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

大谷 明央[1,a)]　肥後 芳樹[1,b)]　楠本 真二[1,c)]

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．その中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，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．

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，変更の内容をわ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 [1][2][3]．例えば，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 [1] 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には，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ものになるとは限らない．

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 [4][5][6][7]．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では，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を表したものである [7]．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にとって変更内容を理解しやすい．

抽象構文木に基づく手法には，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おいて，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．この課題点を解決するために，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大きな抽象構文木に対しても効率的に移動を考慮した編集スクリプトを生成する方法を考案した [8]．また，Falleri らは実際のソフトウェア開発に用いられた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．また他の手法と比較することで，生成する編集スクリプトが人間にとってより理解しやすいことを示した．

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も頻繁に行われる [9][10]．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

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

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操作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操作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った．実験では，14 のオープンソースソフトウェアのリポジトリを含む CVS-Vintage dataset を対象とし，13,699 の変更を対象に提案手法と Falleri らの手法を比較した．CVS-Vintage dataset は Felleri らが実験に利用したデータセットであり，本研究でも同じデータセットに対して実験を行う．実験の結果，提案手法を用いることにより従来手法に比べて短い編集スクリプトと生成できることを確認した．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編集スクリプトより短い．
- 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することができる．
- 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することができる．

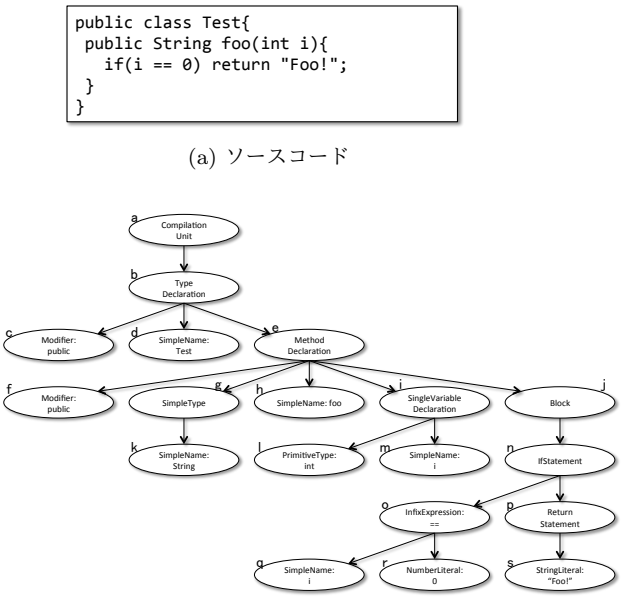
2. 準備

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トークンに対応する値を持つ．例えば，図 1b 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．例えば，図 1b 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，その内容は 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 [9]．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



(a) ソースコード

(b) AST

図 1: AST の例

が同一であることがわかっている [10]．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．

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など，様々な理由で発生する [11], [12], [13], [14] ．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る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．

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 [5][7][8]．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，頂点に格納する新しい値として 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．

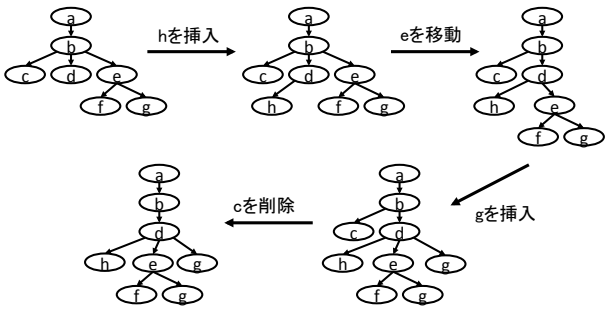
図 2 に AST への編集の例を示す．図 2(a) では，AST において，頂点 h を頂点 d の 1 番目の子として挿入し，頂点 e を頂点 d の 2 番目の子として移動，頂点 g を頂点 d の 3 番目の子として挿入，頂点 c を削除，の順で操作を行っている．このように，AST の編集は対象の AST に操作を適用することで行う．また，この場合の編集スクリプトは図 2(b) である．ただし，ここでは挿入された頂点のラベルと値は省略している．

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．

3.1 既存の編集スクリプト生成手法

編集スクリプトを生成するためにこれまでに多くの手法が研究されている [4][5][6][7] ．編集スクリプトを生成する手法のうち，AST における頂点の追加，削除，更新を考慮する手法が研究されている [15]．その中で最も短い時間で編集スクリプトを生成する手法は RTED[6] 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短い時間で編集スクリプトを生成することができない．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，移動元の頂点を全て削除し移動後の頂点を全て追加するという方法で変更内容を表現することになる．その結果，編集スクリプトは長くなり，理解しづらくなる．

頂点の移動を考慮して短い編集スクリプトを生成することは NP 困難である．そのため，経験則を用いることにより，移動を考慮した上で比較的短い編集スクリプトを生成する手法が研究されている．その内で最も有名なものが Chawathe らのアルゴリズム [7] であり，移動を含む編集スクリプトを効率的に生成することができる．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することができない．



(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

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 [16][17]．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を生成することを重視していない．

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[5] 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生成し出力する．

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生成し出力する．

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．

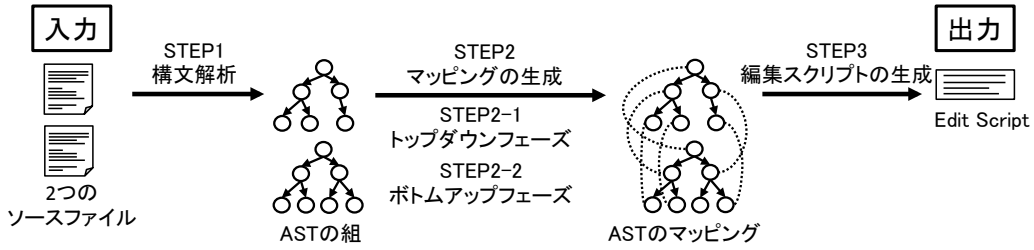


図 3: GumTree の概要

```
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, StringLitteral, 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, NumberLiterral, 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）

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 間において TYPE-2 クローンの検出を行うことで，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 において発見した部分木のマッピングを基準として，ここまででマッピング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 [7] を用いる．

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．緑色でハイライトした部分が挿入に対応する．この例ではソースコードの 2 箇所への挿入が行われている．黄色でハイライトした部分が更新を表している．青色でハイライトした部分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．

```
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

3.3 研究動機

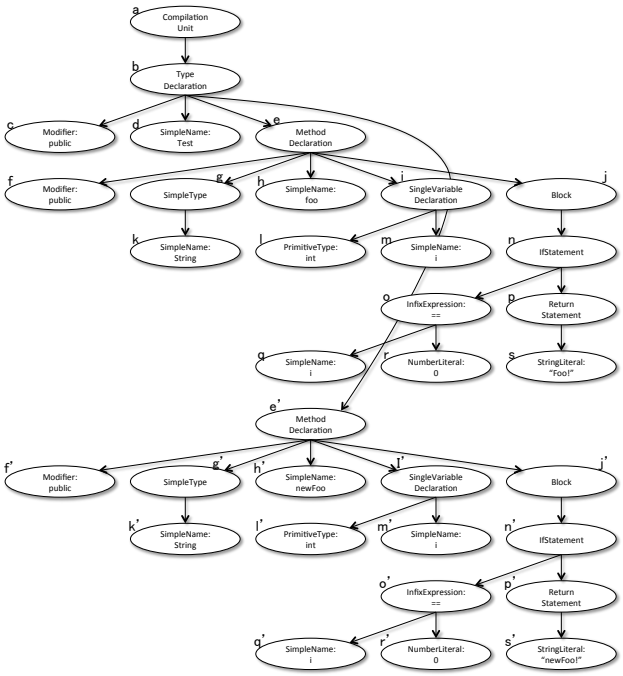
ソースコードの実装において，コピーアンドペーストが頻繁に行われる [9]．しかしながら GumTree は，挿入，削除，更新，移動の 4 種類の操作で編集スクリプトを表現するため，開発者のコピーアンドペーストという 1 つの操作が複数の挿入の操作で表現されることになる．

図 6 のようにコピーアンドペーストを含む変更を GumTree に与えた場合を例として説明する．この変更では，新しいメソッド newFoo が追加されている．メソッド newFoo はメソッド foo と TYPE-2 クローンの関係にある．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，この変更において様々な 15 の新しい頂点が挿入されたことがわかる．

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により，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．

```
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）

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り返しではなく 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 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，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．拡張あるいは追加した 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 間において TYPE-2 のクローン検出を行うことで，2 つの AST に共通して存在する部分木を発見する．さらに，平行してコピーアンドペースト操作の対象となりうる部分木を抽出する．

STEP2-3（C&P 識別フェーズ） STEP2-1 において発見した，コピーアンドペーストによって作成された部分木の候補である，2 つの AST に共通して存在する部分木の組のうち，STEP2-1 と STEP2-2 においてマッピングされていないものを発見し，コピーアンドペーストによって作成された部分木とする．

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．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

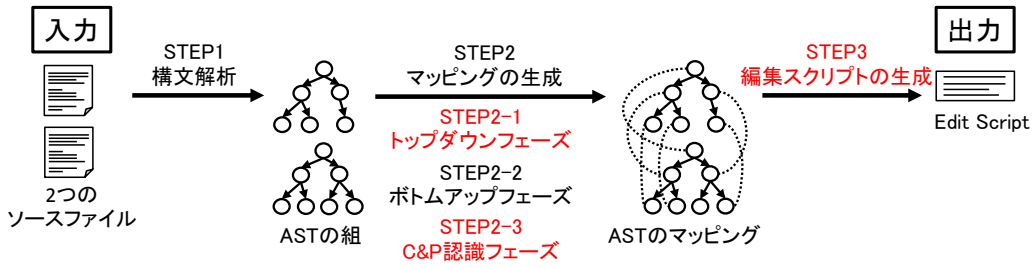


図 7: 提案手法の概要

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 において行われる処理と並行し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．2 つの AST を比較することにより，部分木のマッピングを行う．トップダウンフェーズは以下の手順で実行される．

(1) TYPE-1 および TYPE-2 クローンの関係にある部分木を発見し記録する．
(2) TYPE-1 クローンの関係にある部分木のうち，1 対 1 の関係にある（各 AST に 1 つずつのみ存在している）組をマッピングに追加する．
(3) TYPE-1 クローンの関係にある部分木のうち，1 対多あるいは多対多の関係にある部分木について，最も類似度が高い組のみをマッピングに追加する．ここで用いる類似度は GumTree において定義された類似度である．
(4) ここまででマッピングされていない TYPE-1 クローンの関係にある部分木の組と，TYPE-2 クローンの関係にある部分木の組をコピーアンドペースト操作の候補として記録する．

4.4 STEP2-3（C&P 識別フェーズ）

STEP2-1 では，C&P 操作とみなす変更を決定する．変更後の AST を先行順に探索し，下記の条件を満たす頂点を探す．

- ここまでの STEP においてマッピングに追加されていない．
- コピーアンドペースト操作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全ての子頂点をコピーアンドペースト操作の候補から除外する．

C&P 操作の判定において制約を設けた．提案手法では，TYPE-2 のクローン検出により共通する部分木を発見するため，識別子名やリテラルなどは正規化される．そのため，文法上の理由により，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．例えば，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全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 らのアルゴリズムを用いる [7]．ただし，提案手法では，Chawathe らのアルゴリズムを拡張して，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 を一度ずつ探索することにより編集スクリプトを生成している．

(1) 変更後の AST を幅優先探索することにより挿入，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 を後行順に探索することにより削除を検出する．

幅優先探索の際に，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，その頂点はコピーアンドペーストされた頂点であると判断され，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．

5. 実験

ここでは，提案手法と GumTree の比較を行った実験について述べる．

5.1 準備

3.1 章で述べたように，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 を比較することで，提案手法の有効性を評価する．

提案手法と GumTree において，下記のしきい値を用いた．

- 不要なマッピングを減らすため，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部分木をマッピングする際に用いる類似度の最小値を 0.5 とする．
- 計算時間を短縮するため，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 らの実験において設定された値である [8]．

本実験では，実験対象として 14 のプロジェクトで構成されるデータセットである CVS-Vintage dataset を用いた [18]．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

本実験では，2 つのツールを用いて編集スクリプトを生成し，それらを比較することで評価を行う．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 のプロジェクトのリポジトリから取得する．この実験では各プロジェクトから，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．

以上の手順から，13,699 の変更を取得した．こうして取得した変更を入力とし，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．

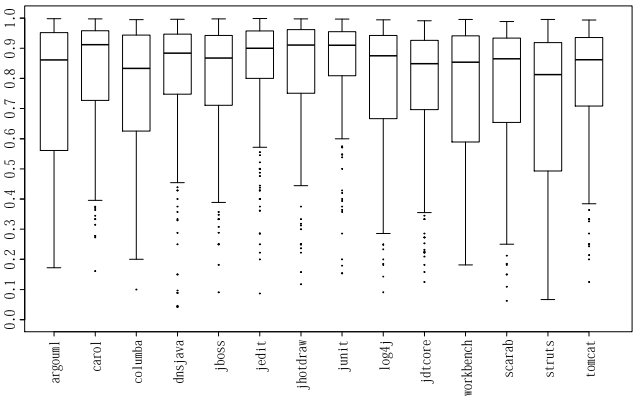


図 8: 編集スクリプトの長さが異なる変更における長さの比較

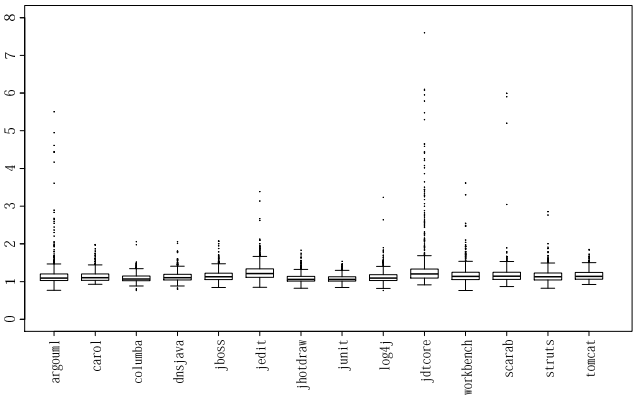


図 9: 全ての変更に対する実行時間の比較

5.3 評価基準

本実験では，各ツールが生成する編集スクリプトの長さと各ツールが編集スクリプトを生成するために要する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．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

13,699 の変更について 2 つのツールの編集スクリプトの長さを比較し，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ることがわかった．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 8 では，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

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

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	230.5	214	116	107	53.75	32	7	3
columba	892	872	128	100	47	39	18	11	5	1
dnsjava	1,632	1,544	192	166.5	75	71	33	18	4	2
jboss	2,876	2,457	223.75	208.5	75.5	61.5	32	21.25	5	2
jedit	2,858	2,853	202	178	79	67	29	24.5	4	2
jhotdraw	1,691	1,686	172.75	144	71.5	64	28.75	18.75	4	2
junit	751	728	141.75	133	74.5	63.5	32.75	25.75	7	2
log4j	3,029	2,827	195	165	75	60	33	24	4	2
jdtcore	13,269	12,628	194.5	168.5	84	65	36	22.5	4	2
workbench	3,651	3,351	195	170	72	55	22	12	4	2
scarab	3,016	2,928	189.5	175.5	83	70	30.5	16.5	5	2
struts	1,486	1,313	159.25	129.25	59	43	33	16	5	1
tomcat	2,152	2,064	162	150.5	54	43	24.5	17.5	4	2

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す．全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ことがわかる．具体的な数値を挙げると，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 9 は，各変更において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．全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 α=0.05 で検定したところ，統計的に優位な差があった．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 10 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．

変更前にあったメソッドを再利用し，図中の点線で囲まれた setReception というメソッドを追加している．setReception の内部では，変数名やメソッド呼び出し文で呼び出すメソッドの名前は異なるが非常に似た処理を行っている．

この変更について，提案手法と GumTree の双方による変更内容の表現は図 10 のようになる．図 10(b) は GumTree による表現である．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わかる．図 10(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わかる．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，変更後のソースコードにおいて，setReception という新しいメソッドが追加されたことがわかる．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わからない．

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．もし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

(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 10: 提案手法によって編集スクリプトを短縮できた例

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が等しい場合

実験では，82%の変更において，提案手法と GumTree の変更の長さは同じであった．これらの変更において，提案手法の生成した編集スクリプトにコピーアンドペーストの操作は含まれていなかった．つまり，コピーアンドペーストの操作とみなせる AST の変更が存在した場合には編集スクリプトは必ず短くなるという結果であった．

理論上はコピーアンドペースト操作を導入することで編集スクリプトが長くなってしまうことはありうる．コピーアンドペーストの後に，コピーアンドペーストされた部分木の全ての頂点において更新が行われた場合，提案手法による編集スクリプトは 1 だけ長くなるはずである．また，コピーアンドペーストの後に，コピーアンドペーストされた部分木のうち 1 つを除く全ての頂点において更新が行われた場合，提案手法による編集スクリプトの長さは GumTree と等しくなるはずである．しかし，提案手法のコピーアンドペーストを含む編集スクリプトが GumTree のコピーアンドペーストを含まない変更以上の長さであった場合がまったくなかった．

この結果に支配的な影響を与えたのは，提案手法においてコピーアンドペースト操作の対象とした部分木を TYPE-1 および TYPE-2 に限定したことだと著者らは考えている．もし TYPE-3 のクローンもコピーアンドペーストの対象にした場合は，コピーアンドペースト後の操作の数が多くなるため，提案手法の編集スクリプトはより長くなってしまうと考えられる．しかしながら，提案手法の編集スクリプト中により多くのコピーアンドペーストが含まれるようになるため，人間がより理解しやすい編集スクリプトになる可能性もある．

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る舞いをする．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。

実験において，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使用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 を比較することで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通りである．

- 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：25220003) の助成を得て行われた．

参考文献

[1] E. W. Myers. An o(nd) difference algorithm and its variations. *Algorithmica*, Vol. 1, pp. 251–266, 1986.

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

[3] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. D. Penta. Lhdiff: A language-independent hybrid approach for tracking source code lines. In *ICSM*, pp. 230–239. IEEE Computer Society, 2013.

[4] M. Hashimoto and A. Mori. Diff/ts: A tool for fine-grained structural change analysis. In *WCRE*, pp. 279–288, 2008.

[5] B. Fluri, M. Wuersch, M. PInzger, and H. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Trans. Softw. Eng.*, Vol. 33, No. 11, pp. 725–743, November 2007.

[6] M. Pawlik and N. Augsten. Rted: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, Vol. 5, No. 4, pp. 334–345, December 2011.

[7] S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change detection in hierarchically structured information. *SIGMOD Rec.*, Vol. 25, No. 2, pp. 493–504, June 1996.

[8] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pp. 313–324, 2014.

[9] M. Kim, L. Bergman, T. Lau, and D. Notkin. An Ethnographic Study of Copy and Paste Programming Practices in OOPL. In *Proceedings of the 2004 International Symposium on Empirical Software Engineering*, pp. 83–92, 2004.

[10] T. M. Ahmed, W. Shang, and A. E. Hassan. An empirical study of the copy and paste behavior during development. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pp. 99–110, 2015.

[11] 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.

[12] 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.

[13] 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, 8 2011.

[14] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software Clone Detection: A Systematic Review. *Information and Software Technology*, Vol. 55, No. 7, pp. 1165–1199, 2013.

[15] P. Bille. A survey on tree edit distance and related problems. *Theor. Comput. Sci.*, Vol. 337, No. 1-3, pp. 217–239, June 2005.

[16] G. Cobena, S. Abiteboul, and A. Marian. Detecting changes in xml documents. In *Inproceedings of the 18th International Conference on Data Engineering*, pp. 41–52, 2002.

[17] R. Al-Ekram, A. Adma, and O. Baysal. diffx: An algorithm to detect changes in multi-version xml documents. In *Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research*, pp. 1–11, 2005.

[18] M. Monperrus and M. Martinez. Cvs-vintage: A dataset of 14 cvs repositories of java software. Technical Report hal-00769121, INRIA, 2012.