Abiteboul, S. and Marian, A.: Detecting changes in XML documents, *Proc. 18th International Conference on Data Engineering*, pp.41–52 (2002).

[8] Falleri, J., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M.: Fine-grained and Accurate Source Code Differencing, *Proc. 29th ACM/IEEE International Conference on Automated Software Engineering*, pp.313–324 (2014).

[9] Fluri, B., Wuersch, M., Pinzger, M. and Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction, *IEEE Trans. Software Engineering*, Vol.33, No.11, pp.725–743 (2007).

[10] Hashimoto, M. and Mori, A.: Diff/TS: A Tool for Fine-Grained Structural Change Analysis, *Proc. 2008 15th Working Conference on Reverse Engineering*, pp.279–288 (2008).

[11] Kim, M., Bergman, L., Lau, T. and Notkin, D.: An Ethnographic Study of Copy and Paste Programming Practices in OOPL, *Proc. 2004 International Symposium on Empirical Software Engineering*, pp.83–92 (2004).

[12] Miller, W. and Myers, E.W.: A file comparison program, *Software: Practice and Experience*, Vol.15, No.11, pp.1025–1040 (1985).

[13] Monperrus, M. and Martinez, M.: CVS-Vintage: A Dataset of 14 CVS Repositories of Java Software, Technical Report hal-00769121, INRIA (2012).

[14] Myers, E.W.: An O(ND) Difference Algorithm and Its Variations, *Algorithmica*, Vol.1, pp.251–266 (1986).

[15] Pawlik, M. and Augsten, N.: RTED: A Robust Algorithm for the Tree Edit Distance, *Proc. VLDB Endow.*,

Vol.5, No.4, pp.334–345 (2011).
[16] Rattan, D., Bhatia, R. and Singh, M.: Software Clone Detection: A Systematic Review, *Information and Software Technology*, Vol.55, No.7, pp.1165–1199 (2013).
[17] 大森隆行，林　晋平，丸山勝久：統合開発環境における細粒度な操作履歴の収集および応用に関する調査，コンピュータソフトウェア，Vol.32, No.1, pp.60–80 (2015).
[18] 肥後芳樹，楠本真二，井上克郎：コードクローン検出とその関連技術，電子情報通信学会論文誌，Vol.J91-D, No.6, pp.1465–1481 (2008).
[19] 堀田圭佑，肥後芳樹，楠本真二：生成防止，分析効率化，不具合検出を中心としたコードクローン管理支援技術に関する研究動向，コンピュータソフトウェア，Vol.31, No.1, pp.14–29 (2014).
[20] 神谷年洋，肥後芳樹，吉田則裕：コードクローン検出技術の展開，コンピュータソフトウェア，Vol.28, No.3, pp.28–42 (2011).

肥後 芳樹 （正会員）

2002 年大阪大学基礎工学部情報科学科中退．2006 年同大学大学院博士後期課程修了．2007 年同大学院情報科学研究科コンピュータサイエンス専攻助教．2015 年同准教授．博士（情報科学）．ソースコード分析，特にコードクローン分析，リファクタリング支援およびソフトウェアリポジトリマイニングに関する研究に従事．電子情報通信学会，日本ソフトウェア科学会，IEEE 各会員．

大谷 明央

2014 年大阪大学基礎工学部情報科学科卒業．2016 年同大学大学院博士前期課程修了．在学時，コードクローン分析やリポジトリマイニングに関する研究に従事．

楠本 真二 （正会員）

1988 年大阪大学基礎工学部卒業．1991 年同大学大学院博士課程中退．同年同大学基礎工学部助手．1996 年同講師．1999 年同助教授．2002 年同大学大学院情報科学研究科助教授．2005 年同教授．博士（工学）．ソフトウェアの生産性や品質の定量的評価に関する研究に従事．電子情報通信学会，IEEE，IFPUG 各会員．