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.

行単位の差分情報を考慮した 抽象構文木のノード単位の差分出力

松本淳之介† 　肥後　芳樹† 　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: †{j-matumt,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　バージョン管理システムを用いた開発において，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表現したものを編集スクリプトと呼ぶ．編集スクリプトを出力するツールとして GumTree がある．GumTree は 2 つのバージョンのソースコードを入力として受けとり，抽象構文木のノード単位の挿入・削除・更新・移動といった編集スクリプトを出力する．しかし，GumTree の更新と移動を検知する精度は低く，差分の理解が困難であるという問題がある．GumTree の精度が低い原因の 1 つとして，抽象構文木の情報のみで編集スクリプトを出力しようとすることが挙げられる．そこで提案手法では抽象構文木の情報のみでなく，行単位の差分情報を用いて編集スクリプトを出力する．そしてオープンソースソフトウェアに対して提案手法が有効か実験を行い，提案手法が差分の理解に有効であることを確認した．

キーワード　差分理解，編集スクリプト，GumTree

1. ま え が き

ソフトウェア開発においてバージョン管理システムの導入は必須といえる．バージョン管理システムを用いた開発において，コードレビューや変更が競合した場合に，ソースコードの差分を理解する必要がある．そのような場面において，開発者を支援するためにソースコードの差分を分かりやすく表現する研究が数多く行われている．

ソースコードの差分を表現したものを編集スクリプトと呼ぶ．編集スクリプトを出力する方法として，一般的に Unix の `diff` コマンドを用いることが多い．`diff` コマンドは Myers のアルゴリズム [1] を基に，行単位でソースコードが挿入・削除されたという編集スクリプトを出力する．Git など，多くのバージョン管理システムでこの `diff` コマンドは利用されている．

しかし，`diff` コマンドには問題が 2 つある．1 つ目の問題として，差分を出力する粒度が粗いということである．開発者が理解したい対象の多くはソースコードである．ソースコードの行の一部しか変更されていないにも関わらず，その全てが変更されたものとして出力されてしまい，編集されていないプログラム要素まで編集されたと出力されてしまう．2 つ目の問題として，操作の種類が少ないということである．`diff` コマンドには挿入と削除の 2 種類の操作しかない．ソースコードの編集において，開発者が挿入や削除以外の操作をする場合，開発者の意図した差分を出力することができない．

このような問題を解決する手法として，GumTree [2] がある．GumTree は抽象構文木のノード単位でソースコードの差分を出力する．さらに，ノードの挿入や削除といった操作に加えて，ノードの更新，ノードの移動といった計 4 種類の操作を出力することができる．

GumTree 及び GumTree を改良したもの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，API のコードのサジェスト [3] や Maven のビルドファイルの解析 [4]，自動プログラム修正 [5] であったり，JavaScript のバグのパターンの発見 [6] などに用いられている．

ただし，GumTree にも問題がある．移動と更新を適切に検知することができないということである [7]．実際に著者らが目視でオープンソースソフトウェア (以下 OSS と呼ぶ) の差分を確認したが，更新であるべき変更が削除して挿入されていた．また，移動していないのに移動と出力されることも多数あった．その結果，編集スクリプトの操作の列が長くなってしまっていた．編集スクリプトが長ければ長いほど，開発者はその編集スクリプトを理解することは難しくなると言われている [7] [8] [9]．

そこで本研究では，編集スクリプトを短くすることを指針とし，より理解のしやすい編集スクリプトを出力することを目標とする．更新や移動の精度をあげるためには編集前のノードと編集後のノードのマッチングを適切に行う必要がある．GumTree では抽象構文木の構造の情報のみを用いてマッチングを行なっているが，本研究ではさらに `diff` コマンドを用いた行単位の差分の情報を用いてマッチングを行っていく．`diff` コマンドによって変更されたと出力された行，変更されていないと出力された行の

```
1.  /**
2.   * Removes all mappings from this map.
3.   */
4.  public void clear() {
5.    modify();
6.
7.    nodeCount = 0;
8.    rootNode[KEY] = null;
9.    rootNode[VALUE] = null;
10. }
```

編集前のソースコード

```
1.  /**
2.   * Removes all mappings from this map.
3.   */
4.  public void clear() {
5.    modify();
6.
7.    nodeCount = 0;
8.    rootNode[KEY.ordinal()] = null;
9.    rootNode[VALUE.ordinal()] = null;
10. }
```

編集後のソースコード

挿入　削除　更新　移動

図 1　GumTree の実行例

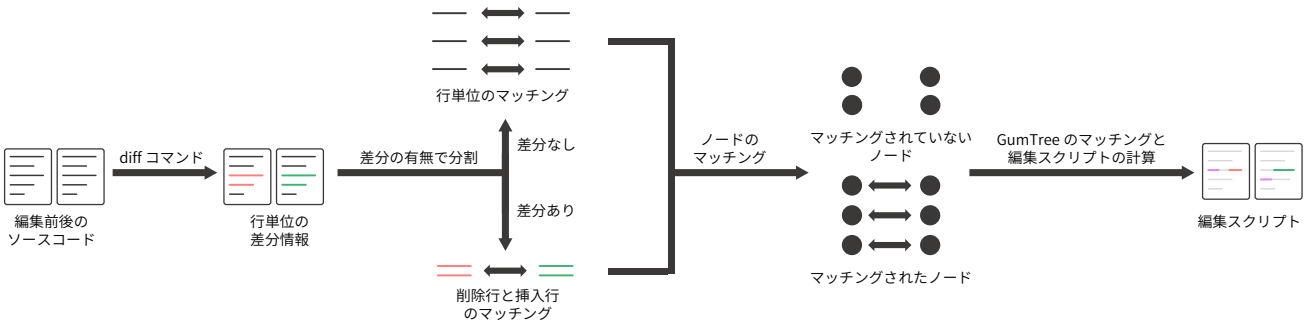

図 2　提案手法の概要

2 つのグループにソースコードを分け，それぞれのグループに対して異なるマッチング手法を用いてノードのマッチングを行う．

GumTree よりも短い編集スクリプトを提案手法が出力していることを確認するため，9 個の OSS のプロジェクトの差分に対して提案手法を実行し，提案手法が有効であることを確認した．また，提案手法で出力された編集スクリプトが理解しやすいものになっていることを確認するために，14 人の被験者に対して，実験を行い，理解しやすいものになっていることを確認した．

2. 抽象構文木の差分検知

2.1 抽象構文木

抽象構文木 (以下 AST と呼ぶ) はソースコードを木構造で表現したものである．1 つのソースファイルに対して 1 つの AST を構築し，AST の各ノードは以下の 3 つで構成されている．

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っている．ただし，根ノードには親が存在していないので，何も保持していない．

ラベル： if 文や変数宣言などといった文法上の型を表している．

値： メソッド名や変数名など各ノードが持つラベル以外の情報である．持つ情報がない場合は null になる場合もある．

2.2 GumTree の差分検知

提案手法は GumTree をベースとしているため，GumTree での AST の差分検知について述べる．GumTree は入力として編集前のソースコードと編集後のソースコードを受けつける．それぞれのソースコードから AST を構築し，それらの木構造の違いを編集スクリプトとして出力する．

GumTree の処理は以下の 2 つで構成されている．

1. ノードのマッチング
2. マッチングの結果を基にした編集スクリプトの計算

ノードのマッチングをすることによって，GumTree は編集前の AST から編集後の AST に対して，どのノードが追加・削除されたか，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という情報を得る．つまり，マッチングされたノードから，GumTree は編集スクリプトを計算することができる．この編集スクリプトの計算に関しては十分に最適化されている [10]．

GumTree が行うノードのマッチングはトップダウンフェーズとボトムアップフェーズの 2 段階で構成されている．まず，トップダウンフェーズでは，入力された 2 つの AST に共通して存在する部分木をマッチングする．その後のボトムアップフェーズでは，トップダウンフェーズでマッチング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マッチングする．マッチングをする際は部分木の類似度を計算し，それが閾値を超えていればマッチングする．

ただし，GumTree のマッチングには問題が残っている．マッチングするべきノードをマッチングさせることができず，必要以上に長い編集スクリプトを出力してしまう．図 1 にその例を示す．このソースコードは Commons-Collections に含まれるあるコミット間の編集スクリプトを GumTree で出力したものの一部である．左右のソースコードを見比べると，8 行目と 9 行目以外は変更されていないにも関わらず，GumTree は変更されたもの

```
1.   /**
2.    * Removes all mappings from this map.
3.    */
4.   public void clear() {
5.     modify();
6.
7.     nodeCount = 0;
8.-    rootNode[KEY] = null;
9.-    rootNode[VALUE] = null;
10.  }
```

編集前のソースコード

```
1.   /**
2.    * Removes all mappings from this map.
3.    */
4.   public void clear() {
5.     modify();
6.
7.     nodeCount = 0;
8.+    rootNode[KEY.ordinal()] = null;
9.+    rootNode[VALUE.ordinal()] = null;
10.  }
```

編集後のソースコード

図 3　行単位の差分

として出力している．例えば 4 行目の public を見比べると同一のコードであるにも関わらず，削除と挿入になっている．これは変更前後の public がマッチングされていないことが原因である．もし変更前後で public をマッチングすることができれば，GumTree は変更の前に存在した public が変更の後にも存在しているものと認識することができ，public が削除・挿入されたとは出力しない．

3. 提 案 手 法

3.1 概　　要

提案手法は GumTree の手法をベースとしている．GumTree では、ノードのマッチングが終わった状態からの編集スクリプトの計算は十分に最適化されているので，提案手法ではノードのマッチングの改善に取り組む．提案手法の概要を図 2 に示す．ノードのマッチングの精度を上げるために，提案手法では AST に関する情報だけでなく，diff コマンドを用いた行単位の差分情報を用いる．GumTree によるマッチングを行う前に diff コマンドから得られた情報を基にマッチングを行う．

diff コマンドで出力される行単位の差分から得られる情報は以下である．

- どの行にどのような行が挿入されたか
- どの行が削除されたか

これらの情報を基に提案手法ではソースコードを，差分がない行と差分がある行の 2 つに分け，それぞれに対して異なるアプローチで探索範囲を狭めたノードのマッチングを行う．

それだけではマッチングされなかったノードも存在するので，その後に GumTree のマッチングを行い，それらの結果から編集スクリプトの計算をする．

3.2 差分がない行

diff コマンドで差分がないと出力された行は，開発者は実際に編集していないものと仮定する．差分がないと出力された行は，編集前後で完全一致する行が存在し，その行に含まれる同じラベルのノードをマッチングしていく．

ノードは必ずしも 1 行に収まっているとは限らない．複数行にまたがるノードは，ノードの先頭の行とノードの最後の行が差分のない行のときに，マッチングを行う．

図 1 で示したソースコードに対して diff コマンドで行単位の差分を計算した結果を図 3 に示す．例えば，編集前の 4 行目と編集後の 4 行目には差分が発生しておらず，完全一致する行である．そこで編集前後の 4 行目に含まれる各ノードのマッチングを行う．このようなマッチングを行うことで 2. 章で説明したような AST の構造のみではマッチングされなかった public などのマッチングが可能となる．

3.3 差分がある行

マッチングされるべきノードは，編集の前後で大きく移動しないと仮定する．その仮定に基づいて，diff コマンドで出力された削除行と挿入行を関連づける．

差分がある行の前後には差分がない行があるので，その行の情報を基に削除行と挿入行を関連づける．連続した複数の行がまとめて削除・挿入されている場合は複数の行をまとめて関連づける．関連づけられた行の中からマッチングするべきノードの探索を行う．

例えば，図 3 の編集前のソースコードの 8 行目と 9 行目には削除が出力されており，その前後の 7 行目と 10 行目は差分がない行である．よって編集前の 7 行目と完全一致する行が編集後のソースコードに存在し，その行は編集後のソースコードの 7 行目である．同様に編集前のソースコードの 10 行目に完全一致する行は編集後のソースコードの 10 行目である．これらの情報から，編集前の 7 行目と 10 行目に挟まれた削除と，編集後の 7 行目と 10 行目に挟まれた挿入を関連づける．削除と挿入の関連づけられたら，その中に含まれるノードのマッチングを行う．

ノードのマッチングは，部分木の類似度を用いて行う．部分木の類似度を用いたマッチングは GumTree でも行われており，部分木の類似度の計算には Jaccard 係数を用いる．

4. 実　　験

提案手法を評価するため，2 つの評価実験を行なった．

表 1　実験対象の OSS

OSS	言語	コミット数
activemq	Java	10,021
commons-collections	Java	3,050
commons-io	Java	2,116
commons-lang	Java	5,263
commons-math	Java	6,317
hibernate-search	Java	7,163
spring-roo	Java	6,132
vue	JavaScript	2,873
jquery	JavaScript	6,363

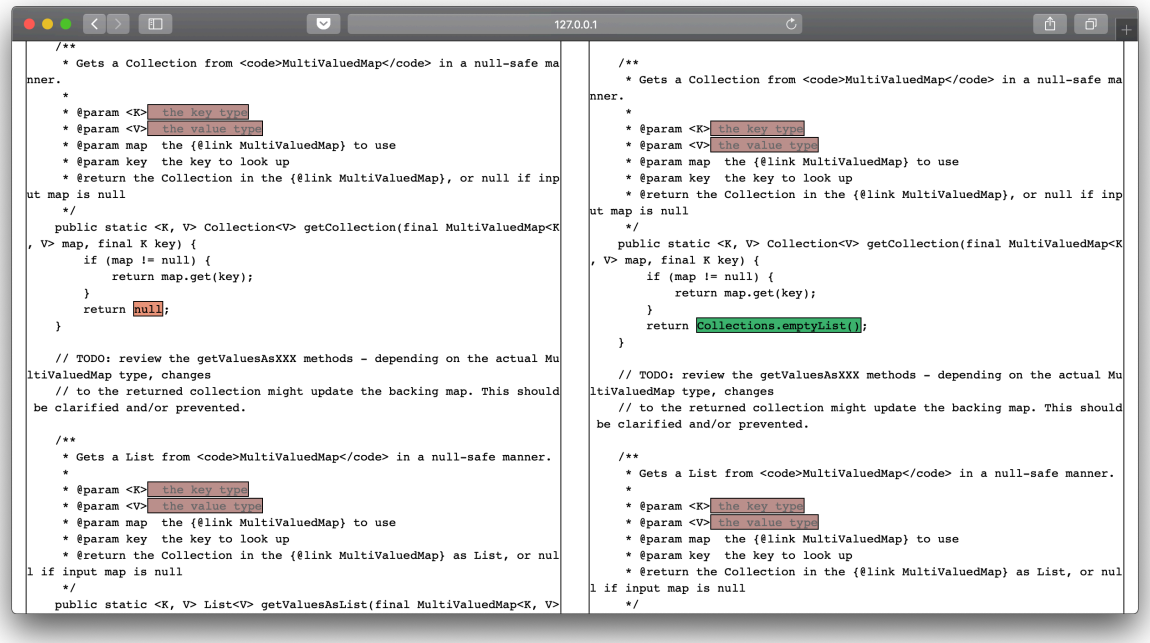

図 4　実験 2 で被験者が編集スクリプトを確認するのに用いたツールの利用例

実験 1：　9 個の OSS を対象に，編集スクリプトが短くなっているか確認する実験

実験 2：　出力された編集スクリプトが実際に開発者にとって理解のしやすいものとなっているか確認するための被験者実験

4.1　実験 1 について

Java, および JavaScript の OSS を対象に提案手法が GumTree よりも短い編集スクリプトを出力できているか確かめる実験を行った．表 1 の OSS の全てのコミットに対して，各ファイルごとに GumTree を実行し，編集スクリプトの長さが 50 以上の Java ファイルおよび JavaScript ファイルを実験対象にした．編集スクリプトの長さが短いものは改善の余地が少ないため，今回は実験の対象に含めていない．

実行した結果を表 2 に示す．どの OSS に対しても，編集スクリプトの長さの合計値と中央値が GumTree と比べて下がっていることがわかる．また，30~80%の差分に対して，提案手法は GumTree より短い編集スクリプトを出力したこともわかる．

得られた結果は正規分布に従っていないため，統計的に優位かどうかの検定には Wilcoxon の符号順位検定を用いた．その結果全ての OSS において，p 値が 0.05 を下回り，GumTree と提案手法の間には統計的な優位差があることが確認できた．

4.2　実験 2 について

提案手法で短くなった編集スクリプトが開発者にとって理解しやすいものになっているか確認する被験者実験を行なった．

被験者は Git と Java を用いた開発の経験が 1 年以上ある 4 人の大学生・9 人の大学院生・1 人の研究者の合計 14 人である．

実験 1 で対象にした commons-math の差分の中から，GumTree が出力する編集スクリプトの長さが 50~200 のもので，GumTree と提案手法の差が大きいもの上位 15 個を対象にした．編集スクリプトの長さが 50~200 のものにしたのは被験者にとって過負荷とならないように調整するためである．

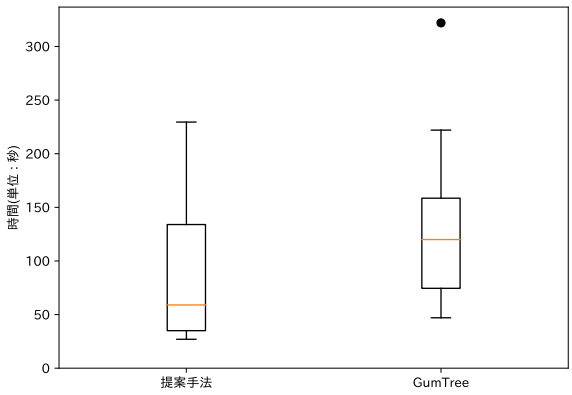

図 5　各差分を理解するのに費やした時間の中央値の箱ひげ図

編集スクリプトの確認には，GumTree に含まれているツールを用いた．このツールは図 4 のように Web ブラウザで視覚的に編集スクリプトを確認することができるツールである．

まず被験者は，実験対象の 15 個の差分に取り組む前に別の 3 つの差分を確認し，ツールの使い方を学ぶ．その後，実験対象の 15 個の差分を 1 つずつ確認し，どのような差分だったかを答え，差分を確認するのにかかった時間を測る．

被験者を X と Y の 2 つのグループに分ける．X のグループは，1 つ目の差分を GumTree で，2 つ目の差分を提案手法で，3 つ目の差分を GumTree で，というように GumTree と提案手法を交互に使って編集スクリプトを確認していく．Y のグループは，1 つ目の差分を提案手法で，2 つ目の差分を GumTree で，3 つ目の差分を提案手法で，というように X と逆になるように確

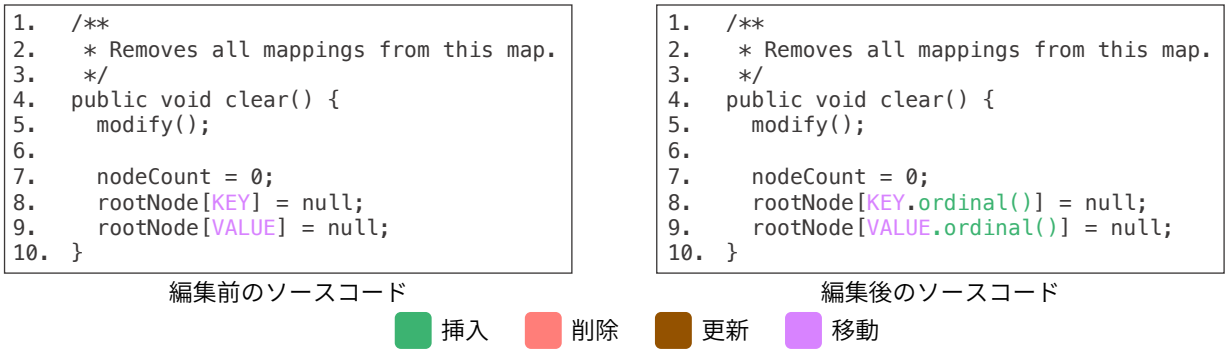

図 6　提案手法の実行例

認していく．

各差分の理解に費やした時間の中央値を箱ひげ図にしたものを図 5 に示す．全体の中央値に関して，提案手法が GumTree の半分の時間になっていることが分かる．

得られた時間の分布は正規分布に従ってないので，マンホイットニの U 検定を用いて，これらが統計的に優位であるか確認した．サンプル数が少ないため，優位差が得られない差分もあったが，全体に関する優位差は得られ，提案手法の方が短い時間で理解することができるといえる．

5. 考　　察

実験 1 では提案手法によって多くの差分で短い編集スクリプトを出力することに成功しており，そのことからも提案手法が有効であることは分かる．commons-math の中にある差分では，GumTree では 19,887 ととても長い編集スクリプトを出力したものもあった．しかし，提案手法ではその差分の長さを 9,742 にまですることができ，およそ 10,000 以上も短い編集スクリプトを出力できている．

短い編集スクリプトを出力できた最大の要因は行単位で差分が生じていない場合に，適切にマッチングをすることができたからだと考えられる．例えば，図 1 について再度考えてみる．GumTree では 4 行目のメソッド宣言について適切にマッチングをすることができておらず，`clear()` を削除して挿入，と出力されている．さらに，メソッド宣言のノードのマッチングができていないということは，5 行目の文の親ノードが変わったと扱われ，5 行目が移動していないのに移動したものと出力されてしまう．図 1 と同一のファイルに対して提案手法を実行した例を図 6 に示す．図 6 は図 1 に比べて理解のしやすい編集スクリプトになったといえる．例えば，4 行目には行単位の差分が発生しておらず，提案手法ではメソッド宣言のノードをマッチングすることができる．その結果，5 行目は編集の前後で同一の親を持つものと扱うことができ，移動していないものとして出力することができる．このように，1 つのノードのマッチングで，他のノードの扱いも変わることが多々あり，結果として短い編集スクリプトを出力できたと考えられる．

実験 2 では提案手法の方が，全体を通して短い時間で差分を理解できることがわかった．理解に要した時間が短く，統計的な有意差がみられた差分は，`@Override` の追加，などといった比較的単純な差分のものが多かった．一方で，提案手法と GumTree の間で統計的な有意差がみられなかったものは，クラス名の変更，メソッドの API の変更といった複数の変更が混じった差分のように，差分として複雑ものであった．このことから，提案手法は複雑な差分よりも単純な差分に対してより有効であることが分かる．

6. 妥当性の脅威

提案手法はプログラミング言語に依存しないため，GumTree が適用可能なプログラミング言語の全てに対して適用可能である．実験 1 では，Java，および JavaScript で書かれたプロジェクトに対してのみ実験を行った．他の言語でも同様の結果が得

表 2　実験 1 の結果

OSS	対象の差分の数	合計		中央値		短い編集スクリプトを出力した割合	
		GumTree	提案手法	GumTree	提案手法	GumTree	提案手法
activemq	5,326	1,001,361	981,435	110	108	8%	33%
commons-collections	1,640	426,383	394,574	122	115	3%	50%
commons-io	782	180,821	176,596	124.5	117.5	8%	34%
commons-lang	2,375	627,934	600,855	130	126	5%	37%
commons-math	4,435	1,276,316	1,223,448	131	128	7%	39%
hibernate-search	4,364	766,425	750,287	105	103	13%	38%
spring-roo	4,880	1,296,941	1,271,632	130	129	12%	42%
vue	553	934,824	832,328	541	503	6%	84%
jquery	2,364	579,349	536,540	103	100	6%	45%

られると予想されるものの，実際に GumTree よりも有用かどうかはわからない．

7. ま と め

本論文では，行単位の差分情報を用いることで，AST ノードのマッチングの精度を上げ，より短く，理解のしやすい編集スクリプトを出力する手法を提案した．提案手法を評価するために，9 個の OSS で実験を行い，全てのプロジェクトで編集スクリプトの長さを短くすることに成功した．また，提案手法が出力する編集スクリプトが理解の助けになることを 14 人の被験者に対して実験を行い，差分理解に費やす時間が全体として減ることを確認できた．

今後の課題としては，Java および JavaScript 以外の言語での実験を行い，提案手法の方が有用かどうかの確認をしていきたい．特に Python のようにインデントが意味を持つようなプログラミング言語の場合，行単位の差分がより発生しにくいと考えられ，Java や JavaScript の差分と比べて提案手法がより有効であると考えられる．

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] E.W. Myers, “Ano(nd) difference algorithm and its variations,” Algorithmica, vol.1, no.1, pp.251–266, Nov. 1986. https://doi.org/10.1007/BF01840446

[2] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, “Fine-grained and accurate source code differencing,” ACM/IEEE International Conference on Automated Software Engineering, ASE ’14, Vasteras, Sweden - September 15 - 19, 2014, pp.313–324, 2014. http://doi.acm.org/10.1145/2642937.2642982

[3] 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,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.511–522, FSE 2016, ACM, New York, NY, USA, 2016. http://doi.acm.org/10.1145/2950290.2950333

[4] C. Macho, S. Mcintosh, and M. Pinzger, “Extracting build changes with builddiff,” Proceedings of the 14th International Conference on Mining Software Repositories, pp.368–378, MSR ’17, IEEE Press, Piscataway, NJ, USA, 2017. https://doi.org/10.1109/MSR.2017.65

[5] J. Yi, U.Z. Ahmed, A. Karkare, S.H. Tan, and A. Roychoudhury, “A feasibility study of using automated program repair for introductory programming assignments,” Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp.740–751, ESEC/FSE 2017, ACM, New York, NY, USA, 2017. http://doi.acm.org/10.1145/3106237.3106262

[6] Q. Hanam, F.S.d.M. Brito, and A. Mesbah, “Discovering bug patterns in javascript,” Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.144–156, FSE 2016, ACM, New York, NY, USA, 2016. http://doi.acm.org/10.1145/2950290.2950308

[7] V. Frick, T. Grassauer, F. Beck, and M. Pinzger, “Generating accurate and compact edit scripts using tree differencing,” 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.264–274, 2018.

[8] G. Dotzler and M. Philippsen, “Move-optimized source code tree differencing,” Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp.660–671, ASE 2016, ACM, New York, NY, USA, 2016. http://doi.acm.org/10.1145/2970276.2970315

[9] Y. Higo, A. Ohtani, and S. Kusumoto, “Generating simpler ast edit scripts by considering copy-and-paste,” The 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE2017), pp.532–542, Oct. 2017.

[10] S.S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, “Change detection in hierarchically structured information,” Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, pp.493–504, SIGMOD ’96, ACM, New York, NY, USA, 1996. http://doi.acm.org/10.1145/233269.233366