情報処理学会論文誌 Vol.56 No.2 611–626 (Feb. 2015)

コードの編集履歴を用いたプログラム変更の検出

木津 栄二郎[1,a)] 大森 隆行[1,b)] 丸山 勝久[1,c)]

受付日 2014年5月19日, 採録日 2014年11月10日

概要：ソフトウェア開発時のプログラム変更情報を検出し保守者に提示することで，保守者によるプログラムの変更過程の理解を促進できる．このため，ソースコードの 2 つの版から行差分を抽出する手法が提案されている．しかしながら，単一の差分情報に目的の異なる複数の変更が混ざり合っている場合，これらの手法では変更を適切に分離できない．このため，保守者は混ざり合った変更を人手で分離し，個々の変更内容を推測しなくてはならず，その負担は大きい．本論文では，統合開発環境により記録される編集操作履歴を用いることで，個々のプログラム変更を自動検出する手法を提案する．この手法では，編集操作履歴から過去におけるソースコードのさまざまな時点のスナップショットを復元することで，コード片レベルではなくプログラム要素レベルの変更を検出する．さらに，検出された変更を開発者や保守者の方針に応じて集約する仕組みを導入することで，開発者や保守者は集約されたプログラム変更を見ることができる．提案手法を実装したツールを用いた評価実験により，時間的あるいは空間的に集約されたプログラム変更が開発者や保守者のプログラム理解活動を支援できることを示した．

キーワード：ソフトウェア進化，ソースコード変更，プログラム理解，開発環境

Detecting Program Changes from Code Change History

Eijirou Kitsu[1,a)] Takayuki Omori[1,b)] Katsuhisa Maruyama[1,c)]

Received: May 19, 2014, Accepted: November 10, 2014

Abstract: Detecting program changes has been a major activity since it helps maintainers to figure out the evolution of the changed program. For this, several approaches based on line-based differencing have been proposed, which extract differences between two versions of the source code. Unfortunately, these approaches cannot untangle multiple changes for different tasks when they are intermingled with each other in a single difference. Therefore, the maintainers have to untangle them by hand. This task is troublesome and time-consuming. This paper proposes a novel mechanism that automatically detects individual changes of program elements not code fragments by using edit operation history of source code. The edit operations are recorded in an integrated development environment and are used for restoration of various snapshots of source code existing in the past. The mechanism also provides the developers and the maintainers to a chance to select their preferable configuration of aggregation of program changes. Experimental results with a tool implementing the detection mechanism show that the temporally or spatially aggregated program changes facilitate developers' or maintainers' comprehension of program evolution.

Keywords: Software evolution, source code change, program comprehension, development environment

1. はじめに

ソフトウェア保守者は，プログラムを変更する前にそのプログラムをよく理解する必要がある．保守者がどれだけ注意深くプログラムのコードをレビューしたとしても，現在の状態のコードをレビューするだけではプログラムの変更過程を十分に理解することはできない．このため，保守者が着目するコードに対して，それが過去にどのように変更されてきたのかを容易に把握できるようにすることは重要である．

[1] 立命館大学
Ritsumeikan University, Kusatsu, Shiga, 525–8577, Japan
[a)] kitsu@fse.cs.ritsumei.ac.jp
[b)] takayuki@fse.cs.ritsumei.ac.jp
[c)] maru@cs.ritsumei.ac.jp

このような観点から，UNIX diff などの行差分ツールを活用し，版管理システムに記録されているソースコードの2つの版から，プログラムの変更の履歴を抽出することが行われている．しかしながら，これらの変更検出手法には限界があることが Robbes ら [1] によって指摘されている．一般的に開発者は任意のタイミングでソースコードを版管理システムへコミット（格納）できる．このため，単一の版間差分（連続する2つの版の間のコード差分）の中に複数の変更が混ざり合うことが頻繁に発生する [2]．また，単一の版間差分の中にプログラムの変更とリファクタリングが同時に行われることが多いという報告もある [3]．このように，版管理システムに記録される単一の版間差分に対して，必ずしも単一の変更やリファクタリングだけが含まれているわけではない．

変更が混ざり合うことを避けるために，開発環境がエディタで行われた細粒度変更を自動的に記録し，過去に行われた個々の変更を追跡するための新しい手法が提案されている [4], [5], [6], [7]．SpyWare [4] は，統合開発環境である Squeak においてプログラムの構文上の変化をすべて記録する．Syde [5] は，Robbes らの提案する CBSE（change-based software evolution）[1] に基づき，Eclipse 上で行われた細粒度のプログラムの変化を記録する．OperationRecorder [6], [7] では，Eclipse 上で開発時にソースファイルに適用されたすべての編集操作を記録する．

これらのツールを導入することで，開発者や保守者が混ざり合った変更を解きほぐす手間が削減される．しかしながら，これらのツールが提示する変更情報はテキストや単一の構文要素の変化であり，開発者や保守者がプログラムの変更を理解するという観点からは，その粒度が細かすぎる．特にオブジェクト指向プログラムにおけるプログラム変更理解においては，クラス，フィールド，メソッドなどのプログラム要素に関する追加や削除など，開発者や保守者にとって直感的な変更を把握することが重要であるといわれている [8], [9]．

本論文では，OperationRecorder により開発時に記録された編集操作履歴から，オブジェクト指向プログラミング言語 Java のプログラム要素に関する変更情報（本論文ではプログラム変更と呼ぶ）を検出する手法を提案する．本手法では，まず編集操作履歴から過去のソースコードのスナップショット（特定の時刻におけるソースコード）を復元し，構文解析を適用する．次に，構文解析可能なスナップショットのうち，時間的に隣接する2つを比較し，プログラム要素の変更を抽出する．

ここで，スナップショットの復元は編集操作の適用ごとに行われる．つまり，復元されるスナップショットは，開発者の版管理システムへのコミットのタイミングとは無関係であり，多くの場合，単一の版間差分において複数のスナップショットが復元される．このため，スナップショット間において変更が混ざりにくくなる．このように，編集操作履歴からソースコードのスナップショットを復元することで，過去に行われた変更を分離して検出することができる．さらに，本手法では，検出したそれぞれの変更をあらかじめ定義した規則に基づいて集約する．その際，開発者や保守者が集約の方針を与えることができる．これにより，冗長な情報を排除しつつ，さまざまな粒度でプログラム変更を提示できるようになる．

本論文の貢献は次のようになる．

- 編集操作履歴から復元したスナップショットに基づくプログラム変更検出手法の提案
- プログラム変更の検出を自動化するツールの実装
- プログラム変更検出ツールを用いた評価実験結果の提示およびその考察

以降，2章で，提案手法で扱うプログラム変更とその利用場面を示す．次に，3章で，プログラム変更を検出する手法と検出したプログラム変更を集約する手法をそれぞれ説明する．4章では本手法を実装したツールを紹介し，そのツールの簡単な実行結果を示す．5章ではツールを用いた評価実験の結果を示し，その結果に基づく考察を述べる．6章では本研究の関連研究を紹介する．最後に，7章で本論文のまとめを述べる．

2. プログラム変更の検出

本章ではまず，提案手法で検出するプログラム変更について述べる．次に，検出の具体例と利用場面を説明する．

2.1 変更過程理解におけるプログラム変更の検出

一般的に，プログラム変更過程は複数のプログラム変更の適用により達成される．このため，プログラム変更過程の理解に有用な変更情報を提示するためには，次に示す2つの作業が必要であると考えられる．

(1) 個々のプログラム変更を（より正確に）検出する．
(2) 検出したプログラム変更に，その内容（理由や意図など）を付与する．

本論文では，プログラム変更にその内容を付与したものを変更タスクと呼ぶ．作業 (1) と (2) を通して，保守者に対して適切な変更タスクを提示することで，プログラムの変更過程の理解が促進できる．

ここで，提案手法の目的は，過去のプログラムの変更過程におけるプログラム変更を適切な粒度で検出することである．つまり，上記の作業 (1) の自動化である．一般的に，作業 (2) において，適切な変更内容を機械的に付与することは非常に困難である．そこで，我々は，作業 (1) を自動化し，開発者や保守者が作業 (2) に専念できる環境を提供することにより，プログラムの変更過程の理解を支援できると考えた．その際，当然ながら，自動検出されるプログラム変更は，作業 (2) において変更の内容を記述する

表 1 プログラム変更の分類
Table 1 Classification of program changes.

タイプ	意味
AC	クラスやパッケージにクラスを追加する
DC	クラスやパッケージからクラスを削除する
MC	他のクラスやパッケージへクラスを移動する
CCN	クラスの名前を書き換える
AF	クラスにフィールドを追加する
DF	クラスからフィールドを削除する
MF	他のクラスへフィールドを移動する
CFN	フィールドの名前を書き換える
CFT	フィールドの型を書き換える
AM	クラスにメソッドを追加する
DM	クラスからメソッドを削除する
MM	他のクラスへメソッドを移動する
CMN	メソッドの名前を書き換える
CMT	メソッドの戻り値の型を書き換える
CMP	メソッドの引数を書き換える
CMB	メソッドの本体を書き換える

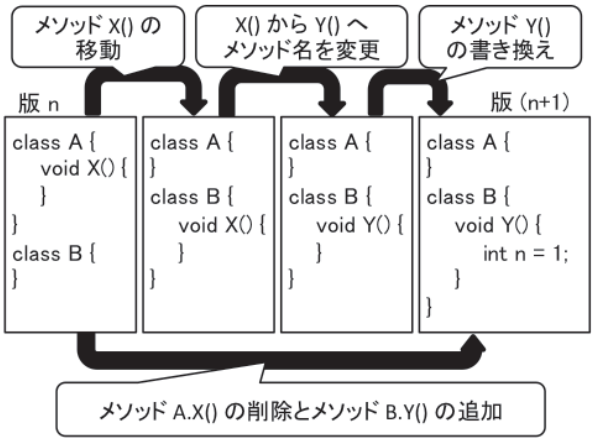


図 1 行差分ツールの限界
Fig. 1 Limitation of a line differencing tool.

開発者や保守者に受け入れられることが重要である．

本手法により検出するプログラム変更の分類を**表 1** に示す．この手法では，文献 [8], [9] と同様に，オブジェクト指向プログラムのソースコードの書き換えを，クラス，フィールド，メソッドに関する変更の種類に分類する．ただし，匿名クラスについてはクラス名が存在しないため検出対象外としている．

2.2 プログラム変更の推測が困難な例

提案手法では，開発者が任意のタイミングで版管理システムにソースコードをコミットするという状況を想定している．このような状況では，個々のプログラム変更が混ざり合うことで，正確なプログラム変更の推測が困難になることがある．ここでは，行差分ツールが版間のプログラム変更を正確に検出できない例を紹介する．**図 1** は以下の 3 つの変更が，版 n のソースコードと版 $(n+1)$ のソースコードの間で行われたことを示している．

(1) メソッド `X()` をクラス `A` からクラス `B` へ移動
(2) クラス `B` のメソッド `X()` を `Y()` に名前変更
(3) クラス `B` のメソッド `Y()` のメソッド本体を書き換え

版 n と版 $(n+1)$ のソースコードに対して UNIX diff を実行した結果は下のようになる．

```
2,3d1
<     void X() {
<     }
6a5,7
>     void Y() {
>         int n = 1;
>     }
```

上から 3 行はメソッド `X()` の削除を表し，下から 4 行はメソッド `Y()` の追加を表している．この結果と上記の 3 つの変更と比較すると，「版 n におけるメソッド `X()` と版 $(n+1)$ におけるメソッド `Y()` が同一である」という情報が欠落していることが分かる．このような情報の欠落により，版間におけるプログラム要素の追跡性が失われてしまい，その版以前のそのプログラム要素の変更過程を理解することができなくなってしまう．

このように，行差分ツールでは個々の変更を分離することが難しい場面がある．このとき，保守者は混ざり合った複数の変更を手動で解きほぐさなければならないが，これは面倒で時間のかかる作業である．また，今回の例では，メソッドどうしの類似度を計算することで，メソッド `X()` とメソッド `Y()` の同一性を別途判定することも考えられる．しかし，本体の一部が書き換えられているメソッドを対象とした場合，類似度の判定基準を設定することは難しく，実際に行われたプログラム変更を再現できる可能性は低い．

本手法では，実際にソースコードに適用された編集操作の履歴を用いることで，上記の例において 3 つのプログラム変更を適切に検出できる（それぞれ，表 1 の **MM**, **CMN**, **CMB**）．さらに，開発者や保守者の方針に従い変更の要約を提示することができる．たとえば，開発者や保守者によっては，3 つの変更が個別に提示されるよりも単一のメソッドの移動（表 1 の **MM**）と提示された方が，その変更内容を付与するうえで有益であるかもしれない．本手法は，このような要求にも応じることができる．

2.3 プログラム変更の利用場面

本手法で利用する OperationRecorder は，Eclipse における Java プロジェクト内のソースコードのすべての編集操作を記録している．このため，プロジェクト内で適用された任意の編集操作を指定することで，その適用直後のソースコードのスナップショットを復元できる．また，各編集操作には，その適用時刻が記録されている．これらの性質を利用して，本手法では以下に示すプログラム変更情報を

出力する．

識別番号（CID） 変更を検出した順番に自動的に割り振られる．

変更の開始時刻と終了時刻 開始時刻は，その変更に関係する編集操作の中で，編集操作が行われた時刻が最も早いものとなる．また，終了時刻は，その変更に関係する編集操作の中で最も遅いものとなる．これらの時刻は変更検出時に自動的に格納される．

変更箇所 変更に関係するプログラム要素を含むソースファイル，パッケージ，クラスの名前を指す．これらの名前はソースコードの変更前後のスナップショットから自動的に取得される．

編集操作 変更に関係する編集操作の集合を指す．1つ前のプログラム変更の直後からこの変更までの間に適用されたすべての編集操作が自動的に格納される．

変更の種類 表1に提示した変更の種類のどれかである．

このプログラム変更情報に対して，我々は，主に2つの利用場面を想定している．

(1) **開発者がソースコードをコミットする場面**

開発者が版 $(n+1)$ のソースコードを版管理システムにコミットしようとした際に，版 n と版 $(n+1)$ の間に行われたプログラム変更を検出して提示する．これにより，版 $(n+1)$ に対するコミットコメント（コミットログ，コミットメッセージ）の入力を支援する．提示するプログラム変更は，版 n および版 $(n+1)$ のソースコードをコミットしたそれぞれの時刻（t_n と $t_{(n+1)}$ とする）を利用して選択する．具体的には，検出したすべてのプログラム変更から，開始時刻が t_n より後ろ，かつ，終了時刻が $t_{(n+1)}$ より前のプログラム変更だけを取り出し，開発者に提示する．たとえば，2.2節の例では，版 $(n+1)$ のソースコードをコミットする際に，開発者に3つのプログラム変更 **MM**，**CMN**，**CMB** の存在が提示される．これにより，開発者は，それぞれのプログラム変更に対して別々に変更の内容を記述すべきかどうかを検討する必要性に気付く．つまり，将来のプログラム変更過程の理解のために，より正確な変更タスクが残される可能性が高まる．

(2) **保守者がプログラムの変更過程を把握する場面**

保守者が特定の Java プロジェクトにおけるプログラムの変更過程を把握しようとした際に，すべてのプログラム変更を検出して提示する．コミットのタイミングやコミットコメントに対する方針が決められていない開発では，版間の差分を単純に利用して検出したプログラム変更は不正確になることが多い．たとえば，2.2節の例において，開発者が3つの変更を実行するごとにコミットしておけば，保守者は版間の差分により3つのプログラム変更を把握できる．あるいは，開発者がコミットコメントとして版間に3つの変更が混在していることを明記しておけば，保守者は3つの変更を把握できる可能性は高くなる．しかしながら，1章で述べたように，単一の版間差分の中に複数の変更が混ざり合っているのが現状である．このような状況において，既存プログラムの変更過程を理解しなければならない場合，より正確なプログラム変更を検出したいと考えるのは自然である．さらには，検出されたプログラム変更にその内容が付与され，より正確な変更タスクが残せれば，将来の保守作業の助けになる．

ここで，提案手法は通常の版管理システムによって記録される版の情報をいっさい利用せずにプログラム変更を検出する．よって，提案手法によるプログラム変更の検出結果は，版管理システムの利用の有無によって変化しない．しかしながら，上記に示した利用場面を実現するためには，版管理システムとの連携は必須である．たとえば，利用場面 (1) では，コミットが行われた時刻の共有やコミットコメント入力時のエディタへのプログラム変更情報の提供が必要になる．また，利用場面 (2) では，保守時に検出したプログラム変更情報を過去のコミットコメントに追加したり，それらを共有したりする機能が必要である．

3. プログラム変更の検出と集約

検出手法の概要を**図 2** に示す．検出手法は OperationRecorder の記録する編集操作履歴を入力とし，2.3節で説明したプログラム変更情報を出力する．また，提案手法は大きく，以下に示す2つのメカニズムで構成されている．

(1) 変更特定メカニズム

(2) 変更集約メカニズム

以下，それぞれのメカニズムについて説明する．

3.1 変更特定メカニズム

変更特定メカニズムは3つの手順で実現されている．

(1) 過去の全状態のソースコードの復元

OperationRecorder により記録された時間的順序に従い，過去の編集操作を適用することで，プロジェクトに存在した Java ソースファイルの全状態のスナップショットを復元する．たとえば，プロジェクト開始から現在に至るまで編集操作を記録している場合，図2における「記録開始」は，プロジェクト開始時点のスナップショットを意味し，「記録終了」は最新のスナップショットを意味する．編集操作には，編集時刻，編集箇所，追加文字列，削除文字列が記録されている．ここで，時刻0および時刻 t における S のスナップショットを S_0 および S_t とすると，S_t は時刻0から時刻 t の間に行われたすべての編集操作を S_0 に適用することで復元することができる．このようにして，各編集操作が適用されるごとに，その時刻のソースコードのスナップショットを取得する．

(2) スナップショットの構文解析とクラス情報の抽出

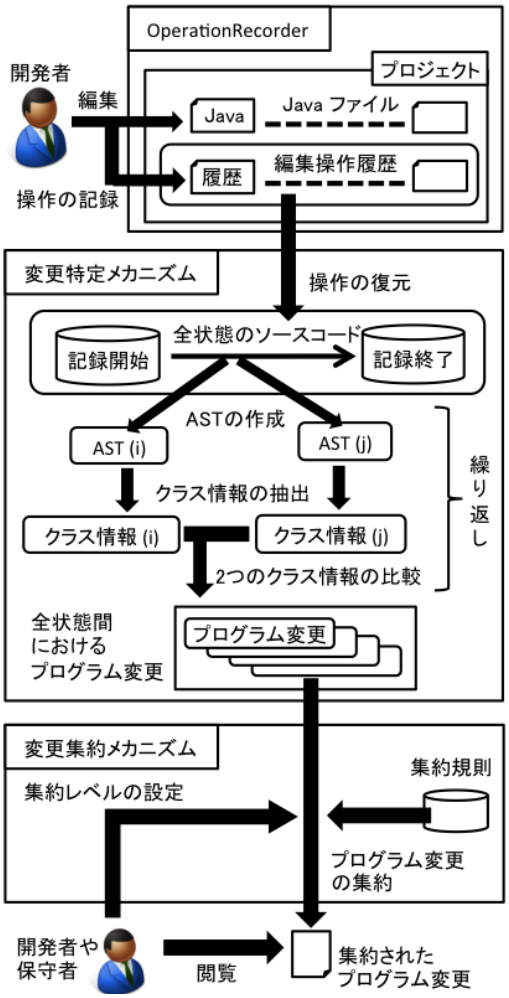


図 2 検出手法の概要
Fig. 2 Overview of the proposed mechanism.

それぞれのスナップショットに対して構文解析を適用する．構文解析が成功した場合，抽象構文木（AST: Abstract Syntax Tree）を作成して，クラスの情報を抽出する．ここでは，S_t に含まれるすべてのクラスに関するクラス情報を集めた集合を I_t と呼ぶ．また，構文解析可能な隣接するスナップショット S_i，S_j（$i < j$）とは，S_i と S_j の両方とも構文解析可能，かつ，$i < k < j$ を満たす構文解析可能なスナップショット S_k が存在しないことを意味する．

(3) クラス情報の比較

隣接する 2 つのスナップショット S_i と S_j のクラス情報 I_i と I_j を比較し，クラス情報の差分からプログラム変更を抽出する．クラス情報とは，各クラスに関する以下の情報を含む．

クラス名 クラスの識別子を指す．クラスが属するパッケージ名を含む完全限定名（fully-qualified name）で表現する．

フィールド集合 クラスに含まれるすべてのフィールドに関する情報の集合を指す．各フィールドは，その名前と型に関する情報を持つ．

メソッド集合 クラスに含まれるすべてのメソッドに関する情報の集合を指す．各メソッドは，その名前，戻り値の型，引数の型のリスト，本体コードに関する情報を持つ．

クラス情報を比較するアルゴリズムの詳細は先行研究 [10], [11] と同様である．また，AST の作成には Eclipse JDT を利用している．

3.2 変更集約メカニズム

本手法では，編集操作履歴から復元されたすべてのスナップショットに対して，構文解析可能かどうかを調べ，構文解析可能な場合にプログラム変更の検出を実行する．このため，同じ種類の変更が繰り返し検出されることがある．たとえば，構文解析が可能であったプログラムに対して，同一のメソッドの本体だけを繰り返し書き換えた場合を考える．この場合，書き換えの途中においてスナップショットが構文解析可能になるたびに，メソッド本体の書き換え（**CMB**）が検出される．

開発者や保守者のプログラム変更過程の理解を支援するという目的においては，同じプログラム要素に対して行われた連続する変更は集約される方がよいことが多い．たとえば，連続する **CMB** が，非常に短時間で行われた書き換えであった場合や同一の式や文に対する書き換えであった場合，これらが別々に提示されることは好ましくないことがある．本手法では，このような **CMB** を集約して，単一の **CMB** として提示する．ここで連続するとは，2 つの変更を時系列に並べたとき，それらの変更の間に他の変更が存在しないことを指す．

本手法におけるプログラム変更の集約は 2 段階で行われる．まず，プログラム変更どうしの結合度に基づき，同一のプログラム要素に関連する一連の変更列から集約の候補となる変更を抽出する．結合の度合いは，連続する 2 つの変更間の時間的距離と空間的距離で決定する．次に，これらの集約候補に対して，集約規則を適用することで，実際にプログラム変更を集約する．

3.2.1 時間的距離と空間的距離

時間的距離とは，ある変更が適用されてから，その次の変更が適用されるまでにどのくらい時間が空いているかを指す．正確には，連続する 2 つの変更において，前側の変更の終了時刻と後側の変更の開始時刻の差である．もし 2 つの連続する変更の時間的距離の値があらかじめ設定された値より小さい場合，それらの変更のペアが集約候補となる．

空間的距離とは，連続する 2 つの変更が行われたソースコード上の位置がどのくらい近いのかを指す．ここで，プログラム要素の追加，削除，移動は，そのプログラム要素全体が変更されたと考えるのが自然である．また，Java において，プログラム要素の名前，引数，型（または戻り値の

表 2 集約規則
Table 2 Aggregation rules prepared in the mechanism.

番号	規則	説明
1	ϕ = **AC**, {**CCN** | **MC**}, **DC**	クラスの追加直後のそのクラスの削除は，変更なしと見なす
2	ϕ = **AF**, {**CFI** | **MF**}, **DF**	フィールドの追加直後のそのフィールドの削除は，変更なしと見なす
3	ϕ = **AM**, {**CMI** | **MM**}, **DM**	メソッドの追加直後のそのメソッドの削除は，変更なしと見なす
4	**AC** = **AC**, [**CCN** | **MC**]	クラスの追加直後のそのクラスの移動や情報の書き換えは，1 つのクラスの追加と見なす
5	**AF** = **AF**, [**CFI** | **MF**]	フィールドの追加直後のそのフィールドの移動や情報の書き換えは，1 つのフィールドの追加と見なす
6	**AM** = **AM**, [**CMI** | **MM**]	メソッドの追加直後のそのメソッドの移動や情報の書き換えは，1 つのメソッドの追加と見なす
7	**DC** = [**CCN** | **MC**], **DC**	クラスの移動や情報の書き換え直後のそのクラスの削除は，1 つのクラスの削除と見なす
8	**DF** = [**CFI** | **MF**], **DF**	フィールドの移動や情報の書き換え直後のそのフィールドの削除は，1 つのフィールドの削除と見なす
9	**DM** = [**CMI** | **MM**], **DM**	メソッドの移動や情報の書き換え直後のそのメソッドの削除は，1 つのメソッドの削除と見なす
10	**MC** = [**CCN** | **MC**], **MC**	クラスの移動や情報の書き換え直後のそのクラスの移動は，1 つのクラスの移動と見なす
11	**MC** = **MC**, [**CCN** | **MC**]	クラスの移動直後のそのクラスの移動や情報の書き換えは，1 つのクラスの移動と見なす
12	**MF** = [**CFI** | **MF**], **MF**	フィールドの移動や情報の書き換え直後のそのフィールドの移動は，1 つのフィールドの移動と見なす
13	**MF** = **MF**, [**CFI** | **MF**]	フィールドの移動直後のそのフィールドの移動や情報の書き換えは，1 つのフィールドの移動と見なす
14	**MM** =[**CMI** | **MM**], **MM**	メソッドの移動や情報の書き換え直後のそのメソッドの移動は，1 つのメソッドの移動と見なす
15	**MM** =**MM**, [**CMI** | **MM**]	メソッドの移動直後のそのメソッドの移動や情報の書き換えは，1 つのメソッドの移動と見なす
16	**CCN** = **CCN**, [**CCN**]	連続で行われたクラスの名前の書き換えは，1 つのクラスの名前の書き換えと見なす
17	**CFN** = **CFN**, [**CFN**]	連続で行われたフィールドの名前の書き換えは，1 つのフィールドの名前の書き換えと見なす
18	**CFT** = **CFT**, [**CFT**]	連続で行われたフィールドの型の書き換えは，1 つのフィールドの型の書き換えと見なす
19	**CMN** = **CMN**, [**CMN**]	連続で行われたメソッドの名前の書き換えは，1 つのメソッドの名前の書き換えと見なす
20	**CMT** = **CMT**, [**CMT**]	連続で行われたメソッドの戻り値の型の書き換えは，1 つのメソッドの戻り値の型の書き換えと見なす
21	**CMP** = **CMP**, [**CMP**]	連続で行われたメソッドの引数の書き換えは，1 つのメソッドの引数の書き換えと見なす
22	**CMB** = **CMB**, [**CMB**]	連続で行われたメソッドの本体の書き換えは，1 つのメソッドの本体の書き換えと見なす

型）は，そのプログラム要素に対して決められた位置（レイアウト上，そのプログラム要素の名前の近く）に存在するため，集約を行うかどうかの基準を用意する必要はないと考えた．以上より，空間的距離による集約は，2 つの変更がどちらもメソッド本体の書き換え（**CMB**）のときのみ活用する．もし空間的距離の値があらかじめ設定されていた値以下となる場合，それらの変更のペアが集約候補となる．

時間的距離はプログラム変更の時刻の差を計算することで容易に取得できる．それに対して，空間的距離の計算はやや複雑であるため，その詳細を説明する．空間的距離を求める際には，前側（時刻が早い方）の変更が行われた直後のスナップショットの AST を利用する．この AST に，それぞれの **CMB** における書き換え範囲（変更箇所を表すスナップショット上のオフセット値と，追加文字列と削除文字列の長さから特定できる範囲）を割り当てる．いま前側の変更が割り当てられた AST 要素（AST 上に存在するノード）の集合を P_b，後側の変更が割り当てられた AST 要素の集合を P_a とおく．ここで，AST の階層において，P_b に含まれるすべての AST 要素を同時に子孫に持つ AST 要素を P_b に対する共通の祖先と呼ぶ．共通の祖先の中で最も深い階層に位置する（AST の頂点から最も距離の遠い）AST 要素を E_b とおく．同様に P_a に対して，最も深い階層に位置する共通の祖先を E_a とおく．次に，このようにして求めた E_b と E_a を結ぶ AST 上のパスを作成する．このパス上に存在するブロックを持つ文（**if**，**while**，**do-while**，**for**，**switch**，**try**）に対応する AST 要素，および，ブロックそのものに対応する AST 要素の数の合計を空間的距離とする．たとえば，E_b と E_a が同じブロック内部に存在する場合，空間的距離は 1 となる．また，E_b が **if** 文の **then** 節に存在し，E_a がその **if** 文の条件式に存在する場合，空間的距離は 2 となる．さらに，E_b と E_a が同じ **if** 文の **then** 節と **else** 節に存在する場合，空間的距離は 3 となる．

3.2.2 集約規則

プログラム変更を集約するためのヒューリスティクス規則を**表 2** に示す．表において，規則は拡張 BNF で記述されている．記述において，コンマはそれを挟む 2 つの変更が連続していることを意味する．波括弧（{ }）はそれによって挟まれる変更の 0 回以上の繰返しを意味し，角括弧（[ ]）はそれによって挟まれる変更の 1 回以上の繰返しを意味する．また，**CFI** は，**CFN** か **CFT** のどちらか一方を指す．**CMI** は，**CMN**，**CMT**，**CMP**，**CMB** のどれか 1 つを指す．

プログラム変更の集約例を，**図 3** を用いて説明する．ここでは，5 つのプログラム変更が事前に抽出され，時間的距離や空間的距離により，前の 3 つの変更から 2 つの集約候補 (**AM**, **CMN**) と (**CMN**, **CMP**)，後ろの 2 つの変更から 1 つの集約候補 (**CMB**, **CMB**) が見つかっているとする．

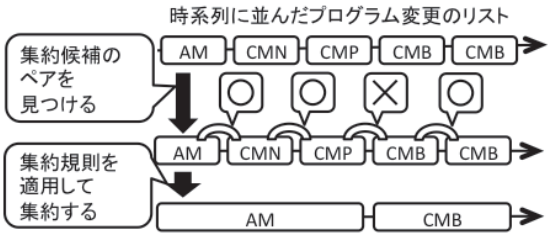


図 3 集約の例
Fig. 3 Example of aggregating program changes.

まず，すべてのプログラム変更を変更時刻が早いものから順に並べた（全体）列を作成する．次に，集約候補のペアを結合する．このようにすることで，集約候補となるプログラム変更の部分列が生成される．この例では，(**AM**, **CMN**, **CMP**) と (**CMB**, **CMB**) という 2 つの部分列が生成される．生成されたそれぞれの部分列に対して，表 2 の規則の右辺のパターンに一致するかどうかを調べ，一致する場合，規則の左辺にあるプログラム変更に置換される．1 つの部分列について，すべてのパターンを適用できなくなるまでパターンの適用を繰り返す．この例では，規則 6 により (**AM**, **CMN**, **CMP**) が **AM** に，規則 22 により (**CMB**, **CMB**) が **CMB** に置換されている．

置換により新しく登場したプログラム変更の種類は，適用された集約規則の左辺の変更と同じである．集約後のプログラム変更が持つその他の情報については，置換された（置換により消去された）プログラム変更に基づき決定される．変更の識別番号と開始時刻には，置換された変更のうち開始時刻が最も早い変更の識別番号と開始時刻が格納される．変更終了時刻については，置換された変更のうち開始時刻が最も遅い変更の終了時刻が格納される．変更箇所と編集操作には，置換されたすべての変更の変更箇所と編集操作を集めたものが格納される．

4. プログラム変更検出ツール

本手法の実現可能性と有用性を確認するために，本手法に基づきプログラム変更を自動検出するツールを，Eclipse のプラグインとして実装した．本ツールは，OperationRecorder によって記録された編集操作履歴と，集約における時間的距離と空間的距離の基準値が記述された設定ファイルを入力として読み込む．変更の抽出および集約を通して得られた結果を，プログラム変更のリストとして出力する．

ここでは，本ツールの出力が妥当であるかどうかを 2.2 節で紹介した例（図 1）を再現することで確認する．このために，まず 2 つの Java ソースファイルを用意し，それらのファイルに図 1 の版 n に存在するクラス A，B を記述した．次に，2.2 節に示した 3 つの変更を手動で行い，OperationRecorder でそれらの編集操作を記録した．メソッドの移動は，Cut & Paste 操作で実現した．

ツールによりプログラム変更を検出した結果を**表 3** に示す．「開始」および「終了」はその変更の開始時刻および終了時刻を指す．「検出結果」はその変更の種類を指す．参考のため，変更に関連するプログラム要素などの情報を「対象」に示した．ここで，CID=1 とは，変更の識別番号（CID）が 1 であるプログラム変更を指す[*1]．

表 3 ツールの実行結果
Table 3 Application of the tool for the demonstration of the example in Fig. 1.

CID	開始	終了	検出結果	対象
1	13:42:16	13:42:16	**AC**	クラス `A`
2	13:42:38	13:42:38	**AC**	クラス `B`
3	13:50:51	13:50:57	**AM**	メソッド `X()`
4	13:54:01	13:54:06	**MM**	メソッド `X()`
6	13:54:37	13:54:37	**CMN**	メソッド `X()`, `Y()`
7	13:54:48	13:54:48	**CMB**	メソッド `Y()`

CID=1，CID=2，CID=3 は，ソースコードの初期状態を準備したときに検出されたものである．つまり，これらの変更は図 1 における版 n の前に行われたプログラム変更を指す．残りの 3 つの変更 CID=4，CID=6，CID=7 が，図 1 の版 n と版 $(n+1)$ の間に行われたプログラム変更である．これらは，それぞれメソッド `X()` の移動（**MM**），メソッド `X()` の名前変更（**CMN**），メソッド `Y()` の本体の書き換え（**CMB**）を指しており，2.2 節に示した変更に正確に対応している．このことより，図 1 に示した 3 つの変更が適切に検出できていることが分かる．

この検出結果に対して時間的距離を 1 分に設定すると，CID=4，CID=6，CID=7 が集約されて，単独のメソッド移動（**MM**）になった．また，時間的距離を 10 秒に設定すると，3 つのプログラム変更は集約されなかった．

このように，本ツールを利用することで，開発者や保守者はさまざまな検出結果を見ることができる．たとえば，時間的距離を 1 分に設定した場合には，1 つのプログラム変更（**MM**）が提示される．また，時間的距離を 10 秒に設定した場合には，ツールにより 3 つのプログラム変更（**MM**，**CMN**，**CMB**）が提示される．

5. 評価実験

本手法の有用性を評価するために実験を行った．実験では，6 個のプロジェクトの開発時の編集操作履歴を用いて，それらのプロジェクトのプログラム変更を検出した．

実験対象プロジェクトの情報を**表 4** に示す．「リバーシ」とはリバーシゲームの開発，「パズル」とは小規模なパズルゲームの開発を指す．「開発者」は計算機科学を専攻する学部生および大学院生を指す．それぞれの開発では，1 人の開発者が単独でプログラムを作成した．また，開発者には，編集操作履歴を記録することだけが伝えられてお

[*1] 本検出手法において CID=5 は内部で消費され，最終結果に表れない．

表 4 プロジェクトの情報
Table 4 Target projects and their characteristics in the experiment.

プロジェクト	開発者	編集操作数	ファイル数	行数	記録開始時刻	記録終了時刻
リバーシ A	学生 A	3,598	6	634	2012/08/10/15:42:45	2012/08/31/01:09:58
パズル A	学生 B	7,610	16	1,245	2008/11/20/20:02:38	2009/02/06/16:50:43
パズル B	学生 C	9,440	28	1,334	2009/01/15/14:34:54	2009/02/02/18:34:37
パズル C	学生 D	16,775	14	2,033	2013/10/03/21:07:47	2013/11/29/23:54:43
パズル D	学生 E	17,299	48	1,890	2009/02/11/19:55:51	2009/03/19/18:54:41
パズル E	学生 F	25,727	26	2,998	2008/11/20/20:53:58	2009/02/10/12:48:59

表 5 検出と集約の結果（集約後/集約前）
Table 5 Results of the detected program changes (with aggregation / without aggregation).

	リバーシ A	パズル A	パズル B	パズル C	パズル D	パズル E
AC	9/9	19/19	34/34	14/14	51/51	26/26
DC	0/0	0/0	2/2	0/0	1/1	0/0
MC	0/0	0/0	0/0	0/0	0/0	0/0
CCN	0/0	0/0	2/2	0/0	11/11	4/4
AF	38/38	64/65	103/104	147/148	221/226	215/225
DF	16/16	4/5	14/15	29/30	94/99	65/75
MF	0/1	0/0	0/0	0/0	6/7	37/38
CFN	0/0	11/15	16/27	23/63	48/78	31/67
CFT	1/1	3/6	9/14	4/15	30/39	13/26
AM	54/54	129/130	169/176	142/143	341/352	263/270
DM	15/15	23/24	36/43	45/46	129/140	73/80
MM	0/0	0/0	7/7	3/3	4/4	27/27
CMN	2/5	3/3	27/64	13/37	39/60	10/28
CMT	1/3	4/9	19/39	10/22	30/61	15/25
CMP	3/4	6/11	27/40	35/59	63/74	44/59
CMB	119/671	160/494	458/1,827	546/2,240	808/2,655	1,092/5,380
Total	258/817	426/781	923/2,394	1,011/2,820	1,876/3,858	1,915/6,330

り，記録した履歴の活用方法に関する情報は与えていない．よって，開発者が実験結果の妥当性に関して影響を及ぼす可能性はないと考えている．

「編集操作数」は，そのプロジェクトの編集操作履歴に含まれる編集操作の総数である．「ファイル数」は，そのプロジェクトから復元したソースファイルの数を指す．「行数」は，そのプロジェクトの編集操作をすべて復元した時点のソースコードの総行数である．「記録開始時刻」は，そのプロジェクトの編集操作の中で最も編集時刻が早い時刻を指す．「記録終了時刻」はそのプロジェクトの編集操作の中で最も編集時刻が遅い時刻を指す．

5.1 定量的評価

ここでは，本ツールを用いることで，どの程度の数のプログラム変更が開発者や保守者に実際に提示されるのかを明らかにする．この実験では，時間的距離や空間的距離の値を設定せず，同じプログラム要素に対して続けて適用された変更は無条件に連続していると見なした．

6 個のプロジェクトに対する検出結果を**表 5** に示す．各行は，プログラム変更の種類ごとに検出されたプログラム変更の数を表している．「/」は区切り記号で，その左側には集約後の変更数を，その右側には集約前の変更数を示した．たとえば，「リバーシ A」において検出された **CMB** の集約前の総数は 671，集約後の総数は 119 である．**Total** は，それぞれのプロジェクトごとに検出された変更の総数を指す．

表 5 より，本ツールでは従来の行差分ツールでは検出が困難な，プログラム要素の移動（**MF**，**MM**）やプログラム要素の名前/型/引数の変更（**CCN**，**CFN**/**CFT**，**CMN**/**CMT**/**CMP**）が検出されていることが分かる．

一方，本実験ではクラスの移動（**MC**）を検出することができなかった．残念ながら，本ツールではパッケージ情報を用いていないため，パッケージをまたぐクラスの移動は検出できない．よって，クラスの移動は内部クラスに限定される．このため，クラスの移動が実験対象プロジェクトには存在しなかったと考えられる．

次に，変更の種類別の変更数の割合に注目すると，すべてのプロジェクトにおいて **CMB** が大量に検出されていることが分かる．また，**AC**，**AF**，**AM** などプログラム要素の追加を行う変更の割合が比較的多いことが分かる．

このように，本ツールを用いることで，過去の変更の傾向を容易に把握することができる．

さらに，それぞれのプロジェクトについて集約前後の検出数を比較すると，その総数は 30.2〜54.5%に減少している．特に，**CMB** については 17.7〜32.3%と大幅な減少を達成している．このことより，集約を適用することによって，開発者や保守者が把握しなければならないプログラム変更の数を抑えることができることが分かる．ただし，削減が適切に行われているかどうかについては，5.2 節で議論する．

5.2 定性的評価

ここでは，特定の開発期間に関する 3 つの事例を取り上げ，検出されたプログラム変更に関する評価を行う．この評価では，ソースコードに適用された変更をあらかじめ調査し，本ツールで検出したプログラム変更と比較した．調査は，著者の 1 人が編集操作再生器 OperationReplayer [12] を用いて開発過程を閲覧しながら行った．

ここで，評価に用いた事例の選定基準について述べる．まず，実際の開発規模に近いプロジェクトという観点から，編集操作数の少ないプロジェクト「リバーシ A」，「パズル A」，「パズル B」を実験対象から除外した．その結果，編集操作数の多いプロジェクト「パズル C」，「パズル D」，「パズル E」を実験対象とした．

さらに，5.1 節の定量的評価において，本手法の提案する変更の種類の中で **CMB** が集約の適用によって最も減少することが分かった．このため，連続する **CMB** に対する集約（表 2 の規則 22）に関して，事例 1 と事例 2 でその効果を考察することにした．その際，紙面の都合から，単一のプログラム要素に対して **CMB** が 30 個程度連続する開発期間を探した．その結果，**CMB** が 29 個連続する開発期間が 4 つ，31 個連続する開発期間が 4 つあった（30 個連続する開発期間は存在しなかった）．結果として，より連続数の大きい 31 個の方の 4 つの開発期間から，無作為に 2 つの開発期間を選んだ．

事例 1 と事例 2 のどちらも **CMB** を対象としているため，事例 3 では **CMB** 以外のプログラム変更を取り上げることにした．その際，5.1 節の定量的評価において，**CMB** の次に検出数が多い（検出数が 2 番目の），メソッドの追加（**AM**）に着目し，**AM** に関する集約（表 2 の規則 6）を事例 3 で取り上げることにした．

事例 3 では．実験対象とした 3 つのプロジェクトのうち，事例 1 と事例 2 で利用していないプロジェクト「パズル C」に着目した．まず，**AM**，**MM**，**CMN**，**CMT**，**CMP** の検出数を検出されたファイルごとに集計し，検出数の合計が最も多いファイルを実験対象ファイルとして選択した．さらに，そのファイルの中で，**AM** に関する集約規則が最も多く一致する開発期間を選んだ．ただし，この集約規則に一致した事例候補の中で，**CMB** が 2 つ以上連続する開発期間はあらかじめ除外した．これは，**CMB** が大量に連続する開発期間を事例に選んでしまうことで，事例 3 の評価の観点と事例 1 や事例 2 の評価の観点が同じになってしまうことを避けるためである．ここで，**AM** に関する集約に直接関係するプログラム変更の数は 5 個であった．事例 3 では，これらのプログラム変更に関係あるプログラム変更として，全部で 15 個のプログラム変更で構成されている開発期間を取り上げる．

この定性的評価では，時間的距離および空間的距離の概念を変更の集約に導入することの意義を明らかにするため，以下に示す 2 つの集約方針を採用した．

1M 時間的距離の値を 1 分に設定

2S 空間的距離の値を 2（AST 要素数を 2 個）に設定

本ツールによるプログラム変更の検出結果を**表 6**，**表 7**，**表 8** に示す．ここでは，まず表 6 を例にとり，実験結果の見方を説明する．表の「CID」，「開始」，「終了」，「検出結果」については，表 3 と同じである．「1M」，「2S」は，それぞれの集約方針での検出結果を示している．たとえば，表 6 の「1M」の欄を見てみると，方針 1M による集約において CID=42 と CID=43 の **CMB** が 1 つの **CMB** に集約されている．「変更タスク」は，人手により検出したプログラム変更の内容を簡潔に示したものである．

以降，3 つの事例を用いて，本手法によって検出されたプログラム変更と，事例の調査によって得られた変更タスクとを比較することで，本手法の有用性を評価する．なお，以降の文章中，CID=1〜3 とは，CID=1，CID=2，CID=3 のように連続するプログラム変更の列を指す．

5.2.1 事例 1

事例 1 では，プロジェクト「パズル D」において，ソースファイル PuyoClient.java を約 40 分間編集した開発期間を取り上げる．この編集では，フレーム，ラベル，パネルなど，ゲームアプリケーションのウィンドウを構成する GUI コンポーネントの作成や，それらの GUI コンポーネントのオプションの設定などに関する処理の記述が行われていた．同時に，例外処理の移動や変数宣言の移動，一時変数の置き換え，依存関係がないと思われる文どうしの入れ替えなどの，コードの可読性を高めるためのコード変換も行われていた．事例 1 における検出結果を表 6 に示す．

まず，1M による検出結果に関して考察する．CID=42〜43 に関しては，検出結果と変更タスク 1 が一致しており，ツールが適切にプログラム変更を検出しているといえる．一方，CID=44〜45 に着目すると，変更タスク 2 と変更タスク 3 が単一のプログラム変更として検出されている（検出漏れが発生している）ことが分かる．同様の現象は，CID=46〜52，CID=54〜55，CID=56〜63，CID=64〜72 にも見られる．このような検出漏れは，プログラム変更過程理解の支援につながらない．たとえば，ソースコードを

表 6 事例 1 における検出結果
Table 6 Experimental results of the detected program changes on Case 1.

CID	開始	終了	検出結果	1M	2S	変更タスク
42	19:50:26	19:50:37	**CMB**	**CMB**	**CMB**	1. 例外処理の修正
43	19:50:44	19:50:44	**CMB**		**CMB**	
44	19:55:49	19:57:25	**CMB**	**CMB**	**CMB**	2. フレームの作成，ゲームボードの作成と削除
45	19:58:21	19:59:06	**CMB**			3. ラベルの作成と登録
46	20:00:10	20:00:11	**CMB**	**CMB**		4. 例外処理の移動
47	20:00:19	20:00:21	**CMB**			
48	20:00:31	20:00:46	**CMB**			5. ラベルの設定
49	20:00:57	20:00:57	**CMB**			6. 変数の宣言文を移動
50	20:01:03	20:01:03	**CMB**		**CMB**	
51	20:01:19	20:01:26	**CMB**		**CMB**	7. ラベルの設定文の引数にキャストを追加
52	20:01:35	20:01:35	**CMB**			8. ローカル変数を null で初期化
53	20:02:45	20:16:00	**CMB**	**CMB**		9. フレームの設定文を追加
54	20:17:47	20:24:27	**CMB**	**CMB**		10. パネルの作成
55	20:24:44	20:24:44	**CMB**			11. フレームの設定文に引数を追加
56	20:26:25	20:26:32	**CMB**	**CMB**		12. フレームが持つフィールドの値を一時変数に代入
57	20:26:54	20:26:54	**CMB**			
58	20:26:57	20:27:09	**CMB**			13. パネルをフレームへ登録
59	20:27:14	20:27:14	**CMB**			14. タスク 12 の一時変数を名前変更
60	20:27:16	20:27:16	**CMB**			
61	20:27:19	20:27:19	**CMB**			
62	20:27:27	20:27:27	**CMB**			15. フレームへ登録する文を入れ替え
63	20:27:28	20:27:28	**CMB**			
64	20:32:12	20:32:12	**CMB**	**CMB**		16. フレームの設定文を移動
65	20:32:14	20:32:14	**CMB**			
66	20:32:57	20:32:58	**CMB**			17. フレーム，パネル，ラベルの構成関係を修正
67	20:33:01	20:33:01	**CMB**			
68	20:33:13	20:33:13	**CMB**			
69	20:33:17	20:33:25	**CMB**			
70	20:33:26	20:33:26	**CMB**			
71	20:33:30	20:33:30	**CMB**			
72	20:33:32	20:33:40	**CMB**			

コミットする場面において，ツールが CID=56～63 を単一のプログラム変更として提示してしまうと，実際に行われた変更 12，13，14，15 の存在を開発者や保守者が見逃してしまう可能性が高くなる．

事例 1 において，1M による集約は検出漏れがあったものの，単一のタスクを複数のタスクと見なすこと（誤検出）はなかった．よって，変更タスクを大まかにとらえていくという利用方法に対しては，有用であるといえる．今回の結果をもう少し詳しく見てみると，変更タスクの切替えは 1 分程度の中断前後で必ず発生している．一方，1 分未満の中断でも頻繁に変更タスクを切り替えており，これが検出漏れを発生させた原因であると考えられる．一般的に，開発者がソースコードの編集を数十分以上中断した場合，中断の前後で変更タスクが切り替わっていると考えるのが自然である．しかしながら，数十秒から数分の中断から，変更タスクが切り替わったかどうかを判断することは難しい．今回のような開発期間において検出結果の妥当性を高めるためには時間的距離を短く設定することを検討すべきである．ただし，時間的距離を極端に小さく設定すると，誤検出の著しい増加を招く恐れがあることに注意する必要がある．

次に，2S による集約の集約結果を見てみると，変更タスク 1 が別々のプログラム変更（CID=42 と CID=43）として提示されてしまっている．同様に，変更タスク 6 も分離されている．一方，CID=51～72 のように，複数の変更タスクを含む非常に大きなプログラム変更を提示している点も問題である．このように，2S による集約では，検出漏れと誤検出が両方とも発生している．特に，誤検出は開発者や保守者を混乱させる可能性が高く，空間的距離の導入の反例となっている．

5.2.2 事例 2

事例 2 では，プロジェクト「パズル E」において，ソースファイル Field.java 内のゲームロジックの修正を約 17 分間行っている開発期間を取り上げる．この事例における編

表 7 事例 2 の実験結果
Table 7 Experimental results of the detected program changes on Case 2.

CID	変更開始	変更終了	検出結果	1M	2S	変更タスク
1001	20:01:03	20:01:18	**CMB**	**CMB**	**CMB**	1. ループ構造の修正
1002	20:03:23	20:03:23	**CMB**	**CMB**	**CMB**	2. 配列の添え字の値を修正
1003	20:03:27	20:03:28	**CMB**			
1004	20:03:32	20:03:32	**CMB**			
1005	20:03:37	20:03:37	**CMB**			
1006	20:04:27	20:04:27	**CMB**		**CMB**	3. タスク 1 を手動で取り消し
1007	20:04:34	20:04:34	**CMB**			4. タスク 2 を手動で取り消し
1008	20:04:35	20:04:35	**CMB**			
1009	20:04:38	20:04:38	**CMB**			
1010	20:04:40	20:04:40	**CMB**			
1011	20:10:27	20:10:27	**CMB**	**CMB**		5. タスク 1 を再度行う
1012	20:10:44	20:10:44	**CMB**		**CMB**	6. 配列の添え字の値を修正
1013	20:10:57	20:10:57	**CMB**			
1014	20:11:01	20:11:01	**CMB**			
1015	20:11:06	20:11:06	**CMB**			7. ロジックの一部をコメントアウト
1016	20:11:30	20:11:31	**CMB**		**CMB**	8. プリント文の挿入
1017	20:11:32	20:11:32	**CMB**			
1018	20:11:36	20:11:36	**CMB**			
1019	20:11:51	20:11:51	**CMB**			9. タスク 8 を手動で取り消し
1020	20:13:12	20:13:18	**CMB**	**CMB**	**CMB**	10. if 文の条件式を書き換え
1021	20:13:29	20:13:29	**CMB**			
1022	20:13:30	20:13:30	**CMB**			
1023	20:14:07	20:14:07	**CMB**		**CMB**	11. タスク 1 を再度手動で取り消し
1024	20:14:12	20:14:12	**CMB**			12. タスク 6 を手動で取り消し
1025	20:14:15	20:14:15	**CMB**			
1026	20:15:17	20:15:17	**CMB**	**CMB**		13. タスク 1 を再度行う
1027	20:15:19	20:15:21	**CMB**			14. イテレータの更新タイミングを修正
1028	20:15:29	20:15:29	**CMB**		**CMB**	
1029	20:16:22	20:16:22	**CMB**		**CMB**	15. 配列の添え字の値を修正
1030	20:16:26	20:16:26	**CMB**			
1031	20:17:41	20:17:59	**CMB**	**CMB**		16. 配列の値を書き換える文を追加

表 8 事例 3 の実験結果
Table 8 Experimental results of the detected program changes on Case 3.

CID	EID	開始	終了	検出結果	1M	変更タスク
366	A	16:55:58	16:55:58	**CMB**	**CMB**	1. パズルピースを操作するメソッドから操作結果を受け取り,
367	A	16:56:07	16:56:10	**CMB**		操作結果を条件式に含めた空の if 文のブロックを挿入
368	A	16:56:12	16:56:12	**CMB**		
369	B	16:57:23	16:57:34	**AM**	**AM**	2. ゲーム中でプレイヤが操作するパズルピースを
370	B	16:58:28	16:58:57	**CMB**		生成するメソッドの作成
371	B	16:59:00	16:59:00	**CMP**		
372	B	16:59:08	16:59:08	**CMT**		
373	B	16:59:08	16:59:08	**CMT**		
374	A	16:59:20	16:59:26	**CMB**	**CMB**	3. タスク 1 で作成した if 文のブロックの中で,
375	A	17:00:24	17:00:24	**CMB**		タスク 2 で作成したメソッドを呼び出してパズルピースを生成し,
376	A	17:00:28	17:00:28	**CMB**		新たに作成したローカル変数で受け取る
377	A	17:00:38	17:00:38	**CMB**		
378	A	17:00:45	17:00:53	**CMB**		4. 生成したパズルピースをプレイヤが操作するパズルピースに設定
379	A	17:01:06	17:01:06	**CMB**		5. タスク 3 で作成したローカル変数をフィールドに変換
380	C	17:01:11	17:01:28	**AF**	**AF**	

集を分析してみると，ループ構造の修正，配列の添え字の値の書き換え，ロジックの一部のコメントアウトなど，さまざまな種類の修正が混ざり合っていた．また，それらの修正を取り消したり再度行ったりするなど，実行結果を確認しながらの試行錯誤が行われていた可能性が高い．事例2における検出結果を表7に示す．

事例1とは異なり，事例2においては1Mによる集約よりも2Sによる集約の方が有用だといえる．特に，2Sによる集約が，変更タスク2と変更タスク10に対応するプログラム変更を適切に検出している点が特徴である．この事例において修正されたソースコードには，**for** 文や **if** 文を用いたループや分岐の処理が多く含まれていた．このため，変更タスクが切り替わると同時に編集箇所も切り替わり，それらの空間的距離が自然に離れた．これにより，2Sによる集約がタスクの切り替わりをうまくとらえることができたと考えられる．

しかし，一方でCID=1023～1027の集約では，単一のプログラム変更が4つの変更タスク11，12，13，14を含んでいる．同時に，タスク14は2つのプログラム変更に分離されている．これらの検出漏れと誤検出により，集約結果の良し悪しがコードの構文に左右されてしまうという問題も明らかになった．これらのプログラム変更を適切に集約するためには，構文上の距離だけでなく意味的な観点での距離も考慮にいれるなど，2つの変更箇所の間の空間的距離の計算方法について検討する必要がある．

1Mによる集約では，16個の変更タスクのうち，正しく検出されたプログラム変更は2個であり，検出漏れが多い．しかしながら，誤検出はなく，時間的距離に基づく集約には一定の効果があると見なせる．事例1と同様に変更タスクを大まかにとらえていく場合に有用だといえる．

5.2.3 事例3

事例3では，プロジェクト「パズルC」において，ソースファイルPuyoPanel.javaを約5分間編集した開発期間を取り上げる．この開発期間では，ゲーム中でプレイヤが操作するパズルピースを構築するメソッドを新たに作成している．事例3における検出結果を表8に示す．この事例では，**CMB** 以外の集約結果を考察するため，空間的距離による集約結果を省略し，1Mによる集約結果のみ示した．また，本事例では複数のプログラム要素が変更対象として登場するため，それらを区別するための識別番号（EID）を記載した．

まず，検出が成功した例として，変更タスク1と2を説明する．変更タスク1では，操作中のパズルピースがゲームフィールドに固定されたときの処理を書くための **if** 文をメソッドA（`run()`）の本体に作成している．本ツールは，CID=366～368が単一のプログラムに変更として集約しており，その結果は変更タスク1と一致する．タスク2では，パズルピースを生成して返すメソッドB（`createPuyoPair()`）を作成し，その本体，引数，戻り値の型を順番に書き換えている．これはオブジェクトを生成して返すだけの単純なメソッドであるが，その作成過程では異なる種類の変更が検出されてしまう．このような複数のプログラム変更を別々に理解することは，開発者や保守者にとって煩雑である．本ツールは，CID=369～373を単一のプログラム変更 **AM** に集約しており，その結果は変更タスク2と一致する．以上より，変更タスク1と2に関しては，適切にプログラム変更が検出されていることが分かる．

これらの結果に対して，変更タスク3，4，5に関しては，本ツールがプログラム変更を適切に検出できているとはいえない．CID=374～379における集約結果と変更タスクを比較すると，3つの変更タスクが単一のプログラム変更として検出されていることが分かる．さらに悪いことに，変更タスク5が2つのプログラム変更に分離されている．実際には，CID=379とCID=380のプログラム変更は1分以内に行われているため，これら2つのプログラム変更が集約されることが好ましい．しかしながら，これら2つのプログラム変更はそれぞれメソッドAとフィールドC（`nextPuyoPair`）に対して別々に実行されている．本手法では，異なるプログラム要素に対する個々の変更を単一の変更として集約することができない．このような本手法の限界により，変更タスク5を適切に検出することは不可能である．このような点から手法の改良に関する検討は必須である．

5.3 妥当性への脅威

ここでは，本実験の評価結果によって得られた有用性を脅かす脅威について，外的妥当性と内的妥当性に分けて議論する．

5.3.1 外的妥当性

本実験で対象としたプロジェクトでは，現実の開発現場でのソフトウェア開発経験のない学生が個人で開発したソースコードを扱っている．また，プロジェクトの規模もかなり小さく，現実的な開発とはいえない．このため，実際の開発現場から得られた編集操作履歴を利用した場合，5.1節と5.2節で示した実験結果と大きく異なる恐れがある．

また，定性的評価において選定した事例は実際の開発期間のごく一部である．別の開発期間では，より正確，あるいは，より不正確なプログラム変更の検出が行われていた可能性は否定できない．よって，別の事例を取り上げることで，有用性の主張が変わる可能性がある．

さらに，本実験で利用した編集操作履歴の中には開発時に行われた編集の過程を再生できない部分がある．本手法では，編集操作の再生に失敗してからファイル操作によって正常なソースコードを取得するまでの間は，すべてコン

パイルエラーと見なしてプログラム変更を検出していない．このため，再生不能の状態から復帰してから初めて変更を検出するときに，再生不能の間に行われた個々の変更が混ざり合って検出されることがある．このことが定量的評価における解釈に影響を与えた可能性がある．なお，5.2 節で示した事例 1，2，3 の開発期間の再生についてはすべて正常に行うことができるので，この観点での評価に与える影響はない．

5.3.2 内的妥当性

本手法における集約規則やその適用の順番は，著者らの知見や経験に基づいて構築されている．これらの知見や経験に関する妥当性は未検証である．

また，本手法で検出可能なプログラム変更は連続的な変更の部分列のみである．つまり，不連続なプログラム変更の集まりを単一の変更タスクとして提示できない．評価実験で選択した開発期間において非連続な変更タスクは存在しなかったが，変更タスクが非連続にならないという状況にどの程度の一般性があるのかは不明である．

さらに，5.2 節で示した事例 1，2，3 において用意した変更タスクは，著者らの経験に基づき設定したものである．集約手法を提案した著者らにとって有利にならないように設定したつもりであるが，心理的バイアスが変更タスクの設定に影響を与えた可能性がある．

6. 関連研究

ここでは，関連研究を紹介することで，本研究の有用性を議論する．

6.1 版間差分を用いた手法

版管理システムが管理する行ベースの差分情報では，ある行に修正を加えた場合や行を移動した場合，それらの行の対応がとれずに個別の行の削除と追加と判断されてしまう．このような行ベースの差分抽出手法の問題を回避するために，新旧のソースコードから直接プログラム変更を検出手法がいくつも提案されている [13]．

Chawathe らは，新旧 2 つの版のソースコードから AST をそれぞれ作成し，それらの AST に含まれるノード間の対応をとることで差分情報を抽出する手法を提案している [14]．また，この手法を改善した手法が Fluri らにより提案されている [15]．これらの手法では，旧版の AST から新版の AST へ変換するための編集スクリプト（ノードの追加，削除，移動）が版間の差分情報として提示される．

ほかにも，Apiwattanapong らは，既存のコールグラフに動的束縛や例外処理のフローを取り入れた拡張コールグラフを定義し，新旧のソースコードの拡張コールグラフの差分から版間の振舞いの変化を検出している [16]．これにより，クラス・インタフェース，フィールド，メソッドなど，オブジェクト指向プログラムの要素に関する変更情報の検出を可能としている．

さらに，Maletic らは，ソースコード間の差分を抽出するために，XML を利用した手法を提案している [17]．この手法では，ソースコードを構造化文書と見なし，元のソースコードのテキスト情報を保持しながら階層情報を XML のメタ情報として埋め込む．新旧のソースコードから生成された XML 文書どうしを比較することで，階層情報を考慮した差分情報を抽出する．

これまでに紹介した手法はどれも，単純な行ベース差分の欠点を補うことに成功している．しかしながら，依然として，版管理システムの提供する新旧のソースコードのみから差分を抽出するという制約に縛られたままである．このため，2 つの版間に複数の異なる変更が混ざり合うことで発生する問題には対処できない．

このような問題は，Herzig らにより指摘されている [2]．Herzig らは，版管理システムに格納された単一のコミットメントに異なる変更が混ざり合うことで，版間差分情報を用いたソフトウェア進化分析の結果にどの程度の悪影響を及ぼすのかを調査している．さらに，版間の差分情報から抽出した変更操作間の距離を用いて，混ざり合った変更の分離を実現している．

ほかにも，Canfora らは，新旧 2 つの版から検出された行ベースの差分情報から，行の更新や移動などの情報を自動的に推測する手法を提案している [18]．この手法では，差分情報における削除コード片および追加コード片にそれぞれ含まれるトークンを特徴としたコサイン類似度を用いて，対応するコード片のペアを見つけ出す．

さらに，Kim らは，版の中に存在するすべてのメソッドヘッダ（パッケージ名，クラス名，名前，引数のリスト，戻り値の型の組）を新旧 2 つの版のソースコードからそれぞれ抽出し，それらの集合を比較することで旧版のメソッドヘッダの集合を新版のメソッドヘッダの集合に変換するための API 変更規則を推論する手法を提案している [19]．この手法ではさらに，オブジェクト指向プログラムの構造に基づく事実（たとえば，継承関係や参照関係）からプログラム要素に関する差分規則を推論する．これら 2 つの推論により，異なる複数の変更が混ざり合うことを回避している．この手法の特徴は，同様のプログラム変更が複数回検出された場合，それらの変更をそのまま提示するのではなく，抽象化された 1 つの規則として提示できることである．

これらの 3 つの手法は，新旧のソースコードに対する差分情報を解きほぐすことで，実際に行われたプログラム変更情報の検出を試みている．しかしながら，差分情報を新旧のソースコードのみから抽出している限り，適切なプログラム変更が検出できるかどうかは，新旧のソースコードの供給先の環境（版管理システムにおけるコミットの方針など）に大きく依存する．たとえば，変更タスクを意識したコミットが強要されているプロジェクトでは，より正

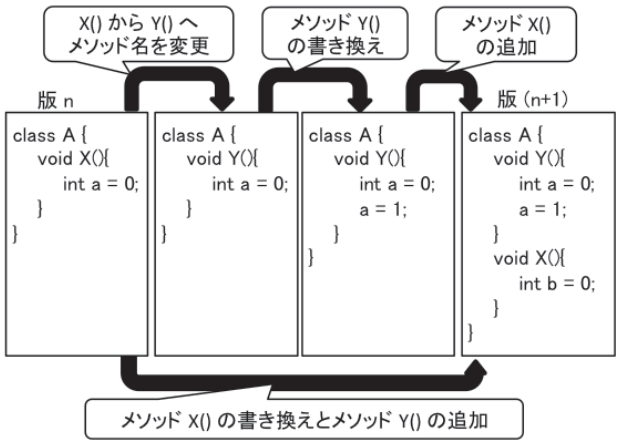


図 4 類似度に基づく差分ツールの限界
Fig. 4 Limitation of a similarity-based differencing tool.

確なプログラム変更が検出される可能性は高い．一方，コミットの方針が明確に規定されておらず，開発者が任意のタイミングでソースコードをコミットすることを許す環境では，差分情報に数多くのプログラム変更が混ざる恐れがある．このような状況において，正確なプログラム変更を検出することは依然として困難である．

ここでは，Kim らの手法で，正確なプログラム変更を検出することが困難な例を示す．**図 4** では，以下の 3 つの変更が版 n のソースコードと版 $(n+1)$ のソースコードの間でクラス `A` に対して行われたことを示している．

(1) メソッド `X()` を `Y()` に名前変更
(2) メソッド `Y()` のメソッド本体を書き換え
(3) メソッド `X()` を追加

このような変更が行われた版 n と版 $(n+1)$ のソースコードから正確なプログラム変更を検出するためには，版 n に存在するメソッド `X()` に対応するメソッド `Y()` を版 $(n+1)$ のソースコードにおいて特定することできるかどうかが鍵である．いま，Kim らの手法において，メソッドの類似度に基づきメソッドの対応を特定すると，メソッドヘッダに関するトークンベースの類似度により，版 $(n+1)$ の `X()` が選ばれる．メソッド本体のコードの類似度を用いても，`X()` が選ばれる可能性が高い．このように，メソッドヘッダや本体コードの類似度を用いても，実際に名前変更が実施された後のメソッド `Y()` の特定に失敗する可能性が残る．

これに対して，我々の変更検出手法では，できるだけ細かい差分情報からプログラム変更を検出する．このために，版管理システムに格納された新旧のソースコードではなく，過去の開発において行われたすべての編集操作履歴から復元した構文解析可能なソースコードのスナップショット群から差分情報を抽出する．一般的に，単一の版間差分には，構文解析可能なスナップショットが数多く含まれる．このため，編集操作履歴から復元したスナップショットベースの差分抽出の方が，新旧のソースコードを対象とした差分抽出よりも，混ざり合う変更の影響を受けにくいといえる．図 4 に示した例においても，版 n と版 $(n+1)$ 間に行われた 3 つの変更を，本手法で検出できることを確認した．

さらに，Negara らは，版管理システムに格納されずに上書きされる変更の存在を指摘している [20]．上書きされる変更を捕捉できる可能性が高い点でも，編集操作履歴を用いた差分抽出の方が有利である．ただし，より細かい差分情報を用いた方が検出の正確さという点で有利な反面，同様のプログラム変更が連続して検出されるという問題が発生する．そこで，提案手法では，プログラム変更の集約により，この問題を解決している．

以上をまとめると，版管理システムに格納された新旧のソースコードを利用するプログラム変更検出手法は，より大きく複雑な差分情報を解きほぐすことで，その目的を達成している．これに対して，我々の手法では，より細かい差分情報を寄せ集めることで，プログラム変更の検出を達成している点が大きく異なる．

6.2 より細かな差分を用いた手法

既存の版管理システムを改良することで，プログラム変更の検出を改善する試みも提案されている．たとえば，Magnusson らは，ソースコードをファイル単位で格納するのではなく，それを構成するプログラム要素ごとに格納する版管理システムを提案している [21]．このような版管理システムを内包する開発環境を構築し，それぞれのプログラム要素に対して開発時に行われた変更操作を記録する．版管理システムには，プログラム要素のスナップショットと，そのプログラム要素に対して行われた過去の変更操作の両方が同時にコミットされるため，プログラム要素の変更が容易に追跡できるようになっている．また，Hata らは，Git によるコード変更の特定と類似度に基づく名前変更の判別により，プログラム要素に対する細粒度な変更を検出し，それを管理するリポジトリを提案している [22]．また，山本らは，開発に関わるすべてのソースコードを自動的に採取し，そのコピーを残す堆積型ファイルシステム Moraine を提案している [23]．編集操作の粒度が同じ，あるいは，編集操作ごとにソースコードのスナップショットの差分が抽出されれば，我々の検出手法と同程度のプログラム変更が検出できる可能性がある．ただし，これらの手法では，我々の提案する検出手法のようなプログラム変更の集約の仕組みを備えていない．よって，開発者や保守者に提示されるプログラム変更の抽象レベルを柔軟に切り替えることはできない．

差分ではなく開発時の編集操作を利用することで，単一の版間差分に目的の異なる変更が混ざるのを防ぐ方法として，林らの研究がある [24], [25], [26]．この手法では，我々の手法と同様に，OperationRecorder により記録された編集

操作履歴を利用する．特に，編集操作に注釈を付与することで過去の変更過程を明示したり，編集操作履歴の加工を支援したりすることで，分割コミットを実現している．さらに，あらかじめ設定した時間基準，プログラム要素基準，コメント基準により，編集操作の自動グルーピングを実現している．グルーピングの方針が我々の手法と類似しているものの，林らの方法では編集操作を直接グルーピングしている．これに対して，我々の手法は，分割コミットの実現を目的としているわけではなく，プログラム変更の検出である．このため，編集操作履歴に基づき抽出したプログラム変更をグルーピングの対象としている点が異なる．

7. おわりに

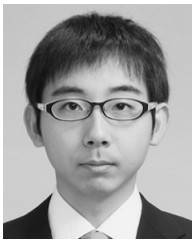
本論文では，開発者や保守者によるプログラム理解を支援するため，開発時にソースコードに対して行われた編集操作の履歴から，オブジェクト指向におけるプログラム要素に関する変更情報を自動的に検出する手法を提案した．また，本手法を実現したツールを Eclipse プラグインとして実装し，それを用いた評価実験の結果より，本手法の有用性を示した．

今後の課題として，集約方針の改良とさらなる評価実験の実施を検討している．たとえば，時間的距離と空間的距離の組合せや，プログラム要素ごとに基準値を複数設定できる環境を考えている．また，今回の実験では，検出されたプログラム変更の数と人手で作成した変更タスクとの比較により，提案手法の有用性を評価した．プログラム変更の理解という観点から，さらに別の評価基準を用意した上で有用性の評価を継続していく予定である．

謝辞 本研究の一部は，科研費（24500050，26730042）の助成を受けたものである．

参考文献

[1] Robbes, R. and Lanza, M.: A Change-based Approach to Software Evolution, *ENTCS*, Vol.166, pp.93–109 (2007).
[2] Herzig, K. and Zeller, A.: The impact of tangled code changes, *Proc. MSR '13*, pp.121–130 (2013).
[3] Murphy-Hill, E., Parnin, C. and Black, A.P.: How we refactor, and how we know it, *Proc. ICSE '09*, pp.287–297 (2009).
[4] Robbes, R. and Lanza, M.: SpyWare: A change-aware development toolset, *Proc. ICSE '08*, pp.847–850 (2008).
[5] Hattori, L., D'Ambros, M., Lanza, M. and Lungu, M.: Software Evolution Comprehension: Replay to the Rescue, *Proc. ICPC '11*, pp.161–170 (2011).
[6] Omori, T. and Maruyama, K.: A change-aware development environment by recording editing operations of source code, *Proc. MSR '08*, pp.31–34 (2008).
[7] 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).
[8] Ryder, B.G. and Tip, F.: Change impact analysis for object-oriented programs, *Proc. PASTE '01*, pp.46–53 (2001).
[9] Ren, X., Shah, F., Tip, F., Ryder, B.G. and Chesley, O.: Chianti: A tool for change impact analysis of java programs, *ACM SIGPLAN Not.*, Vol.39, pp.432–448 (2004).
[10] Kitsu, E., Omori, T. and Maruyama, K.: Detecting Program Changes from Edit History of Source Code, *Proc. APSEC '13*, pp.299–306 (2013).
[11] 木津栄二郎，大森隆行，丸山勝久：ソースコード編集履歴を用いたプログラム変更の検出，コンピュータソフトウェア，Vol.29, No.2, pp.168–173 (2012).
[12] Omori, T. and Maruyama, K.: An editing-operation replayer with highlights supporting investigation of program modifications, *Proc. IWPSE-EVOL '11*, pp.101–105 (2011).
[13] Kim, M. and Notkin, D.: Program Element Matching for Multi-version Program Analyses, *Proc. MSR '06*, pp.58–64 (2006).
[14] Chawathe, S.S., Rajaraman, A., Garcia-Molina, H. and Widom, J.: Change Detection in Hierarchically Structured Information, *Proc. SIGMOD '96*, pp.493–504 (1996).
[15] Fluri, B., Wüersch, M., Pinzger, M. and Gall, H.C.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction, *Transactions on Software Engineering*, Vol.33, No.11, pp.725–743 (2007).
[16] Apiwattanapong, T., Orso, A. and Harrold, M.: JDiff: A differencing technique and tool for object-oriented programs, *Proc. ASE '07*, Vol.14, pp.3–36 (2007).
[17] Maletic, J. and Collard, M.: Supporting source code difference analysis, *Proc. ICSM '04*, pp.210–219 (2004).
[18] Canfora, G., Cerulo, L. and Di Penta, M.: Identifying Changed Source Code Lines from Version Repositories, *Proc. MSR '07*, pp.14–21 (2007).
[19] Kim, M., Notkin, D., Grossman, D. and Wilson, G.: Identifying and Summarizing Systematic Code Changes via Rule Inference, *Transactions on Software Engineering*, Vol.39, No.1, pp.45–62 (2013).
[20] Negara, S., Vakilian, M., Chen, N., Johnson, R. and Dig, D.: Is It Dangerous to Use Version Control Histories to Study Source Code Evolution?, *Proc. ECOOP '12*, pp.79–103 (2012).
[21] Magnusson, B., Asklund, U. and Minör, S.: Fine-grained Revision Control for Collaborative Software Development, *SIGSOFT '93*, pp.33–41 (1993).
[22] Hata, H., Mizuno, O. and Kikuno, T.: Historage: Fine-grained Version Control System for Java, *Proc. IWPSE-EVOL '11*, pp.96–100 (2011).
[23] 山本哲男，松下 誠，井上克郎：堆積型ファイルシステム Moraine とメトリクス環境 MAME への適用，コンピュータソフトウェア，Vol.18, No.3, pp.250–260 (2001).
[24] Hayashi, S. and Saeki, M.: Recording Finer-grained Software Evolution with IDE: An Annotation-based Approach, *Proc. IWPSE-EVOL '10*, pp.8–12 (2010).
[25] Hayashi, S., Omori, T., Zenmyo, T., Maruyama, K. and Saeki, M.: Refactoring edit history of source code, *Proc. ICSM '12*, pp.617–620 (2012).
[26] 星野大樹，林 晋平，佐伯元司：ソースコード編集操作の自動グループ化，ソフトウェア工学の基礎 XX 日本ソフトウェア科学会 FOSE2013，pp.107–112 (2013).



木津 栄二郎

2010 年立命館大学情報理工学部情報システム学科卒業．2012 年同大学大学院理工学研究科情報理工学専攻計算機科学科博士課程前期課程修了．2012 年より同大学院情報理工学研究科情報理工学専攻博士課程後期課程に在学中，現在に至る．日本ソフトウェア科学会学生会員．

大森 隆行 （正会員）

2003 年立命館大学理工学部情報学科卒業．2005 年同大学大学院理工学研究科博士課程前期課程修了．2008 年同研究科博士課程後期課程修了．同年立命館大学情報理工学部助手．2010 年同大学情報理工学部助教，現在に至る．2013 年 9 月～2014 年 3 月 University of British Columbia 客員助教．ソフトウェア開発環境，ソフトウェア進化等の研究に従事．博士（工学）．電子情報通信学会，日本ソフトウェア科学会各会員．

丸山 勝久 （正会員）

1991 年早稲田大学理工学部電気工学科卒業．1993 年同大学大学院理工学研究科修士課程修了．同年日本電信電話株式会社入社．2000 年 4 月より立命館大学理工学部助教授．2007 年 4 月より同大学情報理工学部教授．2003 年 9 月～2004 年 9 月 University of California，Irvine 客員研究員．ソフトウェア保守，ソフトウェア再利用，ソフトウェア開発環境，プログラム理解支援の研究に従事．博士（情報科学）．日本ソフトウェア科学会，電子情報通信学会，IEEE Computer Society，ACM 各会員．