開発履歴情報を用いた修正漏れの検出

肥後 芳樹[1,a)]　楠本 真二[1,b)]

概要：バグ修正や機能追加を行うためにソースコードに対して修正を行う場合は，修正が必要なすべてのコード片を把握しなければならない．修正が必要なコード片を見落としてしまうと，ソースコード中に修正漏れが発生してしまう．ソースコードの修正前であれば，grep 等のキーワード検索ツールを用いることにより，修正が必要なコード片を見落とす可能性は低くなる．しかし，発生した修正漏れに対しては，そのようなツールでは十分な支援を得られない．本論文では，ソースコード中において修正漏れを含むコード片の検出手法を提案する．提案手法は，ソースコードの開発履歴をマイニングし，どのようなコード片がどのように変更されたかを学習する．そして，その学習データに基づいて最新バージョンのソースコードから，修正漏れを含むコード片を検出する．提案手法を実装し，HTTPD と FreeBSD に対して実験を行った．その結果，バグ修正漏れ，リファクタリング実施漏れ，機能変更・機能追加漏れ，コメント修正漏れを含むコード片を多数検出できた．

Identifying Unintended Inconsistencies Based on Mining Development Histories

HIGO YOSHIKI[1,a)]　KUSUMOTO SHINJI[1,b)]

Abstract: When we modify source code for performing a given bug fix or functional addition, we must recognize all the code fragments to be modified. If not, unintended inconsistences occur in the source code. Before modifying the source code, keyword-based search tools like grep are suitable for preventing the code fragments from being overlooked. However, once inconsistencies occur in the source code, such tools cannot help us adequately. In this paper, we propose a new method to identify unintended inconsistencies in source code automatically. The proposed method analyzes source code modifications in a repository to derive modification patterns. A modification pattern indicates what kind of code and how it was modified. The derived modification patterns are queries to identify unintended inconsistencies from the latest version of source files. We implemented the proposed method and applied it to HTTPD and FreeBSD. As a result, we identified many overlooked code fragments for bug fixes, refactorings, functional enhancements, and code comments.

1. はじめに

バグ修正や機能追加を行う場合には，ソースコード中の修正が必要な全てのコード片を把握する必要がある．もしそれらを見落とすと修正漏れが発生してしまう．それらは後にフォールトの原因となり，システムの可用性を下げるばかりではなく，そのコード片を特定し修正するためのコストも必要となる．

ソースコードの修正前であれば，grep 等のキーワード検索ツールやコードクローン検出ツールを用いることにより，修正が必要なコード片を見落とす可能性は低くなる．泉田らは，オープンソースソフトウエアのバッファオーバーフローの修正を題材に，修正が必要なコード片の見逃し防止策として，grep とコードクローン検出ツールが有効であることを示した [1]．また，森崎らは，国内の企業が開発したシステムに対してコードクローン検出ツールを適用し，それがコード片の見逃し防止対策として有効であることを示した [4]．しかし，それらのツールは，発生した修正漏れついては十分に支援できない．

- grep を利用するには検索対象キーワードが必要である．しかし，どのコード片において修正漏れが存在するのか不明なため，キーワードの指定ができない．
- コードクローン検出ツールを用いれば，コピーアンドペースト後の変数名の置換漏れ等の軽微な修正漏れを検出できる [3]．しかし，文の追加・削除等の大きな単

[1] 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
[a)] higo@ist.osaka-u.ac.jp
[b)] kusumoto@ist.osaka-u.ac.jp

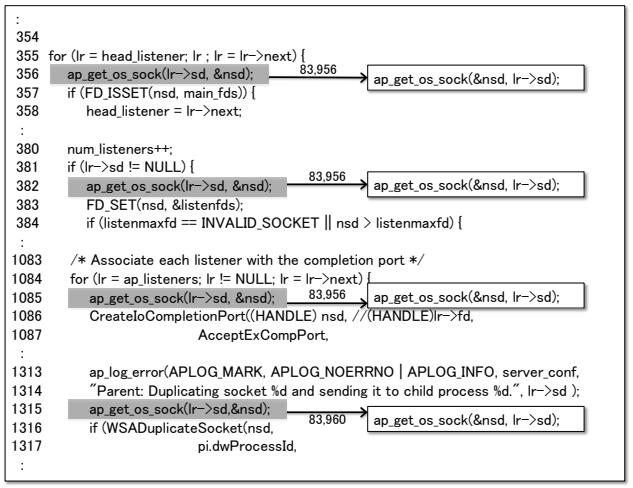


図 1 server/mpm/winnt/mpm_winnt.c の一部（リビジョン 83,955）
Fig. 1 a part of server/mpm/winnt/mpm_winnt.c (revision 83,955)

位の修正漏れは検出が難しい．また，コードクローンとして検出されるためには，コード片が一定以上の大きさで類似している必要がある．

本論文では，開発履歴をマイニングすることにより，修正漏れを含むコード片を検出する手法を提案する．具体的には，開発履歴のマイニングにより，どのようなコード片がどのように変更されているのかを学習する．その学習データを用いて，対象ソースコードから過去に変更されているコード片と同形のコード片を検出する．提案手法は，grep やコードクローン検出ツールの問題点を解決しており，以下の特徴を持つ．

- 字句単位の修正漏れ（変数やリテラルの置換漏れ）だけではなく，文単位の修正漏れ（文の追加・削除漏れ）も検出可能
- 修正漏れを含むコード片がある程度以上の大きさの重複コードである必要がない
- ソースコードの深い解析を必要とせず，複数のプログラミング言語への展開が容易
- スケーラビリィが高く，数百万行規模のソフトウェアであっても数十分程度で解析可能

提案手法をツールとして実装し，HTTPD と FreeBSD に適用し，以下の結果を得た．

- 多くのリビジョンにおいて修正漏れが存在していること，および，それらの多くは後に修正が加えられていること（修正漏れではなくなること）がわかった．
- 1 つのバージョンに着目した場合に，それまでの開発履歴から得た学習データを用いて多数の修正漏れを検出できた．

2. 研究の動機

図 1 は，HTTPD のソースコードである．このソースコードでは，356 行目，382 行目，1,085 行目，1,315 行目に同様の命令がある．リビジョン 83,956 において，そのうちの 3 つ（356 行目，382 行目，1085 行目）が変更されている．その後，リビジョン 83,960 において，1,315 行目に対しても同様の修正が行われている．リビジョン 83,960 のコミットログには，修正漏れがあったことを表す文章が記述されている．このことから，これら 4 つのコード片は同時に同様の修正がされるべきであったが，開発者がこのうちの 1 つを見落としていたことがわかる．この例で示したように，複数のコード片に対して同様の修正を行う必要がある場合に，開発者が修正すべき全てのコード片を把握しているとは限らない．

そこで，本論文では，下記のリサーチクエスチョンを設定する．

RQ1 同時に行われていない同様の修正がどの程度存在するのか

修正すべきコード片を特定した場合に，grep 等のキーワード検索ツールを用いれば，他の修正すべき箇所を列挙することができるため，修正漏れが発生する可能性が低くなる．しかし，発生した修正漏れについては，grep で支援を得ることは難しい．

コードクローン検出手法を用いて修正漏れを検出する方法が提案されている [3]．この手法ではコードクローン検出し，そこに含まれている変数の対応関係を調べる．対応関係がとれていないコードクローンが修正漏れの候補としてユーザに提示される．この手法はコードクローンに含まれる修正漏れを提示するが，コードクローンとして検出されないコード片については修正漏れを指摘できない．図 1 では，修正を行うべき 4 つの行はコードクローンであるが，その前後の行は重複していない．つまり，一行のみの小さなコードクローンが 4 つ存在している．コードクローン検出ツールの設定を変更することで，このような小さいコードクローンを検出することは可能である．しかし，そのような小さいコードクローンの検出を行った場合，多量のコードクローンが検出されてしまうため，検出されたコードクローンを調査する負荷が大きい．さらには，修正内容が図 1 に示すような変数名の変更ではなく，文の追加や削除といったより大きな変更だった場合は，コードクローンとして検出されない．

本論文では，開発履歴をマイニングすることにより修正漏れを含むコード片とその修正方法を提示する手法を提案する．たとえば，図 1 の場合では，リビジョン 83,955 から 83,956 への変更をマイニングすることにより，下記の変更が 3 回起こったことがわかる．

```
ap_get_os_sock(lr->sd, &nsd);
        ↓
ap_get_os_sock(&nsd, lr->sd);
```

この情報を用いることにより，リビジョン 83,956 以降で

は，ソースコード中に“ap_get_os_sock(lr->sd, &nsd);”が存在した場合には，“ap_get_os_sock(&nsd, lr->sd);”に変更すべきである，という提示ができる．

また，提案手法の有効性を調査するために，下記のリサーチクエスチョンを設定する．

RQ2 開発履歴をマイニングすることにより，どの程度の修正漏れを検出できるか

3. 提案手法

本論文では，ソースコード中に存在する修正漏れを自動的に検出する手法を提案する．提案手法は，開発履歴を解析する**マイニング処理**と対象ソースコードから修正漏れを検出する**検出処理**から構成されている．まず，3.1 節で提案手法で利用する用語の定義を行う．次に，3.2 節でマイニング処理，その後，3.3 節で検出処理について述べる．なお，提案手法は，CVS や Subversion 等のバージョン管理システムで管理されているソフトウェアを前提としている．

3.1 用語の定義

まず，**コード片**を定義する．

定義 1 (コード片) ソースファイル中の一部分を表す，長さが 0 以上の文字列である．

修正パターンは，どのようなコード片が，いつ，どのようなコード片に修正されたのかを表す．以下に定義を示す．

定義 2 (修正パターン) 修正前のコード片を CF_{before}，修正後のコード片を CF_{after}，修正が行われたリビジョンを r とした場合，その修正パターンを，$(CF_{before}, CF_{after}, r)$ で表す．

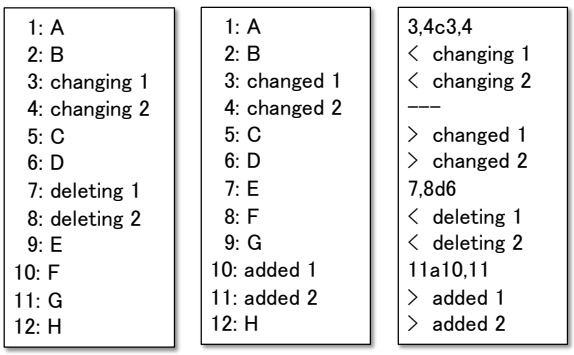
次に，修正パターンの**支持度**[*1]を定義する．

定義 3 (支持度) 修正パターンの出現回数を表す値である．2 つの修正パターン $MP_1 = (CF1_{before}, CF1_{after}, r_1)$，$MP_2 = (CF2_{before}, CF2_{after}, r_2)$, があった場合に, $CF1_{before} = CF2_{before} \wedge CF1_{after} = CF2_{after}$ が真になれば，MP_1 と MP_2 は同じ修正パターンと定義される．よって，MP_1 や MP_2 が表す修正パターンの出現回数は 2 回となる．もし，MP_1 と MP_2 が同じでない場合は，それぞれの修正パターンは 1 回ずつ出現したことになる．

次に，修正パターンの**確信度**[*2]を定義する．

定義 4 (確信度) ある修正前コード片から，ある修正後コード片が生成される確率である．修正パターンの集合が与えられた時に，それらの中で修正前コード片が CF_{before} である修正パターンの数を n とする．また，それらのうち，修正後コード片が CF_{after} である修正パターンの数を m とする．このとき，修正前コード片が CF_{before}，修正後コード片が CF_{after} である修正パターンの確信度は，$\frac{m}{n}$ で表される．なお，一種類の修正前コード片は複数種類の修正後コード片に変化しうるため，$m \leq n$ となる．

[*1] support
[*2] confidence



(a) 修正前　(b) 修正後　(c) 差分

図 2 修正前ソースファイル，修正後ソースファイル，diff コマンドを用いたそれらの間の差分出力

Fig. 2 Source files before and after a modification, and diff output between them

3.2 マイニング処理

マイニング処理の入力と出力を以下に示す．

入力 対象ソフトウェアのリポジトリ

出力 支持度と確信度の付いた修正パターン群

マイニング処理は下記の手順からなる．

手順 1 ソースファイルが修正されたリビジョン群の特定

手順 2 特定したリビジョン群から修正パターンを抽出

手順 3 抽出した修正パターンの支持度と確信度を計算

手順 1 では，少なくとも 1 つのソースファイルが修正されたリビジョンを特定する．リポジトリでは，ソースコードに加えてドキュメント等の成果物も管理されている．そのため，すべてのリビジョンでソースコードが修正されているわけではない．そのようなリビジョンは修正パターンの抽出対象とはならない．バージョン管理システムは，指定したリビジョンで修正されているファイル一覧を表示する機能を持つ．この機能により表示されたファイルの拡張子を調査することにより，マイニング処理が必要であるリビジョン群の特定を行える．

手順 1 において，n 個のリビジョン $\{r_1, r_2, \cdots, r_n\}$ でソースファイルが修正されていた場合，手順 2 では，r_1 と r_2，r_2 と r_3，$\cdots$, r_{n-1} と r_n の間の差分を取得する．取得した差分に現れているコードが，修正パターンの修正前コード片と修正後コード片である．図 2 は，修正前後のソースコードとその差分を diff コマンドを用いて出力した様子を表している．この例の場合だと図 2(c) の 3,4c3,4, 7,8d6，11a10,11 の 3 つが修正パターンとして抽出される．

全ての修正パターンを抽出した後に，手順 3 では，修正パターンの支持度と確信度を計算する．

3.3 検出処理

まず，検出処理の入力と出力を以下に示す．

入力
- 対象リビジョンのソースファイル群
- 支持度と確信度の付いた修正パターン群（マイニング処理の出力）
- 支持度，確信度，一致箇所数のしきい値

出力 対象リビジョンのソースファイル中における修正漏れ候補とその修正方法

一致箇所数とは，1 つの修正パターンによって検出されたコード片（修正漏れ候補）の数である．例えば，多くのコード片が 1 つの修正パターンによって検出された場合は，それらは修正漏れとは考えにくい．過去に修正されたコード片と同形のコード片が，修正前の状態で対象ソースファイルに存在していることは事実であるが，それらは意図的にその状態であると考えられる．このような誤検出（意図的に修正を行っていないコード片の検出）を抑えるために，一致箇所数にしきい値を設ける．たとえ，修正漏れ候補のコード片が見つかった場合でも，そのしきい値以上の数であれば，それらは検出処理において出力されない．

検出処理は以下の手順からなる．

手順 1 修正漏れの検出に用いる修正パターンの抽出

手順 2 抽出した修正パターンを用いて，対象リビジョンから修正漏れ候補を検出

手順 1 では，入力として与えられた修正パターン群のうち，支持度と確信度が共にしきい値以上の修正パターンが抽出される．

手順 2 では,，抽出した修正パターンの修正前コード片を用いて対象リビジョンのソースファイルを検索する．一致したコード片が修正漏れ候補である．もし，一致したコード片の数が一致箇所数のしきい値以下であれば，それらは出力される．なお，その修正パターンの修正後コード片が修正方法となる．

4. 実装

提案手法をツールとして実装した．現在のところ，対応しているバージョン管理システムは Subversion のみである．作成したツールの入力を以下に示す．

- 対象システムのリポジトリ
- 修正漏れコード片の検索対象ソースファイル群
- 支持度，確信度，一致箇所数のしきい値

出力は検索対象ソースファイル群から検出された修正漏れ候補である．

ツールから Subversion リポジトリの操作には, SVNKit[*3] を用いている．マイニング処理では，更新されたファイルの特定を行うために svn log コマンドを，リビジョン間の差分取得のために svn diff コマンドを用いている．しかし，diff コマンドにより得られたコード片をそのまま用いると修正パターンの誤検出[*4]が多数検出されてしまうため，コード片の正規化を行う．正規化後の修正前コード片と修正後コード片を比較し，それらが同じ文字列である場合は，修正パターンとしての抽出は行わない．具体的には下記の手順で処理を行う．

手順 1 svn diff コマンドを用いて差分を取得

手順 2 差分に現れている修正前後のコード片の正規化（空白，タブ，改行文字の削除）

手順 3 正規化後の修正前後のコード片が同一文字列でなければ，修正パターンとして抽出

この実装では，コード片の同値性は以下のように定義されている．

定義 5 (コード片の同値性) 2 つのコード片から全ての空白，タブ，改行文字を取り除いた後に，それらが同一文字列になるのであれば，その 2 つのコード片は同値である．

5. 評価

オープンソースソフトウェアの HTTPD[*5] と FreeBSD[*6]に対して実験を行った．実験対象の詳細については，表 1 に示す．HTTPD の実験対象期間は，1.3.x ブランチが作成される前とした．この期間では，バージョン 1.3.0 のリリースに向けて新機能の追加とバグ修正が行われていた．FreeBSD の実験対象期間は，STABLE8 ブランチが作成されてから STABLE9 ブランチが作成されるまでの期間とした．この期間では，バージョン 9 に搭載する新機能の追加を行い，期間終了間近では，新機能の追加は凍結され，安定化のためのバグ修正のみを行うという開発プロセスが取られていた．これらの期間に存在しており，かつ，1 つ以上の.c ファイルが更新されている 2,784 個のリビジョン（HTTPD）と 7,689 個のリビジョン（FreeBSD）を対象とした．また，この実験では，しきい値を用いて下記の値を用いた．

支持度の下限 3

確信度の下限 1.0

一致箇所数の上限 1

マイニング処理の結果として，HTTPD から 737 個，FreeBSD から 983 個の修正パターンを得た．以降，本節では，これらの修正パターンを用いて，2 節で示した 2 つの RQ について調査した結果を述べる．

5.1 RQ1 に対する実験

HTTPD については 145/737(=19.6%)，FreeBSD については 94/983(=9.6%) の修正パターンが複数のリビジョンに出現していた．これら 145 個と 94 個の修正パターンについて，手作業でその修正内容を確認し[*7]，4 つのカテゴリに

[*3] http://svnkit.com/

[*4] ここでの誤検出とは，修正漏れの指摘には役にたたない修正パターンの意味である．

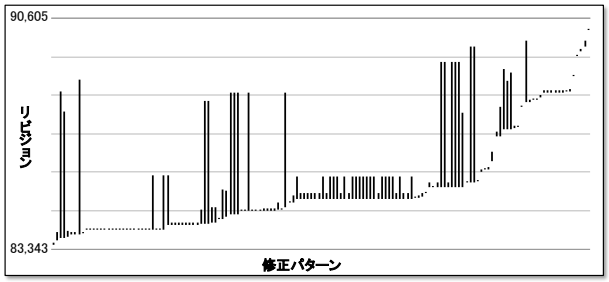
[*5] http://svn.apache.org/repos/asf/httpd/httpd/trunk/

[*6] http://svn.freebsd.org/base/head/sys/

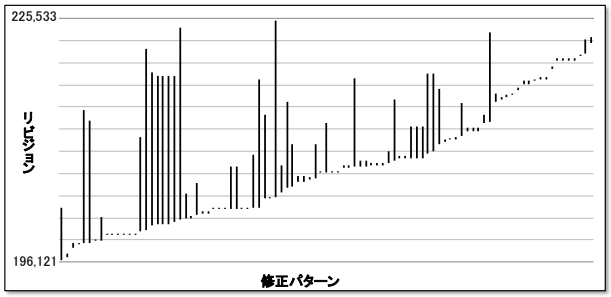
[*7] 著者らがリポジトリのログ，修正パターン周辺のコード，その後

表 1 対象ソフトウェアの概要
Table 1 Outline of target software

ソフトウェア	開始リビジョン (日付)	終了リビジョン (日付)	対象リビジョン数	終了リビジョンの総行数
HTTPD	81,442 (1998-06-02)	90,607 (2001-08-24)	2,784	131,675
FreeBSD	196,121 (2009-08-12)	225,533 (2011-09-14)	7,689	3,570,021



(a) HTTPD



(b) FreeBSD

図 3 RQ1 で抽出した修正パターンの出現期間
Fig. 3 Occurrence periods of extracted modification patterns in RQ1 investigation

分類した（表 2）．各カテゴリについて説明する．

バグ修正 バグを取り除くための修正

リファクタリング リファクタリングのための修正．変数名の変更等の簡単な修正も含む

機能変更・機能追加 すでに実装された機能の変更や，新機能の追加のための修正

コメント ソースコード中のコメントに対する修正．主として著作権関係の記述のため

42 個の修正パターンがバグ修正に分類された．これらは，開発者がバグを取り除くために修正しなければならないコード片を見落としていたことを表している．117 個の修正パターンが機能変更・機能追加に分類された．これらも必要な機能が正しく実装されていないリビジョンが存在

表 2 RQ1 の調査で抽出した修正パターンの内訳
Table 2 Extracted modification patterns in RQ1 investigation

ソフトウェア	合計	バグ修正	リファクタリング	機能変更・機能追加	コメント
HTTPD	145	19	22	93	11
FreeBSD	94	23	39	24	8

の修正の様子を調査した．

```
write(c->fd,request,reqlen);
if (posting) {                    deleted line
if (posting>0) {                  added line
    write(c->fd,postdata,postlen);
    totalposted += (reqlen + postlen);
}
```

(a) HTTPD

```
if (ste_newbuf(sc, cur_rx) != 0) {
    ifp->if_ierrors++;            deleted line
    ifp->if_iqdrops++;            added line
    cur_rx->ste_ptr->ste_status = 0;
    continue;
}
```

(b) FreeBSD

図 4 最も長期間にわたって現れた修正パターンのソースコード
Fig. 4 Source code of modification patterns occuring longest period

していたことを表している．61 個の修正パターンがリファクタリングに分類された．大部分は関数名やマクロ名の変更であった．これらはソフトウェアの振る舞いに直接影響を与えるわけではないが，ソースコード中に一貫性のないコード片が存在しているという点では望ましくない．19 個の修正パターンがコメントに分類された．全ての修正パターンは，著作権記述部分における西暦の修正漏れであった．著作権情報を正しく記述するという点では，これらも望ましくはない．

図 3 は，複数リビジョンに出現していた 145 個と 94 個の修正パターンの出現期間を表している．横軸には修正パターンが並べられており，縦軸はリビジョンを表している．Y 軸方向の線分の両端は，その修正パターンが現れる最初

表 3 最も長期間にわたって現れた修正パターン
(a) HTTPD

リビジョン	日付	修正されたファイル
83,946	1999-10-08	support/ab.c (2 places)
88,628	2001-04-02	support/ab.c

(b) FreeBSD

リビジョン	日付	修正されたファイル
200,965	2009-12-25	sys/dev/ste/if_ste.c
213,438	2010-10-05	sys/dev/usb/net/usb_ethernet.c
218,832	2011-02-19	sys/dev/dc/if_dc.c
223,648	2011-06-29	sys/dev/gem/if_gem.c
223,951	2011-07-12	sys/dev/cas/if_cas.c (2 places)

のリビジョンと最後のリビジョンを表している．たとえば，FreeBSD（図 3(b)）の一番左に配置されている修正パターンは，リビジョン 196,386 で最初に現れ，リビジョン 203,049 で最後に現れている．修正パターンは最初に現れたリビジョンが早い順に並べられている．この図より，大部分の対象リビジョンにおいて，少なくとも 1 つの修正漏れが含まれていることがわかる．具体的には，HTTPD の 92.7%，FreeBSD の 98.4%のリビジョンが該当する．

図 3 より，長期間にわたって出現するパターンも存在していることがわかる．最も長期間にわたって出現していた修正パターンのコードを図 4 に，この修正パターンが出現したリビジョンの情報を表 3 に示す．HTTPD の修正パターン（図 4(a)）では，if 文の条件式が変更されている．FreeBSD の修正パターン（図 4(b)）では，インクリメントする変数が変更されている．これら 2 つの修正パターンはバグ修正に分類された．いずれも軽微な変更ではあるが，最初の修正と最後の修正が一年半以上も離れている．

以上のことから，RQ1 に対して以下のように回答できる：修正パターンのうちの 19.7%と 9.6%が複数のリビジョンに現れていた．また，これらの修正パターンは，対象リビジョン群の 92.7%と 98.4%に存在していた．

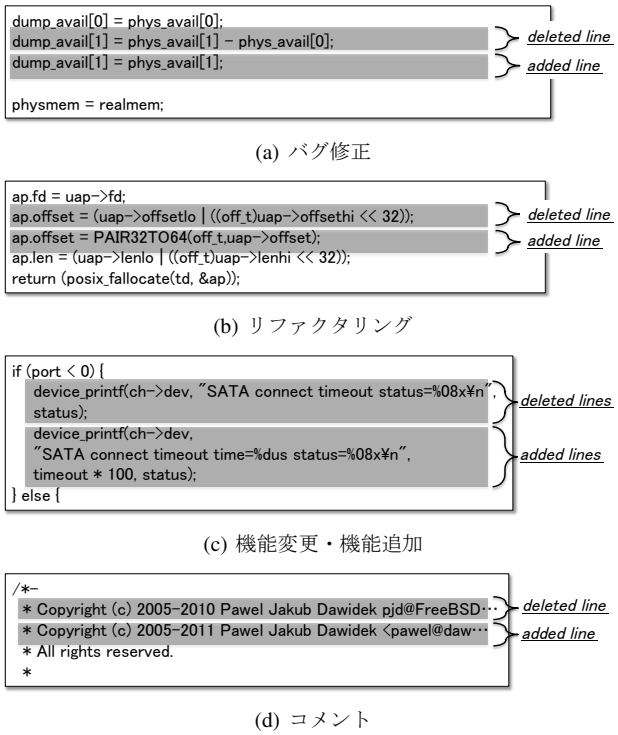


図 5 リビジョン 225,533 において見つかった修正漏れの例

Fig. 5 Examples of detected overlooked code fragments in revision 225,533

5.2 RQ2 に対する実験

マイニング処理で抽出した 737 個と 983 個の修正パターンを用いて，HTTPD のリビジョン 90,607 と FreeBSD のリビジョン 225,533 から修正漏れコード片を検出した（提案手法の検出処理を行った）．その結果，HTTPD から 18 個，FreeBSD から 94 個のコード片が検出された．これらすべてのコード片を手作業により調査し，一致した修正パターンの修正後コード片のように変更すべきかどうかを確認した[*8]．

表 4 は，調査の結果を表している．15 個のバグ修正漏れが検出された．図 5(a) はその一例である．この修正パターンでは，配列の要素 dump_avail[1] の計算方法が変更されている．修正パターンが現れたリビジョンのコミットログには，“dump_avail layout should be sequence of [start, end) pairs, not <start, size>.” と記載されており，この修正パターンがバグ修正であることを表している．FreeBSD のリビジョン 225,533 のファイル ar71xx_machdep.c には “*deleted line*” に示す行が存在しており，この行は修正パターンの修正前コード片と一致するため，バグの修正漏れであると判断した．

55 個のリファクタリング漏れのが検出された．図 5(b) はその一例である．この修正パターンでは，シフト演算および論理演算を行う式がマクロに変更されている．このマクロは，この修正パターンを抽出したリビジョン 205,014 で追加されていた．また，このマクロはこのリファクタリング漏れが検出されたファイルに定義されていた．よって，簡単に置換によるリファクタリングを行える．このコード片は，リビジョン 205,014 において実施したリファクタリング実施時に見落とされたと考えられる．

12 個の機能変更・機能追加漏れが検出された．図 5(c) はその一例である．この修正パターンでは，タイムアウトになった処理時間を表示する機能が追加されている．修正パターンの修正後コード片で出現している変数 timeout は，検出されたコード片では利用可能であるため，簡単に機能追加を行える．

3 つのコメント修正漏れが検出された．図 5(d) にその一例である．この修正パターンでは，西暦と開発者のメールアドレスが古い情報のままであった．

また，27 個のコード片を誤って検出していた．これらは，手作業による調査により意図的に修正を行なっていないと判断された．

以上のことから，RQ2 に対して以下のように回答できる：HTTPD と FreeBSD から，16 個と 69 個の修正漏れを検出できた．HTTPD については 8,229 行あたりに 1 つ，FreeBSD については 51,739 行あたり 1 つの修正漏れが存在していた計算になる．また，検出の精度は 88.8%と 73.4%であった．

5.3 コードクローン検出ツールとの比較

2 節で述べたように，提案手法の目的はコードクローン

[*8] 著者らがリポジトリのログ，修正パターン周辺のコード，その後の修正の様子から判断した．

表 4　RQ2 の調査で検出した修正漏れ候補の内訳
Table 4　Detected overlooked code fragments in RQ2 investigation

ソフトウェア	合計	修正漏れと判断				誤検出数	精度
		バグ修正	リファクタリング	機能変更・機能追加	コメント		
HTTPD	18	2	4	10	0	2	88.9%
FreeBSD	94	13	51	2	3	25	73.4%

検出手法では検出できない修正漏れを検出することである．その目的が達成されているかを調査するために，コードクローン検出ツールが RQ2 で検出した修正漏れコード片を検出できるのかを調査した．この調査では，CCFinder と Nicad を利用した．これらのツールの特徴を以下に述べる．

CCFinder [2]　字句単位でコードクローンを検出するツール．変数名や関数名等のユーザ定義名を置換した上でコードクローンを検出するため，それらが異なっているコード片でもコードクローンとして検出できる．つまり，字句レベルでの修正漏れを検出できる．

Nicad [5]　関数単位やブロック単位[*9]でコードクローンを検出するツール．Nicad は，LCS（Longest Common Sequence）アルゴリズムを用いて連続して字句が一致している部分を検出する．その一致部分が関数やブロック全体の一定以上の割合であれば，その関数やブロックがコードクローンとして検出される．つまり，Nicad は字句単位だけではなく，より大きな単位の不一致部分が含まれていてもコードクローンとして検出できる．

この実験では，デフォルトの設定を用いてコードクローンを検出した．表 5 は，ツールによる修正漏れの検出結果を表している．CCFinder と Nicad 共に，提案手法が検出した修正漏れの大部分を検出できなかった．この理由は，修正漏れの前後に十分な重複コードが存在しておらず，コードクローンとして検出できなかったためである．

FreeBSD の場合は，CCFinder は 38 個のリファクタリング漏れを検出できた．これらは関数名が変更されたリファクタリングであり，字句単位での修正漏れであるため，CCFinder では検出できた．一方，この部分の重複コードはブロック内における割合が低かったために，Nicad では検出できなかった．

表 6 はツールによって検出されたクローンセット[*10]とコードクローンの数を表している．2 つのツール共に多くのコードクローンを検出した．しかし，表 5 に示すように，修正漏れの多くを検出できなかった．この結果は，コードクローンの検出ツールを用いて効率的に修正漏れを検出するのは難しいことを示唆している．

表 5　CCFinder と Nicad により検出された修正漏れの数
Table 5　Number of overlooked code fragments detected by CCFinder and Nicad

(a) HTTPD

手法	合計	バグ修正	リファクタリング	機能変更・機能追加	コメント
提案手法	16	2	4	10	0
CCFinder	2	0	0	2	0
Nicad	2	0	0	2	0

(b) FreeBSD

手法	合計	バグ修正	リファクタリング	機能変更・機能追加	コメント
提案手法	69	13	51	2	3
CCFinder	39	1	38	0	0
Nicad	2	1	1	0	0

[*9] ここでブロックとは，if 文や while 文のように，関数内での構造的なまとまりのある部分を指す．

6. 議論

6.1 正規化について

現在の実装では，空白，タブ，改行文字を取り除く簡単な正規化しか行っていない．そのため，用いている変数は異なるが処理内容自体は同じであるコード片を同値であると判定できない．よって，そのようなコード片に対しては修正漏れの指摘ができない．

全ての変数やリテラルを同一の特別な文字に置き換える正規化を行えば，より多くの修正漏れを検出できるだろう．しかし，そのような正規化を行うためには，ソースファイル全体に対する構文解析を行う必要がある．現在は，svn diff コマンドを用いているため，各リビジョン間において差分の解析を行うコードの量は多くない．よって，大規模なシステムからでも短時間でマイニング処理を行える．全て

表 6　CCFinder と Nicad が検出したコードクローンの数
Table 6　Number of code clones detected by CCFinder and Nicad

(a) HTTPD

検出ツール	クローンセット数	コードクローン数
CCFinder	1,004	3,435
Nicad	117	324

(b) FreeBSD

検出ツール	クローンセット数	コードクローン数
CCFinder	47,016	357,353
Nicad	3,871	138,233

[*10] コードクローンの集合．1 つのクローンセットに含まれる任意のコード片のペアは互いにコードクローンである．

のリビジョンのソースファイルをチェックアウトして（手元にダウンロードして）構文解析を行うことも可能であるが，処理時間が著しく長くなってしまう．

6.2 手作業による確認作業

この実験では，著者らが検出された修正漏れの確認作業を行った．しかし著者らは対象システムの開発者ではないため，バグ修正やリファクタリング等への分類が誤っている可能性がある．より正確に手法を評価を行うためには，対象システムの開発者に協力していただいて実験を行う必要がある．

6.3 実験で用いたしきい値

本実験では，支持度，確信度，一致箇所数の 1 つの組み合わせのみを修正漏れ検出のしきい値として用いた．例えば，支持度を 2 にするなど，より緩いしきい値を用いることにより，さらに多くの修正漏れが検出される．この実験では著者らが検出された修正漏れを手作業で確認したが，著者らは対象システムの開発者ではないために，確認作業には長い時間を必要とした．RQ1 と RQ2 で検出された修正漏れを確認するために要した時間は約 20 時間であり，これ以上検出数が多い場合には手作業による確認が困難であった．対象システムの開発者がこの確認作業を行った場合には，検出された修正漏れの真偽を素早く判断できるだろう．

6.4 比較対象ツール

この実験では，CCFinder と Nicad のデフォルトの設定を用いてコードクローンを検出した．しかし，検出する最小のコードクローンの大きさなどの設定を変更することで検出結果は異なるため，それらのツールがより多くの修正漏れを検出できる可能性がある．また，この実験では，提案手法が検出した修正漏れをコードクローン検出ツールが検出可能か調査したのみである．しかし，提案手法では検出されないが，コードクローン検出ツールでは検出される修正漏れも存在するだろう．なぜなら，提案手法は，過去に同様の修正が行われていることを必要とするが，コードクローン検出ツールは必要としないからである．つまり，本実験の結果と合わせると，提案手法とコードクローン検出ツールは相補的な関係にあるといえる．

7. おわりに

本論文では，ソースコード中から修正漏れを自動的に検出する手法を提案した．提案手法は，対象ソフトウェアの開発履歴情報を用いているため，検出されるためにはある程度以上の大きな重複コードでなければならない，という既存手法の弱点を持たない．

提案手法をツールとして実装し，オープンソースソフトウェアに対して実験を行った．その結果，多数のバグ修正漏れ，リファクタリング漏れ，機能変更・機能追加漏れ，コメント修正漏れを検出できた．今後は，システムの開発者に協力していただいて実験を行い，ツールの検出結果の正しさをより正確に評価する予定である．また，コードの正規化を工夫し，検出の精度も高めていく予定である．

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 (A)(課題番号 : 21240002)，萌芽研究 (課題番号 : 23650014, 24650011)，若手研究 (A)(課題番号 : 24680002) の助成を得た．

参考文献

[1] 泉田聡介，植田泰士，神谷年洋，楠本真二，井上克郎：ソフトウェア保守のための類似コード検索ツール，電子情報通信学会論文誌， Vol. 86-D-I, No. 12, pp. 906–908 (2003).

[2] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Transactions on Software Engineering*, Vol. 28, pp. 654–670 (2002).

[3] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code, *IEEE Transactions on Software Engineering*, Vol. 32, pp. 176–192 (2006).

[4] 森崎修司，吉田則裕，肥後芳樹，楠本真二，井上克郎，佐々木健介，村上浩二， 松井恭：コードクローン検索による類似不具合検出の実証的評価，電子情報通信学会論文誌 D， Vol. J91-D, No. 10, pp. 2466–2477 (2008).

[5] Roy, C. K. and Cordy, J. R.: NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pp. 172–181 (2008).