編集操作履歴の再生における粗粒度な再生単位

Coarse-grained Frame for Replaying Editing Operation History

桑原 寛明* 大森 隆行†

Summary. In this paper, we propose some methods to transform fine-grained editing operation history into coarse-grained one. Fine-grained history is suitable for investigating the detail of editing operations or source code changes, however, not for grasping the outline of editing history. We describe filters of editing operation history, fusion of editing operations, and criteria to transform into coarse-grained history. These are expected as useful devices to reduce the number of frames in replaying editing operation history.

1 はじめに

ソフトウェア保守におけるプログラム理解などを目的として，開発者が統合開発環境 (IDE: Integrated Development Environment) 上で行った操作を記録する手法が提案されている [1] [2] [3] [4]．記録される操作には，文字の挿入や削除といったソースコードの編集，編集中のファイルの保存，IDE のメニュー項目の選択によるコマンドの実行など様々な操作が含まれる．これにより，開発においてソースコードが変化する過程だけでなく，開発者の操作状況を詳細に追跡し分析できる．ただし，一般的に膨大な数の操作が履歴として記録されるため，人間が記録されたままの履歴を直接見て調査することは現実的ではなく，履歴の効率的な調査を支援する履歴再生器も提案されている [5] [6]．

操作履歴の基本的な再生方法は，個々の操作を単位とする再生である [5]．すなわち，一つの操作を 1 コマとするコマ送り再生である．しかし，操作履歴は開発者が行った操作を再現可能な細かい粒度で記録されている．そのため，ソースコード編集の様子を概観したり，履歴上の着目したいポイントを探したりすることが容易ではなく，再生器の利用者にかかる負荷が高い．対策として，対象が同一で時刻の近い操作を一つにグループ化する再生器 [6] や，特定のコード片に関する操作を抽出する履歴のスライシング [7] がある．

本研究では，操作のフィルタリングや融合あるいは複数の操作のグループ化によって再生単位の数を削減する手法を提案する．再生単位を粗粒度化して一単位の再生で再現されるソースコードの変更を大きくすることで，少ない再生操作でソースコード編集の概略を追跡できる．本手法は [6] と異なり，操作をまとめる基準を柔軟に変更できる．操作の融合や順序変更などによって履歴を整理する手法 [8] もあるが，本手法では操作の時系列順は変更せず，編集対象のメソッドの切り替わりなどの一般的な作業の切れ目と推測される時点で操作をまとめることで再生の負荷を軽減することを目指している．本稿では，予備実験としていくつかの手法を選択して実際の操作履歴に対して適用し，それらの再生単位数の削減効果を測定した結果を示す．

2 OperationRecorder と OperationReplayer

2.1 概要

本稿では，OperationRecorder [1] が記録する操作履歴を対象とする．OperationRecorder は開発者が Eclipse 上で行った様々な操作を記録する Eclipse plug-in であり，Eclipse の標準 Java エディタに操作を記録する機構を組み込んだエディタを提供する．開発者は通常の Eclipse を利用する際と同じようにソースコードの編集

*Hiroaki Kuwabara, 立命館大学情報理工学部

†Takayuki Omori, 立命館大学情報理工学部

```
<normalOperation ccp="NO" file="/test/src/CurrentDate.java" offset="796" time="1341644621456">
<inserted>System.o</inserted>
<deleted/>
</normalOperation>
<menuOperation file="/test/src/CurrentDate.java" label="org.eclipse.ui.edit.text.contentAssist.proposals" time="1341644...
<compoundOperation name="Typing" time="1341644626840">
<normalOperation ccp="NO" file="/test/src/CurrentDate.java" offset="803" time="1341644635707">
<inserted>out</inserted>
<deleted>o</deleted>
</normalOperation>
</compoundOperation>
<menuOperation file="/test/src/CurrentDate.java" label="org.eclipse.ui.file.save" time="1341644635685"/>
<fileOperation file="/test/src/CurrentDate.java" time="1341644635706" type="SAVE"/>
```

図 1　記録された操作履歴の例

を行うことが可能であり，操作はバックグラウンドで自動的に記録される．

記録される操作は，エディタ上で行われる文字列の挿入と削除，カット，コピー，ペースト，リファクタリングやコード整形など Eclipse が行う自動編集，ファイルのオープン，クローズ，保存，メニュー項目の選択やキーボードショートカットによる Eclipse の内部コマンドの実行である．それぞれの操作には操作を行った時刻情報が付せられる．記録された操作の履歴は XML 形式のファイルに出力される．

XML は操作の列と開発者情報からなる．操作を表す主要な要素を以下に示す．

`normalOperation` **要素**　ソースコード編集のために開発者が行った文字列の挿入や削除を表す．対象のファイル，編集位置のオフセット，削除された文字列，挿入された文字列などの情報を持つ．

`compoundOperation` **要素**　Eclipse による自動編集を表す．自動編集は複数回の文字列の挿入や削除によって実現されるため，それぞれの挿入や削除を表す複数個の `normalOperation` 要素を子要素として持つ．自動編集は補完やリファクタリング，コード整形など一通りではないので，その種類を表すラベルも持つ．

`fileOperation` **要素**　編集中のファイルの保存，エディタによるファイルのオープンやクローズを表す．操作対象のファイル，操作の種別などの情報を持つ．

`menuOperation` **要素**　メニューやキーボードショートカットによる Eclipse の内部コマンドの実行を表す．実行されたコマンドを識別する ID，実行時にアクティブなエディタで開かれているファイルなどの情報を持つ．キーバインドを設定している場合，カーソル移動などの操作も `menuOperation` として記録される．

図 1 に記録された操作履歴の例を示す．この例では，“`System.o`” と入力したところでコード補完機能を呼び出し，自動編集によって “`o`” の削除と “`out`” の挿入が行われ，その後ファイルを保存している．

OperationReplayer [5] は OperationRecorder が生成した操作履歴 XML を読み込み，記録された操作を時系列に沿って再生する．履歴を再生中の OperationReplayer の様子を図 2 に示す．操作リストは履歴に含まれる操作を時系列に沿って並べたリストであり，復元ソースコードは操作リストで選択された操作が行われた時点のソースコードを表す．着目する操作は，操作リストから選択，タイムラインバーのダブルクリック，コントローラの操作，といった方法で変更できる．

2.2　再生における問題

OperationReplayer は OperationRecorder が記録した各操作を単位として操作履歴を再生する．一般的に，操作履歴には種類が様々で数も膨大な操作が記録される．すべての種類の操作が一つの履歴に含まれることと，操作の数が多く内容も細かいことが履歴の再生において問題となる．

操作履歴の調査を目的とする再生において，多くの場合すべての種類の操作に着目する必要はない．例えば，コード補完の呼び出しとそれに伴うソースコードの変更を追跡したい場合，`fileOperation` やコード補完の呼び出し以外の `menuOperation` は不要である．不要な操作を含むままでは，着目したい操作に集中することは難しい．

調査に不要な操作の除去だけでは不十分な場合もある．例えば，連続する文字列の入力中にファイルの保存を行った場合，保存前の挿入，保存，保存後の挿入の三つ

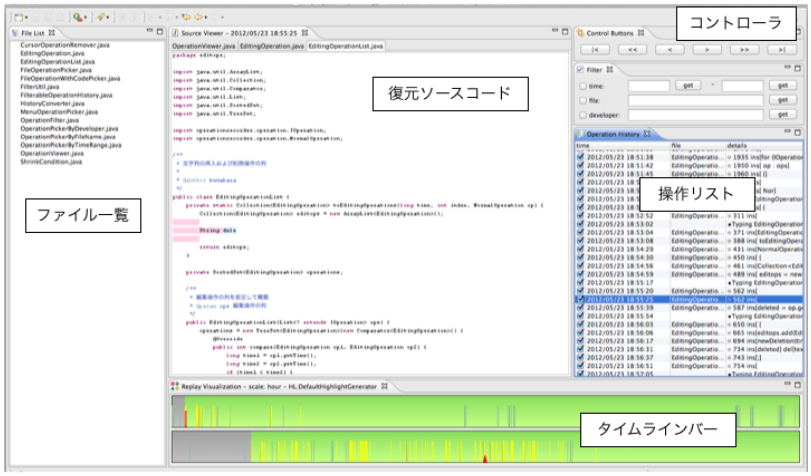

図 2　再生中の様子

の操作が記録される．この時，文字列の挿入に着目するために保存操作を除去しても，挿入操作が二つに分割されたまま残るため再生が不自然に見える可能性がある．

OperationRecorder が記録する操作履歴は粒度が細かいため詳細な調査が可能である一方で，ソースコードの変化や履歴全体を概観することは難しい．詳細を調査したい興味のある箇所を探すために，複数の操作を一つにまとめた大きな単位で再生してソースコードの変化の概略を追跡するようなことは単純にはできない．

これらの問題への対策として，操作履歴のフィルタリング，編集操作の融合，複数の操作を一つの再生単位にまとめるグループ化を提案する．ここで，ソースコードを変更する操作を特に編集操作と呼ぶ．フィルタリングの対象，融合の方法，グループ化の方法は一意ではなく複数の対象や方法があり得る．本稿では予備実験として対象と方法をいくつか取り上げ，適用後の再生単位の数という観点から評価する．

3　操作履歴の変換

3.1　フィルタリング

操作は内容に応じて種類分けされて記録される．操作が行われた時刻，操作対象のファイル，編集位置のオフセット，実行されたコマンドなども併せて記録される．これらの情報に基づいて，目的の操作のみを抽出するフィルタを作成できる．一例として，以下のようなフィルタが挙げられる．

- ソースコードを変更する `normalOperation` と `compoundOperation` を抽出
- ファイル操作を表す `fileOperation` を抽出
- カーソル移動に関する操作を削除
- 特定のファイルを対象とする編集操作を抽出
- 特定の 2 時刻の間に行われた編集操作を抽出

3.2　編集操作の融合

連続する複数の編集操作を操作後の状況が同じになるように一つの編集操作にまとめることを編集操作の融合と呼ぶ．編集操作は `normalOperation` あるいは `compoundOperation` で表されるが，`compoundOperation` は複数の `normalOperation` をまとめたものなので，以下では `normalOperation` の列を考える．`normalOperation` は Eclipse のアンドゥ履歴に基づいて生成される．各 `normalOperation` は 1 回のアンドゥ操作で取り消される文字列の削除と挿入を表

す．削除のみの場合は挿入文字列が空，挿入のみの場合は削除文字列が空である．文字列を選択した状態で新たに入力した場合はいずれも空でない．

アンドゥ履歴に基づくため，タイプミスやその修正も忠実に記録されて再生される．しかし，タイプミスを再現して観察したい場合よりも，最終的に挿入されたひとかたまりの文字列が再現されれば十分な場合が多い．そこで，以下のようにして時系列上で連続する複数の `normalOperation` を一つの `normalOperation` に融合する．

n 個の `normalOperation` が時刻順に並ぶ列に対して，i 番目の `normalOperation` から削除操作 D_i および挿入操作 I_i を抽出する．ここで，削除または挿入操作 O で削除または挿入された文字列を $text(O)$，文字列のオフセットを $offset(O)$，文字列の長さを $len(O)$ と書く．抽出された D_i，I_i の中で $text(D_i)$，$text(I_i)$ が空でないものを i の昇順に並べてテキスト操作リスト L とする．この時，$text(D_i)$ と $text(I_i)$ がともに空でなければ D_iI_i の順に並べる．次に，L の先頭から順に連続する二つの操作 O_i，O_{i+1} を以下のように変更する．操作が削除された場合，リストの前方の要素を後方に詰めるものとする．条件を満たさない場合は何も変更しない．

O_i と O_{i+1} がともに挿入操作の場合 $offset(O_i) \leq offset(O_{i+1}) \leq offset(O_i) + len(O_i)$ ならば，$text(O_i)$ と $text(O_{i+1})$ を適切に合成した文字列を挿入するように O_i を変更し，O_{i+1} を削除する．直観的には，O_i で挿入した文字列の先頭，途中，末尾のいずれかに O_{i+1} で文字列を挿入するならば，全体が一度に挿入されるように変更する．

O_i が挿入操作，O_{i+1} が削除操作の場合 $offset(O_{i+1}) \leq offset(O_i) < offset(O_{i+1}) + len(O_{i+1})$ または $offset(O_i) \leq offset(O_{i+1}) < offset(O_i) + len(O_i)$ ならば $text(O_i)$ と $text(O_{i+1})$ は重複する．この時，重複する文字列を $text(O_i)$ および $text(O_{i+1})$ から取り除く．空文字列になる場合は操作ごと削除する．直観的には，O_i で挿入した文字列（の一部）を O_{i+1} で削除するならば，その文字列（の一部）は初めから挿入されなかったように変更する．

O_i が削除操作，O_{i+1} が挿入操作の場合 $offset(O_i)$ と $offset(O_{i+1})$ が等しく $text(O_i)$ と $text(O_{i+1})$ の先頭 l 文字が等しい場合，O_i は $l+1$ 文字目以降を削除，O_{i+1} は $l+1$ 文字目以降を挿入するように変更する．直観的には，O_i で削除し文字列（の先頭一部）を O_{i+1} で挿入するならば，その文字列（の先頭一部）は初めから削除されなかったように変更する．

O_i と O_{i+1} がともに削除操作の場合 $offset(O_{i+1}) \leq offset(O_i) \leq offset(O_{i+1}) + len(O_{i+1})$ ならば，$text(O_i)$ と $text(O_{i+1})$ を適切に合成した文字列を削除するように O_{i+1} を変更し，O_i を削除する．直観的には，$text(O_{i+1})$ が $text(O_i)$ に接するか $text(O_i)$ の前後をともに含むならば，全体が一度に削除されるように変更する．

例として図1の履歴に含まれる三つの操作，“`System.o`” の挿入，“`o`” の削除，“`out`” の挿入を融合する．初めの2操作は “`o`” の挿入と削除が重複するため，挿入操作を “`System.`” の挿入に変更し削除操作を除去できる．次に，“`System.`” の末尾に接して “`out`” が挿入されるため，二つの挿入操作を “`System.out`” を挿入する一つの挿入操作に融合できる．結果として三つの操作が一つの操作にまとめられる．

3.3 グループ化

現在の OperationReplayer のように一つの操作を再生単位とすると，ソースコード内の離れた領域の変更のような融合できない編集操作は必ず異なる単位として再生されることになる．そこで，再生単位をさらに大きくするために，連続する複数の操作をグループ化して一つの再生単位として扱うことを考える．再生単位を一つの操作に制限せず，融合できない編集操作を一つの再生単位の中に含める．

この場合，どの操作からどの操作までを一つの再生単位にまとめるか決定し，操作履歴中に再生単位の境界を配置する必要がある．しかし，境界の位置は何を意図して再生を行うかに依存するため一意に決められない．例えば，以下のような基準

表 1　基本データ

	開発 A	開発 B
開発期間（日）	18	53
履歴 XML のサイズ (MB)	2.9	9.7
Java ファイル数	9	18
メソッド数	77	106
空行やコメントを含む行数	1848	1890

表 2　再生単位の数

	開発 A	開発 B
オリジナル	10916	47923
`normalOperation` のみ	4774	17604
`compoundOperation` のみ	537	2243
`fileOperation` のみ	1247	1931
`menuOperation` のみ	4251	26089
`copyOperation` のみ	107	56
融合後の `normalOperation`	1185	1814
融合後の `compoundOperation`	47	91
操作対象メソッドの切り替わりごと	416	643

が考えられる．

- 編集対象が異なるメソッドに変わったら境界とする
- 編集対象が異なるブロックに変わったら境界とする
- 二つの編集操作間の経過時間が一定以上であれば境界とする
- 版管理システムへのコミットの直後を境界とする
- 編集対象のファイルの切り替わりを境界とする

本稿では特に再生対象がソースコードであることに着目し，ソースコードの構造に基づくグループ化である一つ目の基準を採用して予備実験を行う．

3.4　予備実験

予備実験として，フィルタリング，編集操作の融合，グループ化によって操作履歴を再生する際の再生単位の数がどの程度削減されるか調査した．フィルタリングあるいは融合を行った場合は履歴に含まれる操作の数が，グループ化を行った場合は履歴中に配置された境界の数が再生単位の数となる．調査対象は Java 言語による 2 種類の小規模なソフトウェア開発（以下，開発 A および開発 B とする）における操作履歴である．それぞれの開発に関する基本的なデータは表 1 の通りである．ファイル数，メソッド数，行数は最後の操作が記録された時点の値である．

基本的なフィルタリングとして特定の種類の操作を抽出するフィルタリングを調査した結果を表 2 に示す．ソースコードを変更する `normalOperation` および `compoundOperation` が全体の 5 割弱を占める．このことから，例えばソースコードの変化を追跡するための再生において，`normalOperation` と `compoundOperation` を抽出すれば再生単位の数を半分以下に削減できることがわかる．

次に，編集操作の融合による削減の効果を調査した．融合されて一つにまとめられた編集操作は基本的には `normalOperation` として表されるが，同一の `compoundOperation` に含まれる `normalOperation` のみが複数の `normalOperation` に融合された場合に限り，それらが同一の `compoundOperation` に含まれるよう融合後に変換している．融合の結果を表 2 に示す．編集操作の数は 1 割から 2 割強ま

で削減されている．変数名や型名の補完，タイプミスの修正などが融合によって一つの編集操作にまとめられるケースが多く見られる．

グループ化に関しては，編集操作対象のメソッドの切り替わりを境界とする手法について調査した．この手法でグループ化すると，連続しており，かつ同じメソッドを対象とする操作が一つの再生単位にまとめられる．グループ化後の再生単位数を得るために，履歴中の各 `normalOperation` および `compoundOperation` について操作対象のメソッドを求めた．各編集操作が行われた時点のソースコードを復元して抽象構文木を生成し，抽象構文木と操作の位置情報から操作対象のメソッドを決定する．操作対象がメソッドの外部の場合，それらの対象はすべて同一であるとみなす．抽象構文木の生成には不完全なソースコードも解析できる Eclipse JDT の構文解析器を用いた．編集中の不完全なソースコードの場合，操作対象を正しく求められない可能性があるが補正はしていない．その他の操作については，現在の OperationRecorder が操作が行われたソースコード上の位置情報を記録しないため，操作対象は直近の `normalOperation` あるいは `compoundOperation` と同じであるとみなす．時系列上で連続する二つの操作の対象メソッドが異なればメソッドが切り替わったと判断する．結果を表 2 に示す．オリジナルの履歴と比較して再生単位数は非常に少ない．一つの再生単位に含まれる操作の数の平均は開発 A が 26.2，開発 B が 74.5 であった．ソースコードを変更する操作に限れば開発 A が 12.8，開発 B が 30.9 であり，ソースコードの編集を大きくまとめることができている．

4 おわりに

本稿では，ソースコード編集の過程の概観や操作履歴上の調査したいポイントの検索を容易にすることを目的として，操作履歴の再生における再生単位数を削減する手法を提案した．手法は，履歴から不要な操作を削減するフィルタリング，連続する複数の編集操作を一つの編集操作にまとめる融合，複数の操作を一つの再生単位にまとめるグループ化からなる．

予備実験として 2 種類の小規模なソフトウェア開発における操作履歴を対象に，いくつかのフィルタリングや融合，グループ化による再生単位数の削減効果を調査し，手法によっては大きく削減できることを示した．今後の課題として，他のフィルタリングや粗粒度化あるいは手法の組み合わせによる削減の実現，ソースコード変更の把握や履歴内検索に対する再生単位数の削減の効果の評価などが挙げられる．

謝辞 本研究の一部は科研費（24700034，24700036）の助成による．

参考文献

[1] 大森隆行, 丸山勝久. 開発者による編集操作に基づくソースコード変更抽出. 情報処理学会論文誌, Vol. 49, No. 7, pp. 2349–2359, 2008.
[2] Romain Robbes and Michele Lanza. SpyWare: A Change-Aware Development Toolset. In *ICSE'08*, pp. 847–850. ACM, 2008.
[3] Lile Hattori and Michele Lanza. Syde: A Tool for Collaborative Software Development. In *ICSE'10*, pp. 235–238. ACM, 2010.
[4] Miryung Kim, Lawrence Bergman, Tessa Lau, and David 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. IEEE Computer Society, 2004.
[5] 大森隆行, 丸山勝久. プログラム開発履歴調査のための編集操作再生器. コンピュータソフトウェア, Vol. 28, No. 4, pp. 371–376, 2011.
[6] Lile Hattori, Mircea Lungu, and Michele Lanza. Replaying Past Changes in Multi-developer Projects. In *IWPSE-EVOL'10*, pp. 13–22. ACM, 2010.
[7] Katsuhisa Maruyama, Eijiro Kitsu, Takayuki Omori, and Shinpei Hayashi. Slicing and Replaying Code Change History. In *ASE'12*, pp. 246–249. ACM, 2012.
[8] 林晋平, 大森隆行, 善明晃由, 丸山勝久, 佐伯元司. ソースコード編集履歴のリファクタリング手法. In *FOSE2011*, pp. 61–70, 2011.