社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2010-31 (2010-10)

コードの生存期間を考慮したコードクローンと欠陥修正の関係調査

齋藤 晃† 吉田 則裕†† 松下 誠† 井上 克郎†

† 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番5号
†† 奈良先端科学技術大学院大学 情報科学研究科 〒630–0192 奈良県生駒市高山町8916-5
E-mail: †{a-saitoh,matusita,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

あらまし これまでにコードクローンと欠陥修正の関係について様々な報告がある．近年の研究においてコードクローンはソフトウェアの欠陥を引き起こす要因に当たらないという報告がされているが，この調査ではコードの生存期間について考慮していないという問題がある．そこで本稿では，コードの生存期間に着目してコードクローンが欠陥修正に含まれる割合の調査を行った．その結果，生存期間が短いクローンの方が欠陥修正に多く含まれることが分かった．

キーワード コードクローン，リポジトリマイニング，ソフトウェア進化

Investigation of Relationship between Code Clones and Defect Fixes by Considering Period of Code Existence

Akira SAITO†, Norihiro YOSHIDA††, Makoto MATSUSHITA†, and Katsuro INOUE†

† Graduate School of Science and Technology, Osaka University
1-5, Yamadaoka, Suita, Osaka, 565–0871, Japan
†† Graduate School of Information Science, Nara Institute of Science and Technology
8916-5, Takayama-cho, Ikoma, Nara, 630-0192, Japan
E-mail: †{a-saitoh,matusita,inoue}@ist.osaka-u.ac.jp, ††yoshida@is.naist.jp

Abstract Various reports exist on relationship between code clones and defects. Especially recent study said that code clones do not cause software defects. However, this study has a problem that the period of code existence is not considered. Consequently, we investigated what proportion of code clones included defect fixes by considering period of code existence. The result showed that defect fixes occurred more frequently in short-lived code clones than long-lived ones.

Key words Code Clone, Repository Mining, Software Evolution

1. はじめに

コードクローンとはソースコード中の全部又は一部の重複したコード片であり，様々なコードクローンの検出手法が提案されている [3] [6] [8]．コードクローンがソフトウェアに与える影響に関してさまざまな報告がある．古典的には，Fowler らは重複したコード片が存在することを不吉な匂い (Bad Smell) と定義し，メソッドの抽出などにより重複を除去するべきと述べている [4]．また，コードクローンはプログラムの可読性を低下させ，ソフトウェアの保守の観点で望ましくないために積極的に除去すべきだとも言われている [2] [7]．その一方，ソフトウェア中の欠陥修正によって変更されたコード片を特定し，それらがどの程度コードクローンを含んでいるのかを調べたところ，コードクローンが欠陥修正によって変更されたコードに含まれている割合は小さく，コードクローンは欠陥を引き起こす要因に当たらないという結果が報告されている [5]．既存研究の調査では公開されている欠陥情報管理システムから修正が完了した欠陥情報を取得し，取得した欠陥情報 (バグレポート) とリポジトリから修正に該当するコミットを対応付ける手法を用いている．対応付けたコミットによって修正されたコード片全てを欠陥コードとみなし，欠陥コードはどの程度コードクローンを含んでいるかを調査している．しかしこの調査では，全ての欠陥コードが単にコードクローンを含むかどうかのみ判断しているだけであり，コードがそれまでにどの程度の期間存在していたかどうかについては考慮していない．

我々は，コードクローンの分析においてコード片の生存期間が重要であると考えている．その理由として，既に修正が頻繁に生じているコード片は今後も修正が生じやすいと考えられる

こと，また，たとえコードクローンであってもよく知られたプログラミングロジックや長期間使われている実績のあるコードであれば，欠陥修正との関連は小さくなると考えられることが挙げられる．

そこで本稿では，既存研究で用いられた手法に加えコードの生存期間を考慮した上で欠陥修正がコードクローンを含む割合の調査を行う．調査の結果，生存期間が短いコードクローンほど欠陥修正に含まれる割合が高く，逆に生存期間が長いコードクローンは欠陥修正に含まれる割合が低くなることが分かった．

本稿の構成を以下に示す．2節では既存研究である Rahman らの手法について述べる．ここでは欠陥コードの特定と欠陥コードがコードクローンを含む割合の算出方法について説明する．3節では本稿で適用する手法について述べ，4節で結果の評価を行う．最後に5節で結論を述べる．

2. 関連研究

本節では既存研究である Rahman らの手法でどのように欠陥修正に含まれるとコードクローンの割合の調査を行ったかについて述べる．Rahman らの手法では下記の手順で解析を行っている．

（1） スナップショットの取得
（2） コードクローンの検出
（3） 欠陥修正と対応するコミットの取得
（4） 欠陥修正に含まれるコードクローンの割合を算出

その後，既存研究の問題点について述べる．

2.1 スナップショットの取得

版管理システムによって管理されているプロジェクトから，過去のソースコード群を取得する．多くの版管理システムはリビジョンと呼ばれる単位でソースコードを含むファイルの変更を管理している．リビジョン r に含まれる情報を形式的に定義すれば $r = \langle A, T, f_1, f_2, ..., f_n \rangle$ と表される．ここで A は修正を行った作業者の名前を，T は時刻を，f_i は修正が完了した後のファイル群を表す．全てのリビジョンを解析対象にすると計算時間・記憶容量が増大するため，既存手法では1ヶ月おきに各リビジョンをスナップショットとして取得している．スナップショットの集合 S は $S = \langle s_1, s_2, ..., s_n \rangle$ と表され，s_i はそれぞれ毎月の最も日が早いリビジョンを表す．それぞれのスナップショットに対して，その時点でのソースコードを全て取得する．

2.2 コードクローン検出

取得したそれぞれの時刻におけるソースコード群に対してコードクローン検出を行う．検出には DECKARD を使用する [6]．コードクローン検出ツールを実行すると出力としてコードクローンの集合 (クローンセット) が得られる．コードクローン検出ツールの実行結果 O は $O = \langle g_1, g_2, ..., g_n \rangle$ と表され，ここで g_i は1つのクローンセットであり，$g_i = \langle c_1, c_2, ..., c_n \rangle$ と表せる．c_i はクローンとなるコード片であり $c_i = \langle s_j, f_k, l_s, l_e \rangle$ と表せる．s_j はコード片が含まれるスナップショットを，f_k はコード片が含まれるファイルを，l_s と l_e はコード片の開始行と終了行を表す．コードクローン検出処理によって，全てのコード片はコードクローンであるかユニークなコード片のどちらかに分類される．

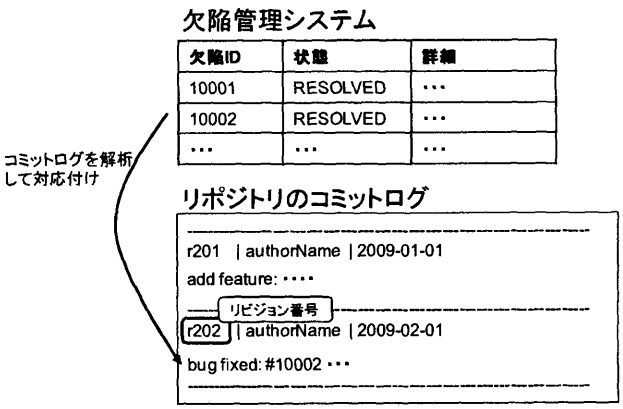


図1 欠陥情報とリビジョンの対応付け

ここでコードクローン検出ツール DECKARD はパラメータとして〈最小トークン長，類似度，ストライド〉の3つを与えることができるが，既存手法では $\langle 50, 1.0, 2 \rangle$，$\langle 50, 0.99, 2 \rangle$ の2通りの実験結果を報告している．

2.3 欠陥修正と対応するコミットの取得

欠陥を含むコード片がどこに存在するのか分かっていることが理想的であるが，一般に欠陥となるコード片の位置を特定することは困難である．そのため BugZilla 等の欠陥情報管理システムに登録されている修正済みのバグレポートを取得し，その欠陥の修正時に変更されたコード全てを欠陥コード (Buggy Code) とみなす．欠陥情報管理システムに登録されているバグレポートから変更されたコードを特定するには，リポジトリから欠陥が修正されたコミットとバグレポートを対応付ける必要がある．この対応付けを行うために図1で示すように欠陥情報管理システムに登録されている欠陥 ID とリポジトリ内に保存されているコミットログを用いる．これらの対応付けの処理はヒューリスティックが用いられている [1]．代表的な対応付け基準は，

- 欠陥 ID がコミットログに含まれている．
- “bug”，“fixed” などの欠陥修正を表す単語が含まれている．

などである．欠陥修正情報とコミットの対応付けが出来れば，リポジトリからをバグレポートの修正に該当するコミットの前後の差分を diff コマンドによって取得できる．欠陥修正がリビジョン r で完了すれば，diff コマンドで得られたリビジョン r とリビジョン $r-1$ との差分が Buggy Code となる．

2.4 欠陥修正に含まれるコードクローンの割合を算出

それぞれのバグレポートと修正されたコミットの対応付けの完了後，該当するリビジョン r と最も近いスナップショット s を対応付ける．そしてスナップショット s 中の Buggy Code の中に含まれるコードクローンの割合を算出する．欠陥修正が行われたリビジョン r とスナップショット s の間にコードの変更が生じる可能性があるので，diff コマンドを用いて調整を行う．例えば，リビジョン r とスナップショット s の間にコードが n 行追加された場合，追加された行以降のコードの対応付けを n

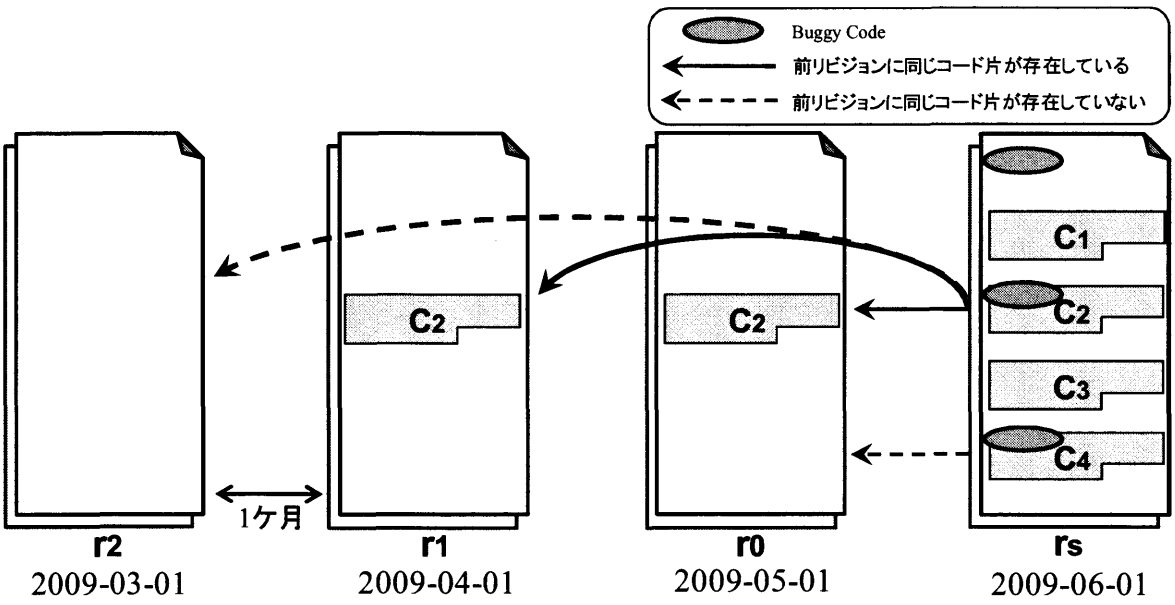


図 2　コード片の生存期間の取得

行ずらして行う．

2.5　既存手法の問題点

既存手法における問題点を以下に挙げる．

- コード片の存在期間について考慮していない．
- Buggy Code を特定したリビジョンとスナップショットとの期間が長いと誤差が大きくなる．

既存手法では，全てのコード片がコードクローンであるかユニークなコード片であるかのどちらかに分類してそれらがどの程度 Buggy Code に含まれているかを調査したが．コードがそれまでにどの程度の期間存在していたかどうかについては考慮していない．コードの生存期間が重要である理由は，既に修正が頻繁に生じているコード片は今後も修正が生じやすいと考えられ，また，たとえコードクローンであってもよく知られたプログラミングロジックや長期間使われている実績のあるコードであればそれらのコードは長期間存在して欠陥修正が行われにくいと考えられるからである．

また，Buggy Code を特定したリビジョンとスナップショットとの期間の間が長く，かつその間に変更が多く加わっているとスナップショットの時点ではコードクローンであったコード片が修正時にはコードクローンでなくなっている可能性がある．そのため欠陥修正に含まれるクローンの割合の計算結果が誤差が生じる問題がある．

3. 実験方法

前述の問題点を解決するために，以下の条件を変更した実験を行う．

（1） Buggy Code に一部でも含まれていたコードクローンに対し，そのコードクローンに包含されるコード片の生存期間かを取得する．

（2） スナップショットを 1ヶ月おきでなく欠陥 ID と欠陥修正のコミットの対応付けが完了した全てリビジョンにおいてコードクローン検出を行う．

このように条件を変更してコードクローンに包含されるコード片の生存期間を取得した後，生存期間の長いコード片と短いコード片で欠陥修正に含まれる割合が異なるかどうかを調査する．

また，欠陥 ID とコミットログの対応付けに関しては既存手法ではヒューリスティックな手法を用いており厳密に定義されていないので．ここでは最も単純に以下のようにして行う．

- コミットログに該当する# で始まる欠陥 ID(# 100001 等) が含まれていれば，その欠陥 ID を持つバグレポートとコミットログを対応付ける．

以降では Buggy Code に含まれていたコードクローンの生存期間の算出方法について説明する．

3.1　コードクローンの生存期間の取得方法

Buggy Code に含まれていたコードクローンの生存期間の算出は，以下の手順を用いて行う．

（1） Buggy Code 内に含まれるコードクローンの抽出と重複率の算出

（2） コード片の変更の有無を過去のリビジョンと比較

（3） コード片の生存期間を算出

3.2　Buggy Code 内に含まれるコードクローンの抽出と重複率の算出

Buggy Code のコードは位置は diff コマンドによって取得され，コードクローンも検出時にコード位置が取得されている．これらのコード位置の情報より Buggy Code 内に含まれるコードクローンを特定し，以下のようにして重複率 d を求める．

$$d = \frac{l_{buggy}}{l_e - l_b} \tag{1}$$

ここで l_s，l_e はコードクローンの開始行と終了行を，l_{buggy} はコードクローンの開始行と終了行の中で Buggy Code が含まれる行数である．また，コードクローンは検出時にコードクロー

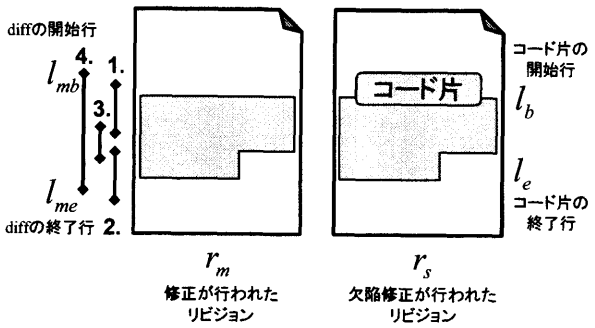


図 3 diff による変更の有無の判定

ンが含まれるリビジョン．ファイル名も取得する．

3.3 コードの変更の有無を過去のリビジョンと比較

手順 1 で抽出されたコードクローンに包含されるコード片に最も新しく変更が加えられた時刻を算出する．図 2 で示すようにコードクローンに包含されるコード片が得られたリビジョン r_s より約 1ヶ月毎に過去のリビジョン群 $r = \langle r_0, r_1, ..., r_n \rangle (r_i < r_s)$ を取得する．取得したリビジョン群から下記の条件を満たすリビジョン r_m を取得する．

$$r_m = \max_i r_i * \delta_{diff}(r_i, r_s, f, l_b, l_e) \quad (2)$$

ここで δ_{diff} は，リビジョン r_s 中のファイル f 内の開始行 l_b，終了行 l_e にあるコード片がリビジョン r_i まで変化がなければ 1，そうでなければ 0 を返す関数と定義する．つまり r_m はコードクローンに包含されるコード片のリビジョンより古く，かつその中で最も新しい変更が生じたリビジョンを表す．コードの変化は diff コマンドにより変更のあった開始行 l_{mb}，終了行 l_{me} が得られるので，図 3 に示すようにコードクローンに包含されるコード片の開始行 l_b，終了行 l_e との間に以下の関係が満たされれば，変更があったと判定する．

（1） $l_b > l_{mb}$ かつ $l_e \geqq l_{me}$：コード片の開始行をまたいだ変更

（2） $l_b \leqq l_{mb}$ かつ $l_e < l_{me}$：コード片の終了行をまたいだ変更

（3） $l_b \leqq l_{mb}$ かつ $l_e \geqq l_{me}$：コード片の内部の変更

（4） $l_b > l_{mb}$ かつ $l_e < l_{me}$：コード片全体を包含する変更

上記の 4 つのそれぞれの条件は，図 3 において各番号の l_{mb}，l_{me} の位置に差分が存在した時に対応している．

3.4 コード片の生存期間の算出

手順 2 で求めたリビジョン r_m と r_s からそのコミットが行われた日時を取得する．その両者の日時の差を，コードクローンの生存期間として算出する．

表 1 調査を行ったプロジェクトの概要

プロジェクト名	Gimp	Evolution
バグレポート数	698	11973
対応付けたリビジョン数	387	1676
最も古いリビジョンの日付	2004-01-11	2005-04-13
最も新しいリビジョンの日付	2009-01-23	2010-08-20

4. 適用実験

本節では前節で述べた実験を行った際の条件とその結果について述べる．

4.1 実験条件

既存研究では 4 つのオープンソースソフトウェアを実験対象としていたが，今回の実験ではそのうち 2 つのオープンソフトウェアを実験対称にした．対象とした 2 つのオープンソースソフトウェアの概要を以下に示す．

a） Gimp

オープンソースソフトウェアの画像編集ソフトウェア，2010 年 7 月時点でファイル数 2831，約 946kLOC

b） Evolution

Gnome デスクトップ環境に付随している標準メールクライアント，2010 年 2 月時点でファイル数 1127，約 456kLOC

いずれの対象もバグレポートは BugZilla で管理され，ソースコードは Git リポジトリによって管理されている．BugZilla で管理されているバグレポートのうち，下記の 2 つの条件を満すもののみを抽出した．

- バグレポートの “Status” が “RESOLVED”，“VERIFIED”，“CLOSED” のいずれかに設定
- バグレポートの “Resolution” が “FIXED” に設定

コードクローン検出には既存手法同様に DECKARD を使用した．パラメータ〈最小トークン長，類似度，ストライド〉は $\langle 50, 1.0, 2 \rangle$ を用いた．

4.2 実験結果

各プロジェクトにおける抽出したバグレポートの数，対応付けを行ったリビジョンの数を表 1 に示す．その後バグレポートと修正を行ったリビジョンを対応付け，その中で欠陥コードに含まれるコードクローンを抽出した．抽出したコードクローンの数とその生存期間を表 2 に示す．ここでは生存期間最大とは，コードクローンに該当するコード片が修正されるまでの最も長い期間を日数で表したものである．

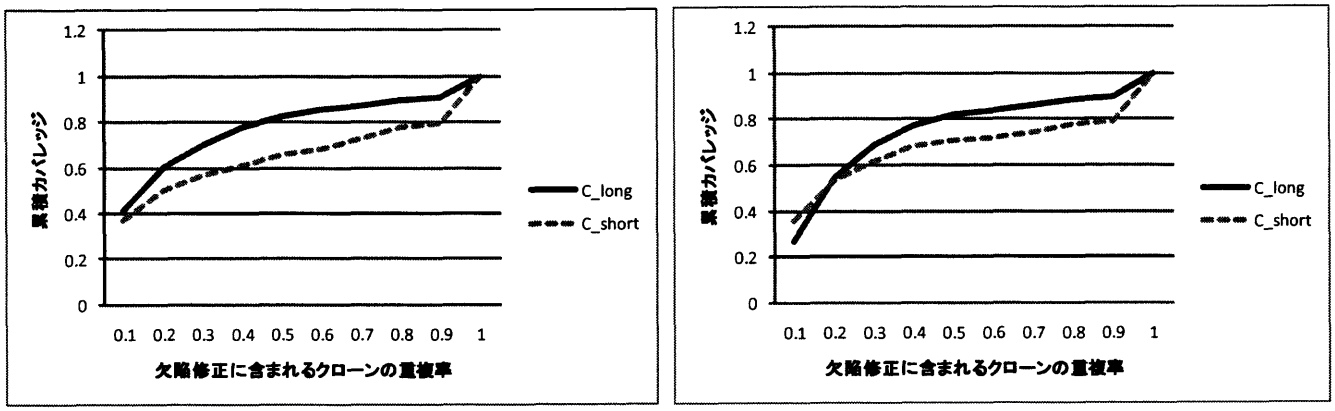
次に生存期間を取得したコード片を以下の 2 つのグループに分類する．

- C_{long} … 生存期間 (中央値) よりも生存期間が長いコード片のグループ
- C_{short} … 生存期間 (中央値) よりも生存期間が短いコード片のグループ

それぞれのグループにおいて，各コードクローンに包含されるコード片が，BuggyCode 内に何割含まれていたのかを算出する． Gimp プロジェクトと Evolution プロジェクトにおいて欠陥修正コード内に占めるクローンの割合を図 4 に示す．

表 2 欠陥情報に含まれるコードクローンの概要

プロジェクト名	Gimp	Evolution
コードクローンの数	1093	1114
生存期間最大 (日)	3460	3812
生存期間最小 (日)	32	41
生存期間中央値 (日)	574	1791



(a)Gimp プロジェクト　　(b)Evolution プロジェクト

図 4　欠陥修正コード内でクローンが占める割合

図 4 の横軸は欠陥修正に含まれるクローンの重複率であり，これはコードクローンを含む Buggy Code の全体行のうち，何割がコードクローンであったかを表している．縦軸はそれらのコードクローンが全体の何割を占めているかの累積カバレッジであり，欠陥修正に含まれるクローンの重複率 d 以下の欠陥修正が全体の何割存在するかを表している．図 4 では，Gimp, Evolution プロジェクトともに，生存期間が長いグループ C_{long} のほうが欠陥修正に多く存在する傾向があることが分かる．例えば，Gimp プロジェクトにおいてクローン率 0.4 以下の欠陥修正は，生存期間が長いコード片のグループでは，78.0%を占めていたが，生存期間が短いコード片のグループでは 61.2%しか占めていない．同様に Evolution プロジェクトでもクローン率 0.5 以下の欠陥修正は生存期間が長いコード片のグループでは 82.0%を占めていたが生存期間が短いコード片のグループでは 71.0%であった．

4.3　既存手法との結果の比較

既存手法においても欠陥コードが含むコードクローンの割合を示しており，Gimp プロジェクトでは約 2 割の欠陥修正がコードクローンを含んでいた．既存手法では類似度 1.00 と類似度 0.99 でコードクローンを検出した結果を示しているが，どちらもほぼ同じ分布を表していた．既存手法で示された欠陥コードが含むコードクローンの割合を，コードクローンの生存期間に着目して分類することで図 4 に示すような差異を得ることができた．

4.4　考　　察

今回の実験における問題点・考慮すべき点を述べる．

4.4.1　コード片としての生存期間の取得方法の妥当性の検討

今回の手法ではコード片の生存期間の取得方法として，過去のリビジョンでコード片の位置に該当する場所に変化があるかを調べることで判断した．しかし，図 5 のように，リビジョン r_s より過去のリビジョンにおいて，リビジョン r_s に存在するコード片と別の位置にコード片が出現した場合，コード片そのものが存在する期間が取得できない．ソースコード全体からコード片が存在してるかどうかを判定するためには，過去のリビジョンにおいてもコードクローン検出を実行しそれらの存在

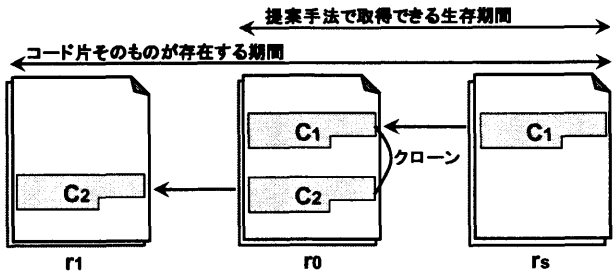


図 5　提案手法で取得できる生存期間

する期間も考慮する必要がある．

4.4.2　コードクローンでないコード片に対しての生存期間の検討

本手法では生存期間が短いコードクローンはより欠陥修正が生じる頻度が高いかどうかを調査したが，コードクローンでない一般のコード片に対しては調査を行っていない．コードクローンと同様に，生存期間が短いコード片の方が欠陥修正が多くなるかどうかを調べ，どの程度コードクローンとの差異が生じるかを調べる必要がある．そのためにはコード片の開始位置を終了位置を設定する必要がある．

4.4.3　欠陥情報と対応付けが完了したリビジョン数の不足

コミットログを用いて欠陥管理システムに登録されている欠陥情報と欠陥修正に該当するコミットの対応付けを行ったが，対応付けが完了した欠陥情報は全体の一部しかない．Gimp では約 55%，Evolution は約 14%の欠陥情報しか対応付けできていない．そのため，欠陥情報全体の性質を把握できているわけではない．この問題は既存手法も同様であり，対応付けをより正確に行える手法を適用し，既存手法と本実験の両者の方法で再度比較検討する必要がある．

5.　おわりに

本稿では，コード片の生存期間を考慮してコードクローンは欠陥修正にどの程度含まれるかを調査した．既存手法である Rahman らの手法は公開されている欠陥情報管理システムから欠陥情報を取得し，コミットログを用いて欠陥修正が行われた

リビジョンを対応付ける手法を用いている．さらに欠陥修正によって修正されたコード片がどの程度コードクローンを含むのかを調査した．本手法では既存手法に加えてコードクローンの生存期間に着目した．生存期間が短いコード片のグループと生存期間が長いコード片のグループの２つに分割し，それぞれが欠陥修正に含まれている割合を調査したところ，生存期間が短いコード片のグループの方が欠陥修正に多く含まれるという結果が得られた．今後はコードの生存期間の取得方法を変更し，コード片がソースコード中に含まれる期間を算出して同様の実験を行うことを考えている．また，コードクローンでないコード片に対しての生存期間を取得し，それがコードクローンであるコード片と差異が存在するか等を検討する予定である．今回使用したコードクローン検出ツールは既存手法と同じにするためにDECKARDを用いたが，CCFinder等の異なるコードクローン検出ツールでも同様の実験を行うことを考えている．

謝辞 本研究は一部，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)，基盤研究 (C)(課題番号:22500026)，研究活動スタート支援 (課題番号:22800040) の助成を得た．

文　献

[1] A. Bachmann and A. Bernstein. Data retrieval, processing and linking for software process data analysis. Technical report, Dynamic and Distributed Information System Group, Departmenet of Informatics, University of Zurich, 2009.

[2] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced clone-analysis to support object-oriented system refactoring. In *Proceedings of the 7th Working Conference on Reverse Engineering(WCRE'00)*, pp. 98–107, Los Alamitos, USA, 2000.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of the IEEE International Conference on Software Maintenance(ICSM'98)*, pp. 368–377, Los Alamitos, USA, 1998.

[4] M. Fowler. *Refactoring: improving the design of existing code.* Addison Wesley, 1999.

[5] F.Rahman, C.Bird, and P. Devanbu. Clones: What is that smell? In *Proceedings of the Seventh IEEE Working Conference on Mining Software Repositories(MSR' 10)*, pp. 72–81, Cape Town, South Africa, 2010.

[6] L. Jiang, Z. Su G. Misherghi, and S. Glondu. DECKARD :scalable and accurate tree-based detection of code clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE' 07), pp. 96–105, Minneapolis, USA, 2007.

[7] L. Jiang, Z. Su, and E. Chiu. Context-based detection of clone-related bugs. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering(ESEC-FSE '07)*, pp. 55–64, Dubrovnik, Croatia, 2007.

[8] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, Vol. 28, No. 7, pp. 654–670, 2002.