特別研究報告

題目

ソースコードの重複関係が修正に及ぼす影響の調査
-様々な条件の下で計測を行った結果の比較-

指導教員
楠本 真二 教授

報告者
佐々木 唯

平成 23 年 2 月 16 日

大阪大学 基礎工学部 情報科学科

平成 22 年度 特別研究報告

ソースコードの重複関係が修正に及ぼす影響の調査
-様々な条件の下で計測を行った結果の比較-

佐々木 唯

内容梗概

ソースコード中で内容が一致または類似したコードが存在すると, その一部に修正が必要な場合, 重複関係にあるコードに対して同様の修正が必要になる可能性がある. このように重複コードの存在は一般的にソフトウェアの保守に影響を及ぼすと考えられており, これまでに重複コードと修正の関係について調査する研究が行われている. しかし既存の研究では調査の方法や, 用いる重複コード検出ツール, 対象ソフトウェアも様々で, 一貫した結論を導きだすことが難しい. 実際に個々の調査での結論も様々である.

本研究では, 3 種類の調査手法と, 4 種類の重複コード検出ツールを用いて, 同一のソフトウェアに対してそれらの組み合わせを変えることで結果にどのような違いが出るのか調査した. また, ソースコード修正の中でも特にバグの修正による作業量はソフトウェアの保守に大きく影響を与えると考えられる. そこで, バグの修正情報のみを対象とした実験と, 全ての修正情報を対象とした実験を行い, それぞれ対象としたソフトウェアごとに結果の比較を行った.

調査の結果, 全ての組み合わせで一貫した結論の得られたソフトウェアはなかったが, いくつかの組み合わせはどのソフトウェアについても過半数の組み合わせが示す結果と一致した. また, 重複コードの検出単位が異なる場合, 計測結果に違いが出やすいという結果が得られた.

主な用語

重複コード
ソフトウェア保守
バージョン管理システム

目 次

1　まえがき

近年, ソフトウェアの大規模化, 複雑化に伴い, ソフトウェアの保守に要する作業量が増大する傾向にある. ソフトウェア開発において保守の作業量を増大させる原因の 1 つに重複コードの存在が挙げられる. ソースコード中で内容が一致または類似したコードが存在すると, その一部に修正が必要な場合, 重複関係にあるコードに対して同様の修正が必要になる. 重複コードがソフトウェア保守に及ぼす影響についてこれまでに様々な研究がなされており, 重複コードの検出手法の提案, 自動的に検出の実行や可視化を行うツールの開発, リファクタリング支援など多岐にわたる.[1][2][3][4][5][6]

しかし, 重複コードがソースコードの修正にどの程度影響を与えているのかを定量的に調査した研究は少ない. また, 既存の研究で提案されている手法は調査の粒度や修正量の定義が異なり, 対象ソフトウェアも様々である. 更に重複コードに普遍的な定義が存在しないため, 同じソースコードを入力とした場合でも重複コード検出ツールによって重複コードと判断される箇所が異なる場合もある. 研究によってそれぞれ異なった結論が導かれているが, その原因は対象ソフトウェアが共通のものでないこと, 各提案手法による修正量の定義の違い, 利用する重複コード検出ツールの違いにあると考えられる.

本研究では, 同一のソフトウェアに対して重複コード検出ツールと調査手法の組み合わせを変えることで, 結果に違いが出るかどうか調査を行った. 比較する重複コード検出ツールとして, *CCFinder*[2], *CCFinderX*[3], *Simian*[4], *Scorpio*[5] の 4 種類を用いた. また, 比較する手法として, 佐野らの手法 [7], Krinke の手法 [8], Lozano らの手法 [9] の 3 種類を用いた. 更に, ソースコードに対する修正の中でも, バグの修正による作業量が多ければソフトウェアの保守コストも大きくなると考えられ, これに基づいて重複コードとバグ修正の関係を調査した研究も行われている [10]. 今回適用した 3 種類の手法を用いた既存の研究では, 修正の内容をバグを含んでいたものに限定した対象については行われていないため, 実験対象のソフトウェアとしては規模の異なる 5 種類を選び, そのうちの 1 つはバグの修正情報のみを用いた.

調査の結果, バグの修正のみを対象とした場合, 重複コードへの修正が多い傾向がみられた. 全ての修正情報を対象とした場合は, ソフトウェアごとに傾向の異なる結果となった. また, いずれのソフトウェアについても, 調査手法と検出ツールの全ての組み合わせで同じ結果は得られなかった. しかし, 重複コードと非重複コードどちらに対して修正が多いか, あるソフトウェアにおいて各組み合わせの結果の過半数を占める方を正しい結果とするならば, いくつかの組み合わせはどのソフトウェアについても正しい結果を得られた. 更に, 重複コードの検出単位が異なるとき, 調査手法が同じであっても計測結果に違いをもたらす傾向があるという結果が得られた.

以降 2 節で関連研究について述べ, 3 節では, 重複コード, 本研究に用いた検出ツール, バージョン管理システム *Subversion*[11], 及びソースコード正規化ツール *CommentRemover*[12] について述べる. 4 節では各重複コード検出ツールについて, 5 節では各手法における計測方法や計測値の定義について説明する. 6 節では, オープンソースソフトウェアを対象に各手法, 各重複コード検出ツールによって計測した実験の内容, 結果について述べ, その考察を 7 節で行う. 8 節では本研究で得られた結果の妥当性について述べ, 最後に 9 節で本研究のまとめと今後の課題について述べる.

2　関連研究

門田ら [13] は, COBOL で記述された大規模なレガシーソフトウェアに対し, 信頼性・保守性と重複コードとの関係を分析している. 重複コードを含むモジュールは含まないモジュールより信頼性が約 40%高いが, 200 行を超える大きな重複コードを含むモジュールは信頼性が低下し, また重複コードを含むモジュールは含まないモジュールより改版数が約 40 %高い (保守性が低い), と報告している.

Lozano ら [9] は, ソフトウェア保守に対する重複コードの影響について, メソッド単位で調査を行った. その評価指標として, likelihood（あるメソッドに対して変更の行われる割合）, impact（あるメソッドが変更される際, 同時に変更されるメソッド数の割合）, work（likelihood と impact の積）を定義し, work を保守コストと定義した. 4 つのオープンソースの Java プロジェクトについて, 同一メソッドで重複コードを含む期間と含まない期間, 常に重複コードを含むメソッドと常に含まないメソッド, それぞれの保守コストを比較した結果, likelihood はどちらもあまり変わらなかったが, impact に関しては重複コードを含んでいるほうが大きくなる傾向を示すものがいくつかあった. そして, 重複コードの存在期間の割合が高くなると, work が急激に増加したと報告している.

Krinke[8] は, もし重複コードが非重複コードの部分より安定性に欠くなら, 保守において重複コードに要するコストが高いと仮定し, システムの進化において, 重複コードの安定性について調査を行った. 調査対象は 5 つの大規模なシステムから, 1 週間ごとに区切ったバージョンを 200 ずつ抽出して使用した. それらの調査対象に対して重複コードと非重複コードの部分のそれぞれに対して行われる追加, 変更, 削除の行数を計測し, 重複コードと非重複コードの部分のそれぞれに対する割合を比較した. その結果, 重複コードの方が追加, 変更, 削除が行われる割合の平均が低く, また, 追加, 変更, 削除が行われる割合が重複コードの方が低い週が多かった. この結果から, 重複コードの方が非重複コードの部分より安定しており, 一般的に重複コードの保守に要するコストの方が非重複コードの部分の保守に要するコストより高いとは仮定できない, と報告している.

Eick ら [14] は, ソースコードを運用・保守していくうちに Code Decay (ソースコードが保守し難くなること) が引き起こされているのか調査している. Eick らは, Code Decay を示す指標として, 変更によって増加 (または減少) した行数や変更に要した時間, 変更に関わった開発者数, ある期間中にあるモジュールに関わる変更を行った数やある変更に関わったファイル数など, 様々なものを提案した. そして, 15 年にわたって運用された大規模なソフトウェアについて調査を行った結果, 1 つの変更に要するコストが増加していく傾向にあると報告している.

Nils ら [15] は, コーディングスタイルを除けば完全に一致する重複コードに関して, 重複

コードが生成され発展する様子を個々のコード片に着目してモデル化する手法を提案している. また, その提案手法を 9 つのオープンソースソフトウェアに対して適用し, 重複コードの発展の様子を調査している. その結果, 重複コードの割合は時間経過とともに減少していること, 重複コードは平均で約 1 年以上重複コードとして存在していること, また, 重複コードに一貫性のない変更が加わった場合, その変更がのちのバージョンにおいて一貫性のある変更に修復されることは少ないことなどを報告している.

Lozano ら [16] は, バージョン管理システム *CVS* で管理されているソフトウェアに対して, メソッドが変更された期間, ある期間においてメソッドが含んでいる重複コードの割合, 及びある期間において複数のメソッドが共有している重複コードの数を算出するツール *CloneTracker*[17] を作成した. また作成したツールをある Java ソフトウェアに適用した結果, 全てあるいは一部の期間で重複コードを含んでいたメソッドは, 全ての期間で重複コードを含んでいないメソッドと比較して, より高い頻度で変更が加えられていると報告している.

3　準備

3.1　重複コード

重複コードとはソースコード中に存在する同一, あるいは類似するコード片のことで, コードクローンと呼ばれる場合もある. ただし, どのような基準で類似していると判断するかは検出手法や検出ツールによって異なる.

また, 重複コード間の類似の度合に基づき重複コードを次の 3 つのタイプに分類することができる [19][20].

Type-1

空白やタブの有無, 括弧の位置などのコーディングスタイルに依存する箇所を除いて, 完全に一致する重複コード.

Type-2

変数名や関数名などのユーザ定義名, また変数の型などの一部の予約語のみが異なる重複コード.

Type-3

Type-2 における変更に加えて, 文の挿入や削除, 変更が行われた重複コード.

3.1.1　発生の原因

重複コードがソフトウェアの中に作りこまれる, もしくは発生する原因として次のようなものが挙げられる [2][21][22].

コーディングスタイル

規則的に必要なコードはスタイルとして同じように記述される場合がある. 例えば, ユーザインターフェース処理を記述するコードなどである.

定型処理

定義上簡単で頻繁に用いられる処理. 例えば, 所得税の計算や, キューの挿入処理, データ構造アクセス処理などである.

適切な機能の欠如

抽象データ型やローカル変数を用いることができないプログラミング言語を開発に用いている場合, 同じようなアルゴリズムを用いた処理を繰り返し書かなくてはならないことがある.

パフォーマンス改善

リアルタイムシステムなど時間制約のあるシステムにおいて, インライン展開などの機能が提供されていない場合に, 特定のコード片を意図的に繰り返し書くことによってパフォーマンスの改善を図ることがある.

コード生成ツールの生成コード

コード生成ツールによって生成されるコードは, あらかじめ決められたコードをベースにして自動的に生成される. このため, 類似した処理を目的としたコードを生成した場合, 識別子名等の違いを除き, 類似したコードが生成される.

複数のプラットフォームに対応したコード

複数の OS や CPU に対応したソフトウェアは, 各プラットフォームを対象に生成されたコード部分に重複した処理が存在する傾向が強い.

偶然

偶然に, 開発者が同一のコード片を書いてしまう場合もあるが, 大きな重複コードになる可能性は低い.

3.1.2 検出ツールの分類

重複コードを検出する手法はこれまでに多数提案されている. またそれらを実装した, 重複コードを自動的に検出するツールも多数開発されている. これらの検出技術は重複コードをどの単位で検出するかによって, 大まかに以下の 5 つに分類することができる [1].

行単位の検出

行単位の検出は, ソースコードを行単位で比較して重複コードを検出する手法であり, 閾値以上連続して一致する行を重複コードとして検出する. 他の手法と異なり, ソースコードに対する事前処理を必要としない. このため, 他の手法と比べて高速に重複コードを検出可

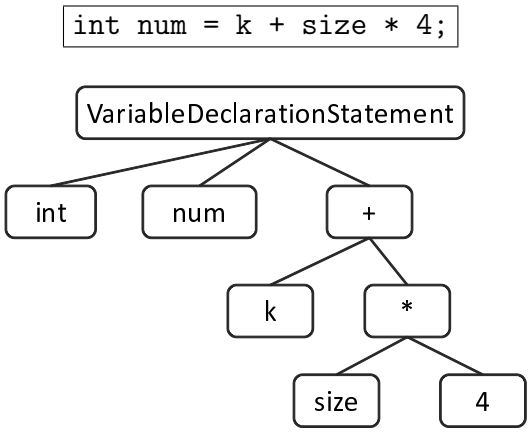


図 3.1: 抽象構文木の例

能である. しかし, 同じ処理を行っているコードであっても, 例えば長い行を複数行に分割した場合と分割しなかった場合など, コーディングスタイルが違う場合は重複コードとして検出できないという弱点を持つ.

字句単位の検出

字句単位の検出は, ソースコードを字句単位に分割し, 閾値以上連続して一致する字句の部分列を重複コードとして検出する手法である. 行単位の検出と異なり, コーディングスタイルのみ違う場合なども重複コードとして検出することが可能である. ソースコードを検出用の中間表現に変換する必要がないため, 高速に重複コード検出を行うことができるという利点もある. また, 字句に事前処理を行うことで変数名などのユーザ定義名のみ異なる重複コードなども検出可能となる.

抽象構文木を用いた検出

抽象構文木 (図 3.1) を用いた検出は, ソースコードに対して構文解析を行い, 抽象構文木を構築した後, その抽象構文木を用いて重複コードを検出する手法であり, 抽象構文木上の同形の部分木が重複コードとして検出される. 抽象構文木を構築するという事前処理を要するため, 行単位の検出や字句単位の検出と比べ, 時間的, 空間的コストが高くなるという欠点はあるものの, ある関数定義の終わりから次の関数定義の先頭までの類似部分など, プログラムの構造を無視した重複コードを検出しないという利点がある.

```
1: void sample() {
2:   for ( int i = 0 ; i < 10 ; i++ ) {
3:     System.out.println(i);
4:   }
5: }
```

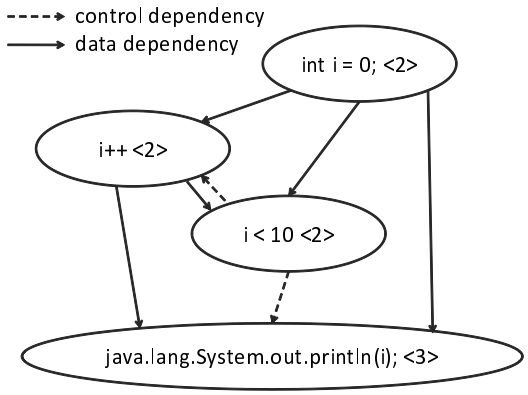


図 3.2: プログラム依存グラフの例

プログラム依存グラフを用いた検出

プログラム依存グラフ (図 3.2, [23]) を用いた検出は, ソースコードに対して意味解析を行い, ソースコードの要素間の依存関係を表すプログラム依存グラフを構築した後, そのプログラム依存グラフを用いて重複コード検出を行う手法である. プログラム依存グラフ上の同形部分が重複コードとして検出される. 抽象構文木を用いた検出と同様に事前処理を必要とするため, 時間的, 空間的コストが高くなるという欠点を持つ. ソースコードの順番が入れ替わっていても意味的に同一である重複コード (順序入れ替わり重複コード) などは意味的な処理を考慮しなければ検出できないが, この手法はこれらの重複コードを検出することができるという点が特徴として挙げられる.

順序入れ替わり重複コードの例を図 3.3 に示す. この例の場合, %で表されているコード片と, #で表されているコード片が順序入れ替わり重複コードとなる.

その他の技術を用いた検出

その他の技術を用いた検出手法として, プログラムのモジュール (ファイル, クラス, メソッドなど) に対してメトリクスを計測し, その値の一致または近似の度合いを検査することによって, そのモジュール単位での重複コードを検出する手法であるメトリクスを用いた検出

```
  fp3 = lookaheadset + tokensetsize;
  for (l = lookaheas(state); l < k; i++) {
%       fp1 = LA + i * tokensetsize;
%       fp2 = lookaheadset;
%       while (fp2 < fp3)
%             *fp2++ |= fp1++;
  }
```

(a) コード片 1

```
  fp3 = base + tokensetsize;
  ...
  if (rp) {
       while ((j = *rp++) >= 0) {
              ...
#      fp1 = lookaheadset;
#      fp2 = LA + j * tokensetsize;
#      while (fp1 < fp3)
#            *fp1++ |= *fp2++;
       }
  }
```

(b) コード片 2

図 3.3: 順序入れ替わり重複コード

や, プログラムの盗用の検出やプログラムの作者を特定することを目的とした, フィンガープリントやバースマークを用いた検出手法などがある.

3.2 バージョン管理システム

バージョン管理システムとは, 主にプログラム開発においてソースコードやその他のデータを管理するために用いるシステムである. バージョン管理システムは主に開発情報 (ソースコード, リソースなど) を開発者間で共有する機能, 及び開発履歴情報を保持する機能を提供する. バージョン管理システムを用いることで離れた場所にいる開発者間で開発情報の共有が容易になるという利点があるため, 商用ソフトウェア開発やオープンソースソフトウェアの開発など, 多数の開発者によってソフトウェア開発が行われる際に一般的に使用されている.

本研究で用いたバージョン管理システム *Subversion*[11] における上述の機能の概要は以下の通りである.

開発情報の共有

Subversion では, ソフトウェアの開発情報 (ソースコード, リソースなど) はリポジトリと呼ばれるデータ格納庫に保存される. 開発者はソフトウェアの開発を行う際, リポジトリから開発情報を手元にコピーし (この動作をチェックアウトと呼ぶ), 修正, 変更を加えた後, リポジトリに反映させる (この動作をコミット, あるいはチェックインと呼ぶ). コミットしたとき, 別の開発者が同じファイルに変更を加えていた場合は, ファイルが別の開発者によって変更されていることが通知されるので, 開発者は 2 つのファイルをマージし, コミットする. これにより別の開発者が行った変更を誤って上書きすることを防ぐことができる.

開発履歴情報の保持

Subversion では, 開発履歴情報をリビジョンと呼ばれる単位で保持している. リビジョンとは開発の状態を表す単位で, リポジトリがコミットを受け付けるたびに生成され, それぞれのリビジョンにはユニークな自然数 (リビジョン番号) が割り当てられる. 開発履歴情報にはソースコードなどの開発情報の他, 変更を加えた開発者名や日時, 変更が加えられたファイル名, 変更を加えた開発者の変更に関するメッセージ等のログも含まれている. 開発履歴情報を利用すれば, ソースコードに誤った変更を加えてしまった場合などに過去の状態に復元することが可能となる.

3.3 ソースコード正規化ツール *CommentRemover*

CommentRemover[12] は以下の 5 種類の正規化を実行できる.

- 空白行の削除.
- ブロックコメント (/*…*/) の削除.
- ラインコメント (//…) の削除.
- インデントの削除.
- 中括弧 ({ }) のみの行を削除し, 1 つ上の行に追加.

これらの正規化を行うか行わないか個別に設定することで, 5 節で述べる実験を行うにあたり, 手法によって指針の異なるソースコードの正規化に対応できる.

4　本研究で用いる重複コード検出ツール

本節では, 比較対象とする 4 種類の重複コード検出ツールについて述べる.

4.1　*CCFinder*

CCFinder は, 単一または複数のファイルのソースコード中から全ての重複コードを検出し, それらの位置情報を出力する. *CCFinder* は字句単位の検出に分類される. *CCFinder* の持つ主な特徴は次の通りである. [18]

細粒度の重複コードを検出

検出対象ソースコードに事前処理として字句解析を行うことにより, 字句単位での重複コードを検出する.

大規模ソフトウェアを実用的な時間とメモリで解析可能

例えば 10MLOC のソースコードを 68 分 (実行環境 Pentium3 650MHz RAM 1GB) で解析可能である.

細かい設定が可能

- 言語依存部分を取り替えることで, 様々なプログラミング言語に対応できる. 現在は, C, C++, Java, COBOL/COBOLS, Fortran, Emacs Lisp に対応している. またプレーンテキストに対しても, 分かち書きされた文章として解析可能となっており, 未対応の言語に対しても完全一致判定による重複コードは検出可能である.
- 重複コードは小さくなればなるほど偶然の一致の可能性が高くなるが, 最少一致トークン数を指定することでそのような重複コードの検出を防ぐことができる.

ある程度の違いは吸収可能

- ソースコード中に含まれる変数などのユーザ定義名, 定数をパラメータ化することで, その違いを吸収することができる.
- クラススコープや名前空間による複雑な名前の正規化を行うことで, その違いを吸収することができる.
- その他, テーブル初期化コード, 可視性キーワード (protected,public,private 等), コンパウンド・ブロックの中括弧表記等の違いも吸収することができる.

4.2 *CCFinderX*

CCFinderX は *CCFinder* のメジャーバージョンアップであり, *CCFinder* と同様に字句単位の検出に分類される. *CCFinder* からの改良点として以下のような点が挙げられる [3].

- マルチコア CPU 向けにマルチスレッド化.
- 抽象構文木ベースの前処理.
- 検索機能を追加.
- 対応している全ての言語を可能な限り等しくサポート.
- 利用者によるプログラミング言語の方言, 新しいプログラミング言語への対応が可能.
- 重複コードに関するメトリクスを使った分析に対応.
- 他のツールとの連携のために, TSV(タブ区切り形式) でデータを入出力.
- 複数のビューによる対話的な分析が可能.

4.3 *Simian*

Simian は, 行単位の検出に分類される重複コード検出ツールである. *Simian* の特徴として, 以下のような点が挙げられる [4].

様々な言語, 及びテキストにも対応

Java, C#, C++, C, Objective-C, JavaScript(ECMAScript), COBOL, Ruby, Lisp, SQL, Visual Basic, Groovy に対応している. また, これらの言語以外のファイルについても, テキスト形式のファイルであればテキストファイルとみなして類似する部分の検出を行うことができる.

少ないメモリで, 高速に検出

390,309 LOC, ファイル数 4,242 のソフトウェアに対して実行した結果, 使用メモリ 46MB, 実行時間 10 秒以内で重複コードを検出した.

細かい設定が可能

コメント, 空白, import 文, include 文, パッケージ宣言等を無視するように設定することで, 実用的でない重複コードの検出数を軽減することができる. この他に, 重複コードの最小行数の設定や, 文字の大文字小文字の違い, 変数名の違い, 数値の違い等を無視するかどうかなども設定することが可能である.

4.4 *Scorpio*

Scorpio はプログラム依存グラフを用いた検出に分類される重複コード検出ツールであり, Java に対応している. *Scorpio* の特徴として以下の点が挙げられる [5][23].

非連続重複コードが検出可能

プログラム依存グラフを用いて検出を行うため, 行単位の検出手法や字句単位の検出手法では検出できない順序入れ替わり重複コードなどの非連続重複コードを検出することが可能である.

グラフ探索方法の併用

プログラム依存グラフをたどって同形部分グラフを検出する際, グラフを探索する方法として, グラフを順方向に探索する方法 (フォワードスライス) と, グラフを逆方向に探索する方法 (バックワードスライス) の 2 種類が存在する. 一方では検出できない重複コードがもう一方では検出できる場合が起こり得るが, *Scorpio* では両方の探索方法を併用するため, 双方で共通して検出できる重複コードに加え, どちらか一方の探索方法でしか検出できない重複コードを検出することが可能である.

実用的な時間で検出可能

プログラム依存グラフを用いた検出ではソースコードの意味解析を行いプログラム依存グラフを構築するという事前処理を必要とするため, 他の検出手法と比べて検出に要するコストが高いという欠点がある. *Scorpio* では, プログラム依存グラフを探索する際に基点として用いる頂点に制限を設けることで, 検出の際の計算コストを最大 36%にまで削減した.

5　本研究で用いる手法

本節では比較対象とする 3 種類の手法について述べる.

3 種類の手法について, その違いを表 1 にまとめる.

計測対象リビジョンの違い

各手法における計測対象リビジョンを図 5.1 に示す. X などの値はリビジョン番号を表している. 佐野ら, Lozano らの手法では, 各計測対象リビジョンについて, 変更を加えられた直後のリビジョンとの間のソースコードの差分を検出する. Krinke の手法では, 1 週間ごとの最新リビジョンを計測対象リビジョンとし, 次の計測対象リビジョンとの間のソースコードの差分を検出する. 図 5.1 の場合は, 4 月 8 日時点での最新リビジョン $X+3$ と 4 月 15 日時点での最新リビジョン $X+3$ はソースコードの変更の行われていない別のリビジョンとみなしている.

なお, 各計測対象リビジョンに対してソースコードの差分の比較対象となるリビジョンを, 以降ではどの手法においても「次のリビジョン」として表現する.

正規化方法の違い

図 5.2 に, 中括弧を上げる正規化を行わなかった場合と行った場合の例を示す.

表 1: 手法の比較

項目＼手法	佐野らの手法 [7]	Krinke の手法 [8]	Lozano らの手法 [9]
計測対象リビジョン	修正を含むリビジョン (ファイルの追加除く)	1 週間ごとに抽出	修正を含むリビジョン (ファイルの追加含む)
正規化内容	コメントの削除 空白行の削除 インデントの削除 中括弧を上げる	コメントの削除 空白行の削除 インデントの削除	コメントの削除 空白行の削除 インデントの削除
計測単位	箇所 (連続する行)	行	メソッド
評価の指針	修正頻度	修正行数の割合	メソッドの変更割合 × 同時に変更された メソッドの割合

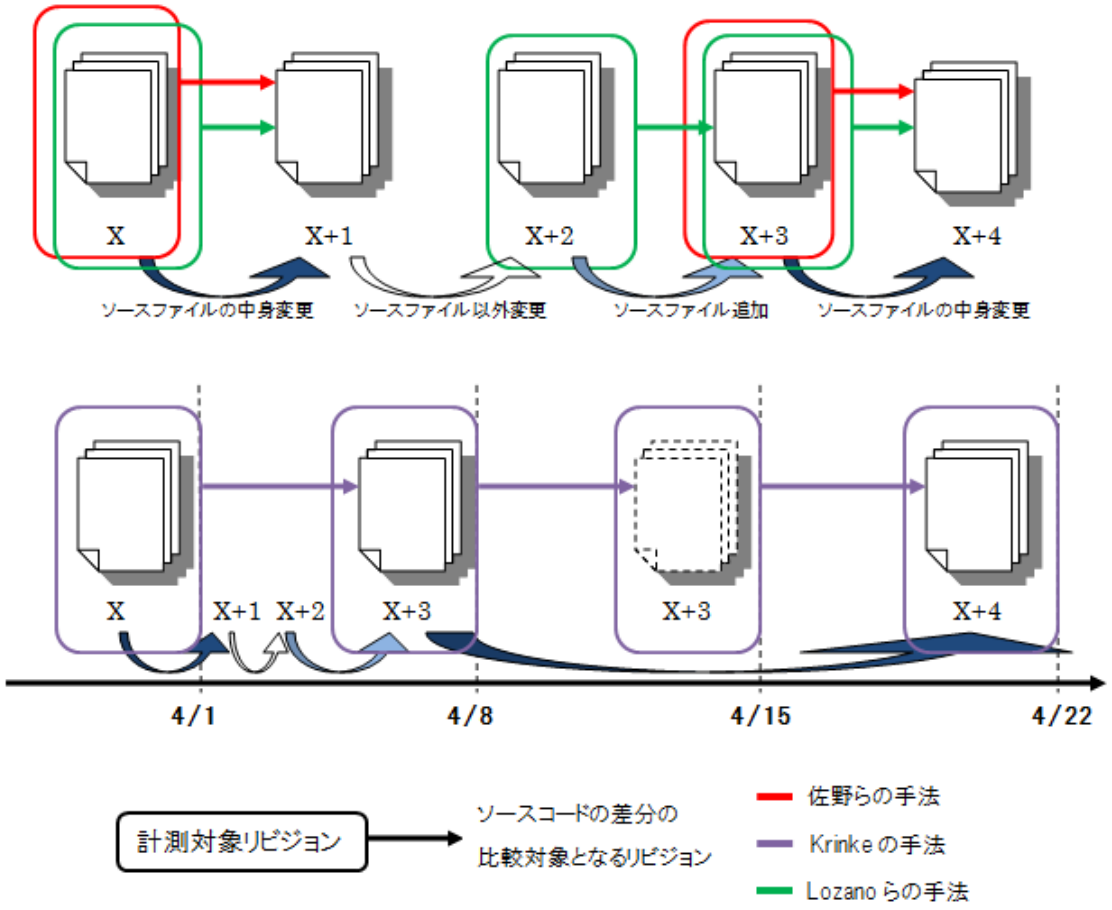


図 5.1: 各手法における計測対象リビジョン

```
1: public static void main(String[] args) {
2: methodA();
3: if(methodB(args[0] != 0) {
4: methodC(args[1]);
5: }
6: }
```

(a) 中括弧を上げる正規化を行わなかった場合

```
1: public static void main(String[] args) {
2: methodA();
3: if(methodB(args[0] != 0) {
4: methodC(args[1]);}}
```

(b) 中括弧を上げる正規化を行った場合

図 5.2: ソースコードの正規化の例

5.1 佐野らの手法

重複コードに対する修正頻度と非重複コードに対する修正頻度を, 修正箇所数を用いて計測する手法である. 修正箇所数と修正の頻度については, それぞれ 5.1.1 と 5.1.2 で述べる. 重複コードと非重複コードの修正頻度を比較することで, どちらに多くの修正が行われているのかを調査する手法である.

5.1.1 修正箇所数

連続して修正の加えられた行を 1 箇所の修正箇所と定め, 修正箇所の全ての行が重複コードの場合重複コードへの修正とみなし, 全ての行が重複コードでない場合重複コード以外への修正とみなす. また修正箇所に重複コードと重複コード以外の部分の両方が含まれる場合, 重複コードと重複コード以外へそれぞれ 1 度ずつ修正が加わったものとみなす.

5.1.2 修正頻度

計測対象リビジョンの集合を R とし, ある計測対象リビジョン $r \in R$ のソースコードの行数を $l(r)$, 重複コード関係にある行数を $lc(r)$, 重複コード関係にない行数を $ln(r)$ とする. また, r について, 重複コードに加えられた修正箇所数を $mc(r)$, 重複コード以外に加えられた修正箇所数を $mn(r)$ とする.

このとき, 次式で算出される MF_c を**重複コードへの修正頻度**, MF_n を**非重複コードへの修正頻度**と定義する.

$$MF_c = \frac{\sum_{r \in R} mc(r)}{|R|} \cdot \frac{\sum_{r \in R} l(r)}{\sum_{r \in R} lc(r)} \tag{1}$$

$$MF_n = \frac{\sum_{r \in R} mn(r)}{|R|} \cdot \frac{\sum_{r \in R} l(r)}{\sum_{r \in R} ln(r)} \tag{2}$$

5.2 Krinke の手法

開発期間のうち 200 週間分を抽出し, 1 週間ごとの最新リビジョンを計測対象リビジョンとした上で, 重複コード, 非重複コードの行数の割合を修正内容別に比較する手法である.

5.2.1 修正行数

コードの追加, 削除, それ以外の修正という内容別に行単位でソースコードの修正情報を得る. 計測対象リビジョン r と次のリビジョン間との差分を次の 6 種類の集合で表現する.

$$
\begin{aligned}
AC(r) &= \text{重複コードの位置に追加された行の集合} \\
AN(r) &= \text{非重複コードの位置に追加された行の集合} \\
DC(r) &= \text{削除された重複コードの行の集合} \\
DN(r) &= \text{削除された非重複コードの行の集合} \\
CC(r) &= \text{修正された重複コードの行の集合} \\
CN(r) &= \text{修正された非重複コードの行の集合}
\end{aligned}
$$

ただし $AC(r)$, $AN(r)$ は r の次のリビジョンに含まれる行の集合である. 追加の開始行が r における重複コードであれば全て重複コードへの追加行とし, そうでなければ全て非重複コードへの追加行とする. 追加された行そのものが重複コードを含んでいるかどうかを判定しているわけではない.

5.2.2 コードの割合

計測対象リビジョンの集合を R とし, ある計測対象リビジョン $r \in R$ の重複コードを含む行の集合を $C(r)$, 非重複コードの行の集合を $N(r)$ としたとき, 次式で算出される AC を**重複コードへの追加割合**, AN を**非重複コードへの追加割合**, DC を**重複コードの削除割合**, DN を**非重複コードの削除割合**, CC を**重複コードの修正割合**, CN を**非重複コードの修正割合**, と定義する

$$
AC = \frac{\sum_{r \in R} |AC(r)|}{\sum_{r \in R} |C(r)|} \tag{3}
$$

$$
AN = \frac{\sum_{r \in R} |AN(r)|}{\sum_{r \in R} |N(r)|} \tag{4}
$$

$$
DC = \frac{\sum_{r \in R} |DC(r)|}{\sum_{r \in R} |C(r)|} \tag{5}
$$

$$DN = \frac{\sum_{r \in R} |DN(r)|}{\sum_{r \in R} |N(r)|} \tag{6}$$

$$CC = \frac{\sum_{r \in R} |CC(r)|}{\sum_{r \in R} |C(r)|} \tag{7}$$

$$CN = \frac{\sum_{r \in R} |CN(r)|}{\sum_{r \in R} |N(r)|} \tag{8}$$

5.3 Lozano らの手法

重複コードを含むか否かという視点からメソッドを分類し, メソッドのある期間における保守コストの分布を比較する手法である.

5.3.1 メソッドの分類

各計測対象リビジョンについて, 存在するメソッドの位置情報（定義されているクラスの完全限定名, メソッドの開始行と終了行）, 及びシグネチャを取得する. 次に計測対象リビジョンに存在するメソッドと, その次のリビジョンに存在するメソッドについて, 以下の基準でメソッドの同一性を判定する. ただし, 最も類似するメソッドは, 比較するメソッドの行ごとに StrikeAMatch アルゴリズム [24] を適用して判定している.

1. 定義されているクラスの完全限定名とシグネチャの両方が完全一致するメソッドは同一メソッドとみなす.

2. 1. で同一メソッドの存在しなかった両リビジョンのメソッド間で, クラスの完全限定名が一致しシグネチャの異なる組み合わせのうち, 最も類似するものを同一メソッドとみなす.

3. 2. で同一メソッドの存在しなかった両リビジョンのメソッド間で, シグネチャが一致しクラスの完全限定名が一致しなかった組み合わせのうち, 最も類似するものを同一メソッドとみなす.

4. この時点で同一メソッドの存在しなかった対象リビジョン中のメソッドは, このリビジョンにおいて削除されたメソッドと判定する. 同様にこの時点で同一メソッドの存

在しなかった次のリビジョン中のメソッドは, 次のリビジョンにおいて新しく追加されたメソッドと判定する.

このように全対象リビジョンにおいてメソッドの情報を追った結果, 同一メソッドと判断された各リビジョンにおけるメソッドの繋がりを**メソッドチェイン**と定義する. メソッドチェイン m は以下の情報を持つ.

- 存在期間 $P_L(m)$
- 重複コードを含んでいた期間 $P_C(m)$
- 重複コードを含んでいなかった期間 $P_N(m)$

期間とはリビジョンの集合で, $P_L(m) = P_C(m) \cup P_N(m)$ が成立している. この情報から, 以下に示す通り各メソッドチェイン m を, 重複コードを含んでいた期間と含んでいなかった期間を持つメソッドチェイン (*SC-Method*), 常に重複コードを含んでいたメソッドチェイン (*AC-Method*), 常に重複コードを含んでいなかったメソッドチェイン (*NC-Method*) の 3 種類に分類する.

$$
\begin{aligned}
m \in \textit{SC-Method} &\Leftrightarrow P_C(m) \neq \phi \wedge P_N(m) \neq \phi \\
m \in \textit{AC-Method} &\Leftrightarrow P_C(m) = P_L(m) \\
m \in \textit{NC-Method} &\Leftrightarrow P_N(m) = P_L(m)
\end{aligned}
$$

5.3.2 対象の絞り込み

3 種類の分類したメソッドチェインのうち, 計測の対象とするメソッドを絞り込む. *AC-Method*, *NC-Method* は存在期間中 1 度でも修正のあったメソッドチェインのみを対象とする. *SC-Method* は両期間中それぞれ 1 度以上修正があるメソッドチェインを対象とするが, 片方の期間が存在期間の 15 %に満たないようなメソッドチェインは不安定な期間を持つとみなし, 対象から除外している.

また, ソフトウェアの機能の追加などによってソースコードに大幅な変更が加えられる場合, 計測値が大きくなるが, これが保守へ影響を及ぼすとは考えにくい. そのため, この手法では修正の行われたメソッド数の割合の高い上位 2.5 %のリビジョンを対象リビジョンから除外している.

5.3.3 保守コスト

m をメソッドチェイン, P をリビジョンの集合が表すある期間, r をある計測対象リビジョンとし,

$$
\begin{aligned}
ChangedRevisions(m, P) &= \text{期間 } P \text{ 中に } m \text{ に対して変更が行われたリビジョンの集合} \\
Methods(r) &= r \text{ におけるメソッドの集合} \\
ChangedMethods(r) &= r \text{ で変更されたメソッドの集合} \\
CoChangedMethods(m, r) &= r \text{ で } m \text{ が変更されたとき, 共に変更されたメソッドの集合}
\end{aligned}
$$

と定義する. このとき次式で算出される $likelihood$, $impact$, $work$ を評価指標とし, 特に $work$ の値を**保守コスト**とする.

$$
likelihood(m, P) = \frac{|ChangedRevisions(m, P)|}{\sum_{r \in P} |ChangedMethods(r)|} \tag{9}
$$

$$
impact(m, P) = \frac{\sum_{r \in P} \frac{|CoChangedMethods(m, r)|}{|Methods(r)|}}{|ChangedRevisions(m, P)|} \tag{10}
$$

$$
work(m, P) = likelihood(m, P) \cdot impact(m, P) \tag{11}
$$

各指標における変数 m, P の組み合わせは以下の通りである.

1. $m \in SC\text{-}Method$ について, $P = P_C(m)$
2. $m \in SC\text{-}Method$ について, $P = P_N(m)$
3. $m \in AC\text{-}Method$ について, $P = P_L(m)$
4. $m \in NC\text{-}Method$ について, $P = P_L(m)$

上記の 1. と 2. の保守コストの分布の比較, 3. と 4. の保守コストの分布の比較を行う.

6 実験

本節では, 上述の調査手法, 重複コード検出ツールの組み合わせによって, 結果に違いが出るかどうか調査するために行った実験, 及びその結果について述べる.

6.1 実験手順

本実験の手順は以下の通りである.

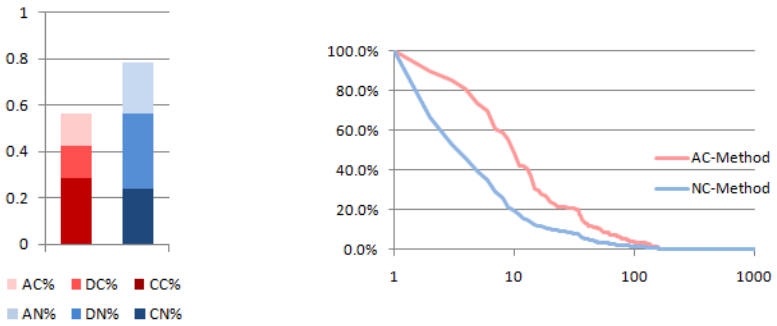
1. 計測対象リビジョンのソースファイルをリポジトリから取得.
2. ソースファイルのうちテストケース, 及び重複するクラスを含むファイルを削除.
3. *CommentRemover* を使用してソースコードを正規化.
4. 複数の検出ツールを用いてそれぞれ重複コードの位置を特定.
5. 計測対象リビジョンとその次のリビジョンのソースコード間の差分情報を, Unix の diff コマンドから取得.
6. 4. と 5. で得た内容, 更にメソッドの位置情報を照合し, 手法ごとに必要な値を計測.

2. について, 一般的にテストケースは多くの重複コードを含むため, より一般的な結論を導くために対象ファイルから除外している. また, メソッド情報取得のために用いた *MASU*[25] では, 完全限定名の一致するクラスがあればエラーを出力してしまう. これは 1 つのソフトウェア内で完全限定名の一致するクラスは存在するはずがないという前提で分析を行っているためである. しかしリポジトリから取得したソースコードでは, 開発の途中段階で変更前のソースコードを別ディレクトリに残し続ける場合がいくつか見受けられた. この場合そのソースコードには基本的に変更が加えられておらず, ソフトウェアを構成するものと考えられないため, このようなソースコードを対象から除外した.

6.2 計測結果の見方

佐野らの手法では, 重複コード, 非重複コードへの修正頻度を単純に棒グラフで表現しており, 値の大きい方が修正に影響を及ぼすコードであると判断している.

Krinke の手法では重複コード, 非重複コードに対して修正, 削除, 追加割合をそれぞれ積み上げ棒グラフで表示している. 測定値は%表記している. 削除と修正は同じリビジョンの行に対するもので, これらの行の集合は互いに素であることから, $DC + CC$ を重複コードの変更割合, $DN + CN$ を非重複コードの変更割合と捉えることができる. 図 6.1(a) に示す例では, ソースコードの追加, 削除についてはいずれも非重複コードの割合が高く, ソース



(a) Krinke の手法　　(b) Lozano らの手法

図 6.1: 計測結果のグラフの例

コードの修正については重複コードの割合が高い．しかし削除と修正をまとめて変更と捉えれば, 非重複コードの割合が高いといえる．

Lozano らの手法では, *SC-Method* の期間 P_C, P_N における保守コストの分布, また *AC-Method*, *NC-Method* の存在期間における保守コストの分布を比較する．ただしメソッドによって値に大きなばらつきがあるため, 累積度数で表している．x 軸は比較対象の値集合における最大値を 1000 等分したときの各値を表し, それ以上の値を持つメソッド数の割合をプロットしている．保守コストの最大値が平均値に比べて大きく外れているものが多く, 単純にプロットすると左端に値が偏ってしまうため, x 軸は対数軸としている．図 6.1(b) に示す例では, ある値以上の保守コストの値を持つメソッド数の割合が常に *AC-Method* の方が高いということから, 重複コードを含むメソッドの保守コストが高いといえる．

また, Lozano らの手法では比較する値の集合に対して検定を行い, 統計的に有意な差があるかどうかを判定している．*SC-Method* の, 期間 P_C, P_N における結果に対して Wilcoxon の符号順位和検定を, *AC-Method*, *NC-Method* の結果に対して Mann-Whitney の U 検定を適用した．これらの検定は今回のように結果の集合に外れ値を含む場合に有効な検定方法である．有意水準を 5%と定め, これらの検定の結果と累積度数分布の結果から, 次の手順で Lozano らの手法における結果を決定する．

1. *SC-Method* の期間別の比較, *AC-Method* と *NC-Method* の比較の両方に検定による有意な差があれば, *AC-Method* と *NC-Method* の比較の結果を手法における結果とする．
2. 上記のうち片方にのみ有意な差があれば, その結果を手法における結果とする．
3. 上記の両方に有意な差がなければ, Mann-Whitney の U 検定において *AC-Method* と *NC-Method* それぞれの値の集合で平均順位の大きい方を手法における結果とする．

1. について, *SC-Method* では同じメソッドの比較を行っており, 本質的に修正されやすいメソッドであれば重複コードの存在とは関係のないところで結果に影響を及ぼすことがある. Lozano らの手法では, 上記の方法で保守コストの比較を行い, それを裏付ける形で違う比較の手段を用いているが, 今回はその手段を用いず, *AC-Method* と *NC-Method* の比較の結果を優先する形で手法を再現した.

2. までの過程で結果の得られたものは, 有意差のある結果とする. 3. で得られた結果は有意差がない結果ではあるが, 今回は他の手法と結果の傾向の違いを比較するために, このような方法で結果を算出している.

なお, 佐野らの手法, Krinke の手法についても, 値に差があるかどうか判定を行う. 佐野らの手法では修正頻度の差が, Krinke の手法では変更割合の差が, 比較する値のうち大きい値の 5%以下であれば, 結果に差がないと判定する.

6.3 計測対象

本研究では *SourceForge*[26] で公開されているオープンソースソフトウェアの中から, 5 つのソフトウェアを対象に計測を行った. 対象としたソフトウェアを表 2 に示す. これらのソフトウェアを選択した基準は以下の通りである.

1. バージョン管理システム *Subversion* を用いて開発を行っている.

2. 開発に Java のみを用いている.

Lozano らの手法はメソッド単位での計測を行うなど仕様が Java の記述に依存しているため, 開発言語を Java に限定している.

Ant の計測対象リビジョンは, いずれの検出手法においてもバグの修正を含むリビジョンだけに絞っている. なお, バグ情報の抽出は 1 リビジョンごとに人手によってバグの修正か

表 2: 計測対象

プロジェクト名	リビジョン数		総行数	開発期間
	全	対象	(最新リビジョン)	
Ant	1,069,274	460	93,681	2001.09.21 -
OpenYMSG	194	192	6,023	2008.11.19 - 2010.12.06
EclEmma	1,220	490	13,698	2006.09.21 - 2010.12.29
MASU	1,620	1,450	37,375	2006.11.08 - 2010.12.25
TVBrowser	6,829	5,278	132,274	2003.04.22 - 2010.11.15

どうか判断して行っているため, 信頼性が高い. EclEmma は全リビジョンに対して計測対象リビジョンが少ないが, これはテストケースに対するコミットが多かったためである.

各手法, 検出ツールによる重複コード含有率を表 3 に示す. 手法ごとにおける重複コード含有率の定義は次の通りである. なお, MASU, TVBrowser に対する *Scorpio* での計測は, 実行に大きな時間を要したため処理を中断した.

佐野らの手法 全対象リビジョンの重複コード行数/全対象リビジョンの総行数

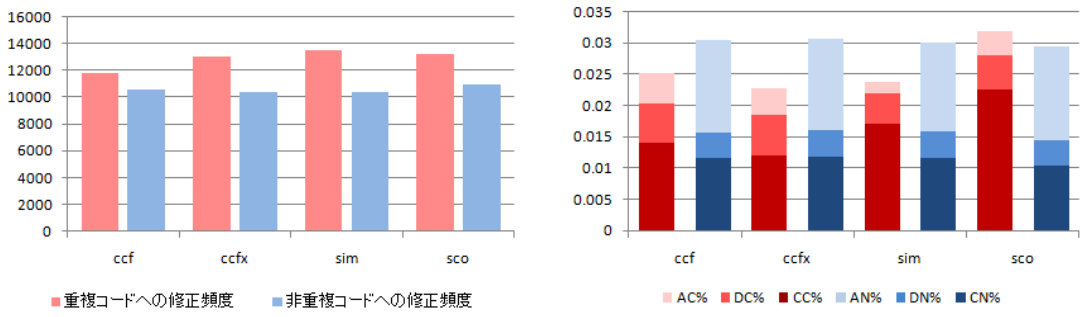
Krinke の手法 重複コード行数/総行数 の平均値

Lozano らの手法 *SC-Method* と *AC-Method* の数/対象リビジョン中のメソッド数

また, 次の節での計測結果を示した図において, 検出ツール名を表 4 のように略記する.

表 3: 重複コード含有率

プロジェクト名	手法	重複コード含有率 (%)				
		CCFinder	*CCFinderX*	*Simian*	*Scorpio*	平均
Ant	佐野	10.85	9.56	4.87	13.00	9.57
	Krinke	12.92	12.17	5.83	12.92	10.96
	Lozano	12.53	18.38	12.46	12.33	13.93
OpenYMSG	佐野	12.30	6.14	2.62	5.53	6.65
	Krinke	11.78	6.39	2.35	9.13	7.41
	Lozano	7.24	8.50	9.02	8.18	8.24
EclEmma	佐野	6.97	4.76	2.02	3.71	4.36
	Krinke	7.53	4.63	1.92	3.84	4.48
	Lozano	5.72	10.61	8.79	3.87	7.25
MASU	佐野	25.62	26.49	11.31	—	21.14
	Krinke	24.30	26.29	11.04	—	20.54
	Lozano	17.85	38.45	23.50	—	26.60
TVBrowser	佐野	13.64	10.86	5.39	—	9.96
	Krinke	12.30	9.78	4.22	—	8.77
	Lozano	24.09	21.78	14.34	—	20.07



(a) 佐野らの手法　　(b) Krinke の手法

図 6.2: 計測結果 —Ant—

6.4 バグ修正のみを用いた場合の計測結果

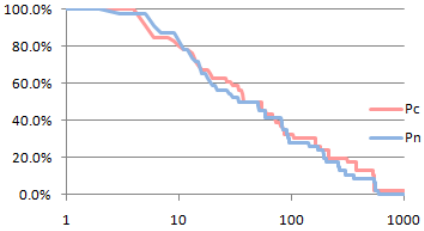
本節ではバグに対する修正のあったリビジョンのみを対象にした, Ant に対して計測を行った結果を述べる.

佐野, Krinke, Lozano の各手法での計測結果を, 図 6.2, 図 6.3 に示す. 佐野らの手法では, いずれの検出ツールにおいても重複コードの修正頻度の方が高いという結果になった. Krinke の手法では, コードの削除, 修正は重複コードに対する割合が高く, コードの追加は非重複コードへの割合が高いという結果になった.

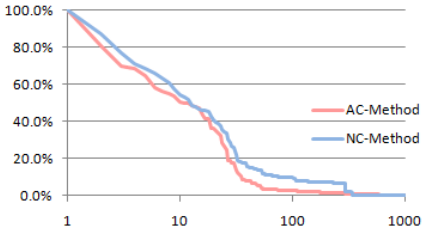
Lozano らの手法において, *CCFinder*, *Scorpio* を用いた検出で *NC-Method* の保守コストが高いという結果になった. 他の 2 つのツールでは結果に有意差はなかったが, *SC-Method* は *Simian* を用いた検出で期間 P_C の保守コストが高いという結果になった. *CCFinderX* ではどちらも有意な差がなかったが, *AC-Method* と *NC-Method* の値の平均順位を比較すると, *NC-Method* の方が大きかった.

表 4: 検出ツール名の略記

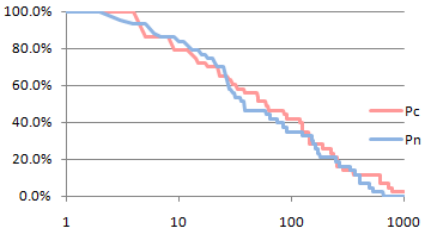
検出ツール名	略記
CCFinder	**ccf**
CCFinderX	**ccfx**
Simian	**sim**
Scorpio	**sco**



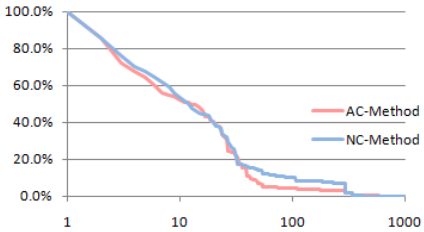
(a) *CCFinder* (SC)



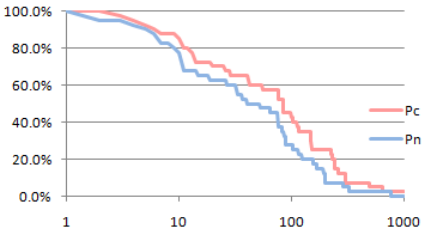
(e) *CCFinder* (AC vs NC)



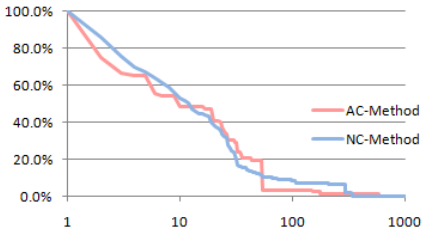
(b) *CCFinderX* (SC)



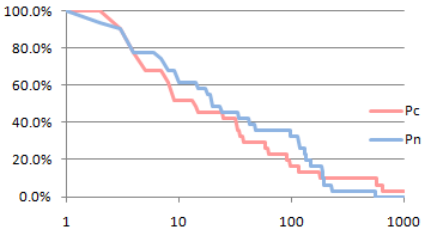
(f) *CCFinderX* (AC vs NC)



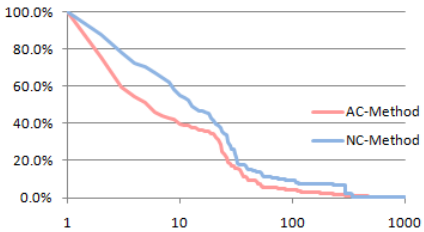
(c) *Simian* (SC)



(g) *Simian* (AC vs NC)

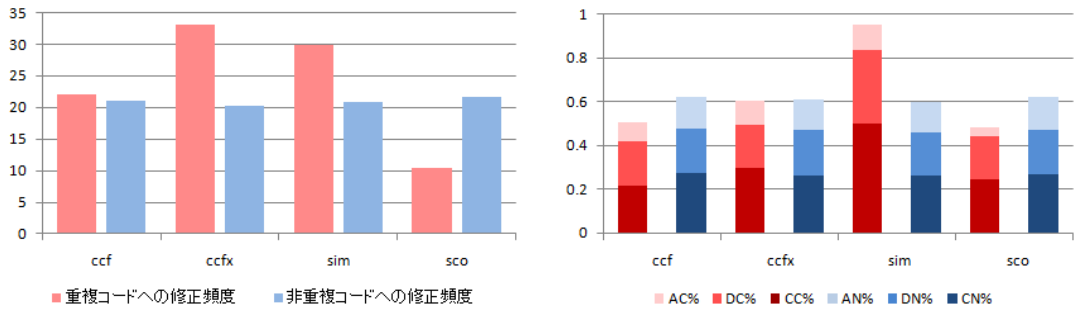


(d) *Scorpio* (SC)



(h) *Scorpio* (AC vs NC)

図 6.3: 計測結果 (Lozano らの手法) —Ant—



(a) 佐野らの手法　　(b) Krinke の手法

図 6.4: 計測結果 —OpenYMSG—

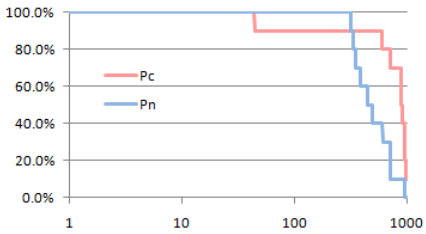
6.5 全ての修正を用いた場合の計測結果

本節では, Ant 以外のソフトウェアに対して計測を行った結果を述べる.

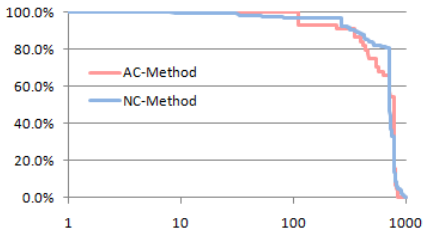
6.5.1 OpenYMSG

OpenYMSG を対象とした結果を図 6.4, 図 6.5 に示す. OpenYMSG は開発期間が短くリビジョン数も小さいため, Krinke の手法の際, 計測対象リビジョンは佐野らの手法における計測対象リビジョンを用いている. 佐野らの手法では *Scorpio* を用いた検出結果だけが非重複コードに対する修正頻度が高く, それ以外では重複コードに対する修正頻度が高いという結果になった. Krinke の手法では, コードの追加はいずれの検出ツールでも非重複コードへの割合が高いが, コードの削除, 修正割合は *CCFinder*, *Scorpio* を用いた検出で非重複コードの割合が高く, *CCFinderX*, *Simian* を用いた検出で重複コードの割合が高いという結果になった.

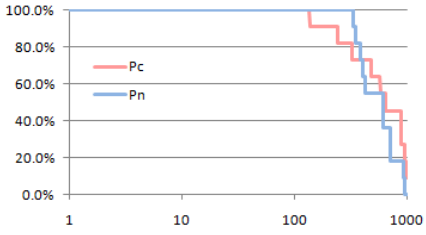
Lozano らの手法において, *AC-Method* と *NC-Method* の比較では, *CCFinderX*, *Scorpio* を用いた検出では *AC-Method* の保守コストが, *Simian* を用いた検出では *NC-Method* の保守コストが高かった. *CCFinder* を用いた検出ではどちらも有意な差がなかったが, *AC-Method* の平均順位が *NC-Method* の平均順位より高かった.



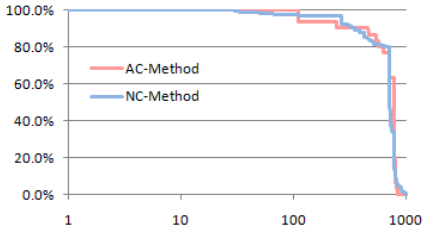
(a) *CCFinder* (SC)



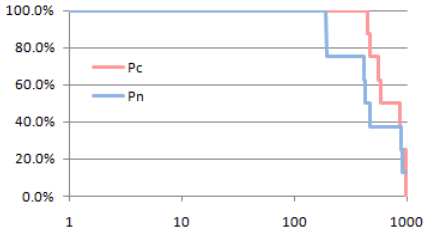
(e) *CCFinder* (AC vs NC)



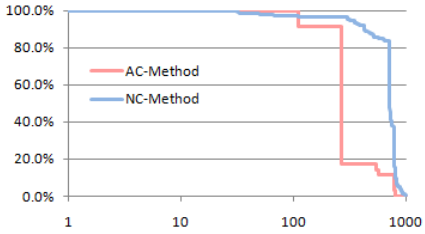
(b) *CCFinderX* (SC)



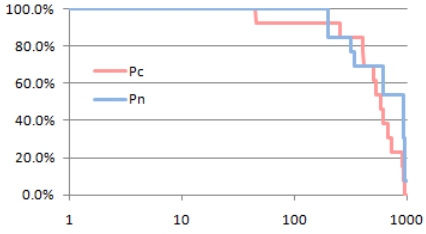
(f) *CCFinderX* (AC vs NC)



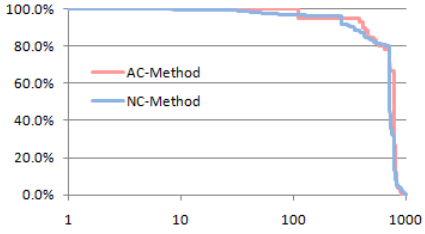
(c) *Simian* (SC)



(g) *Simian* (AC vs NC)

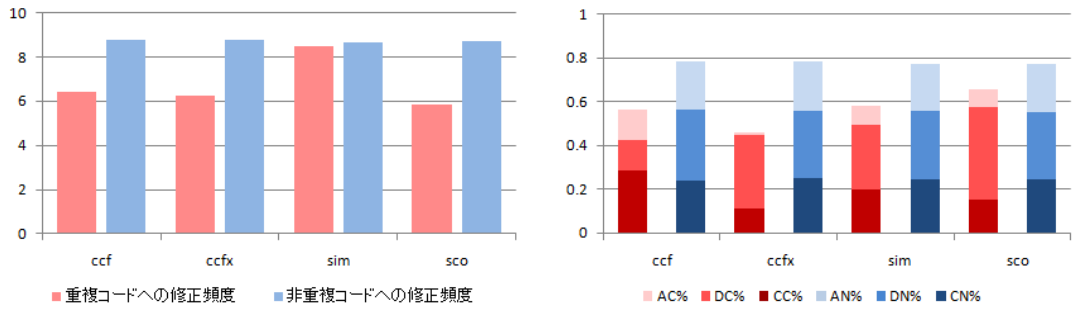


(d) *Scorpio* (SC)



(h) *Scorpio* (AC vs NC)

図 6.5: 計測結果 (Lozano らの手法) —OpenYMSG—



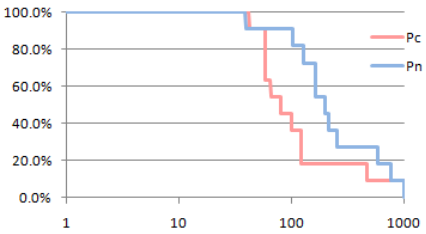
(a) 佐野らの手法　　(b) Krinke の手法

図 6.6: 計測結果 —EclEmma—

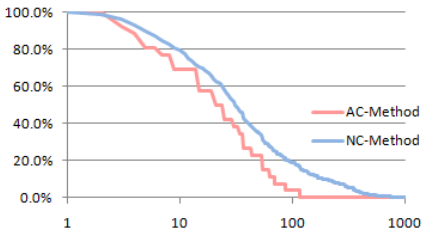
6.5.2 EclEmma

EclEmma を対象とした結果を図 6.6, 図 6.7 に示す. 佐野らの手法ではいずれの検出ツールでも非重複コードに対する修正頻度が高いという結果になった. ただし, *Simian* を用いた場合の計測値の差は 5%以下であった. Krinke の手法では, コードの追加は非重複コードへの割合が大きいという結果になった. コードの削除, 修正は, それぞれについては検出ツールによって結果が異なるものの, 合計値では, *Scorpio* では重複コードの割合が大きく, それ以外では非重複コードの割合が大きいという結果になった. ただし, *Scorpio* を用いた場合の計測値の差は 5%以下であった.

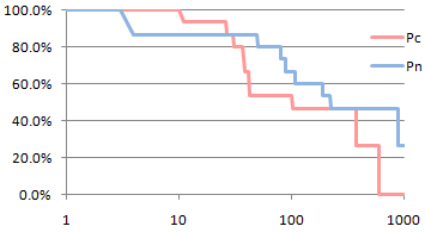
Lozano らの手法において, *AC-Method* と *NC-Method* の比較では, いずれの検出ツールでも有意差はなかった. *SC-Method* は *CCFinderX* を用いた検出で期間 P_N の保守コストが高いという結果になった. また, 他の検出ツールでは有意差はなかったので, *AC-Method* と *NC-Method* の値の平均順位を比較した結果, *CCFinder*, *Scorpio* では *NC-Method* の, *Simian* では *AC-Method* の保守コストが高いと判断した.



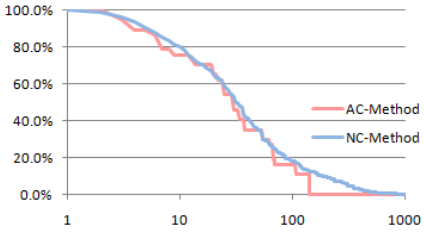
(a) *CCFinder* (SC)



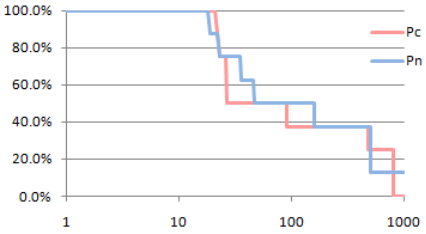
(e) *CCFinder* (AC vs NC)



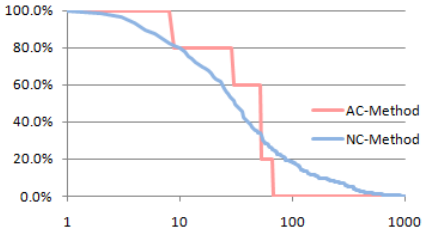
(b) *CCFinderX* (SC)



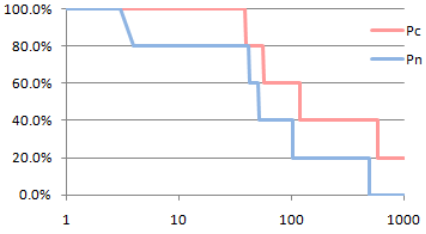
(f) *CCFinderX* (AC vs NC)



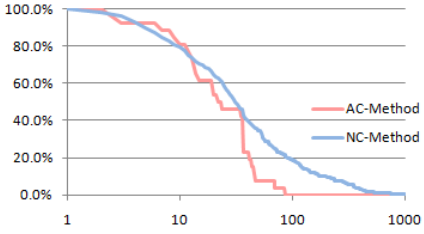
(c) *Simian* (SC)



(g) *Simian* (AC vs NC)

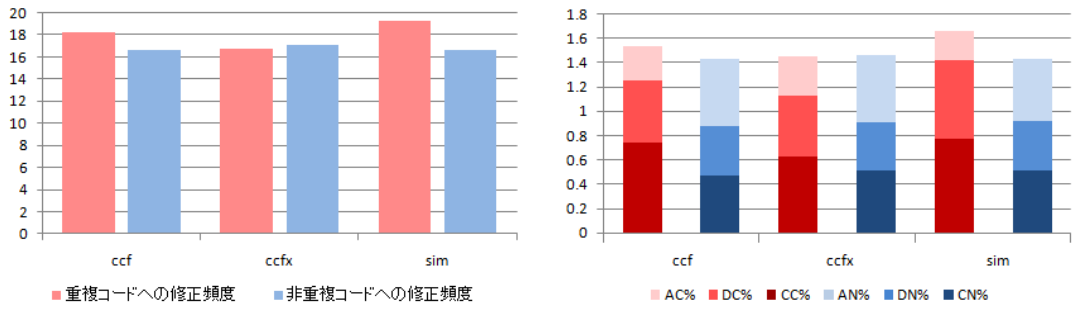


(d) *Scorpio* (SC)



(h) *Scorpio* (AC vs NC)

図 6.7: 計測結果 (Lozano らの手法) —EclEmma—



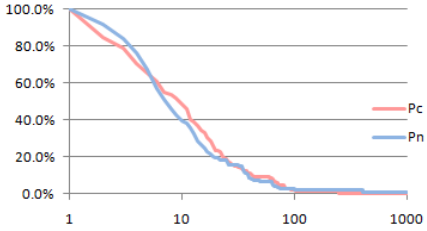
(a) 佐野らの手法　　(b) Krinke の手法

図 6.8: 計測結果 —MASU—

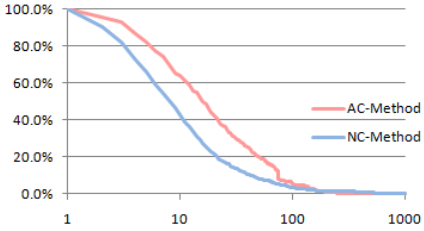
6.5.3 MASU

MASU を対象とした結果を図 6.8, 図 6.9 に示す. 佐野らの手法では *CCFinderX* における検出結果は非重複コードの修正頻度が高く, それ以外は重複コードの修正頻度が高いという結果になった. ただし, *CCFinderX* を用いた場合の計測値の差は 5%以下であった. Krinke の手法では, いずれの検出ツールでもコードの追加は非重複コードへの割合が大きく, コードの削除, 修正は重複コードの割合が大きいという結果になった.

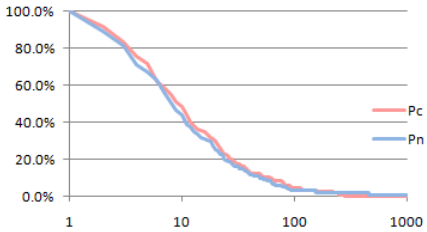
Lozano らの手法において, *SC-Method* はいずれの検出ツールでも有意差がなかった. *AC-Method* と *NC-Method* の比較では, いずれの検出ツールでも *AC-Method* の保守コストが高いという結果になった.



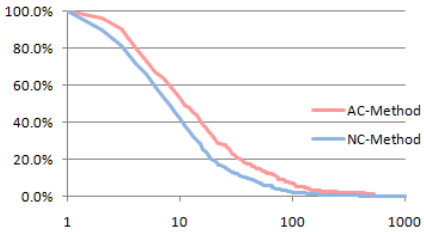
(a) *CCFinder* (SC)



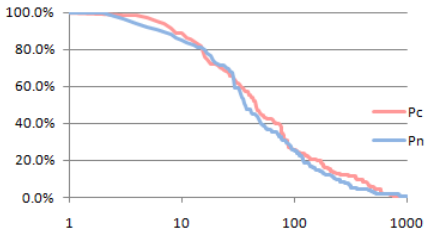
(d) *CCFinder* (AC vs NC)



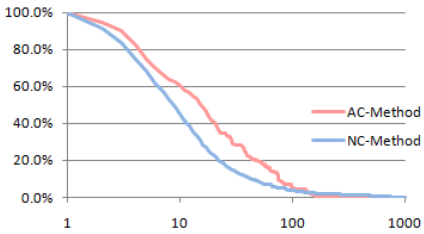
(b) *CCFinderX* (SC)



(e) *CCFinderX* (AC vs NC)

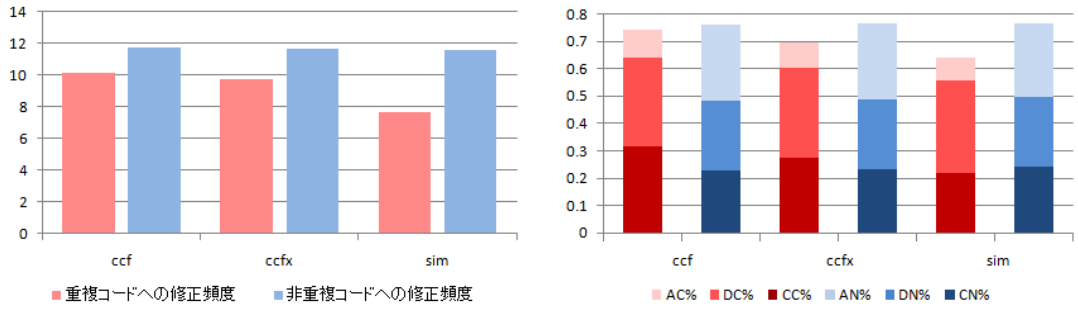


(c) *Simian* (SC)



(f) *Simian* (AC vs NC)

図 6.9: 計測結果 (Lozano らの手法) —Masu—



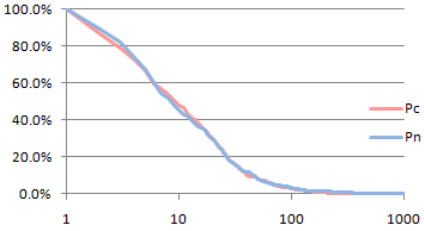
(a) 佐野らの手法　　(b) Krinke の手法

図 6.10: 計測結果 —TVBrowser—

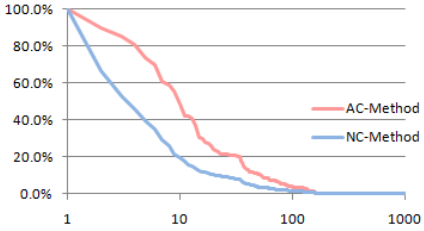
6.5.4 TVBrowser

TVBrowser を対象とした結果を図 6.10, 図 6.11 に示す. 佐野らの手法ではいずれの手法でも非重複コードの修正頻度が高いという結果になった. Krinke の手法ではコードの追加は非重複コードへの割合が大きく, コードの修正, 削除は重複コードの割合が高いという結果になった.

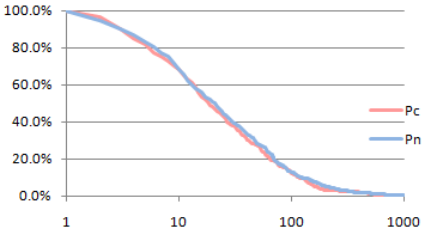
Lozano らの手法において, *SC-Method* はいずれの検出ツールでも有意差がなかった. *AC-Method* と *NC-Method* の比較はサンプル数が多く統計テストができなかったが, グラフの形状, また値集合の平均値, 中央値から, いずれの検出ツールでも *NC-Method* の保守コストが高いと判断した.



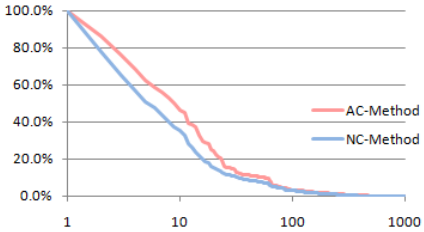
(a) *CCFinder* (SC)



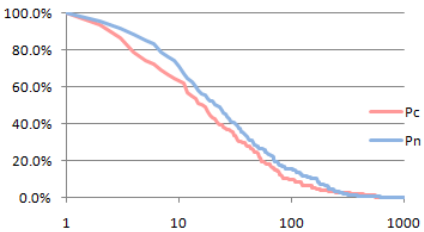
(d) *CCFinder* (AC vs NC)



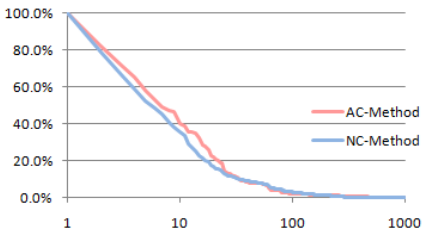
(b) *CCFinderX* (SC)



(e) *CCFinderX* (AC vs NC)



(c) *Simian* (SC)



(f) *Simian* (AC vs NC)

図 6.11: 計測結果 (Lozano らの手法) —TVBrowser—

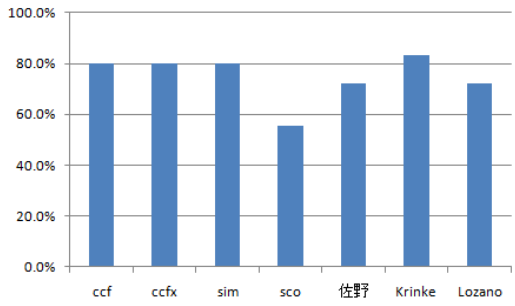


図 7.1: ソフトウェアの結果と一致する割合

7 考察

7.1 ソフトウェアごとの結果の傾向と違い

本実験より, 各ソフトウェアに対して得られた全ての組み合わせの結果を表 5 に示す. ここでは, 重複コードに対する修正が大きいという結果を C, 非重複コードに対する修正が大きいという結果を N と表し, 結果に差がないものは*を付加している. また, Ant, OpenYMSG, EclEmma については, 3 つの手法と 4 つの重複コード検出ツールの組み合わせ 12 通りによるそれぞれの結果を, MASU, TVBrowser については, 3 つの手法と 3 つの重複コード検出ツールの組み合わせ 9 通りによるそれぞれの結果を比較し, そのソフトウェアに対して重複コードと非重複コードのどちらが修正に影響を及ぼしているか判断した. その結果を表 6 に示す. ここでは過半数を占める結果をそのソフトウェアの結果とし, その結果が得られた組み合わせの割合を表示している.

検出ツールごと, 手法ごとに, ソフトウェアの結果と一致する結果が得られた割合を図 7.1 に示す. ツールを比較すると, *Scorpio* を用いた検出では約半数が異なった結果となっているが, *CCFinder*, *CCFinderX*, *Simian* ではいずれも 80%が一致する結果を得られている. また, 手法を比較すると, Krinke の手法が他の手法に比べて同じ結果となった割合がやや高いことが分かる.

また, 計測できた全てのソフトウェアについて, この結果と同じ結果を得られた組み合わせは以下の 2 つであった.

- *CCFinderX* を用いた Krinke の手法
- *Simian* を用いた Krinke の手法

以上より, Krinke の手法では他の手法と比べて比較的安定した結果を得られると考えられ, あるソフトウェアについて結果を得たいときにこの手法の適用は最適である可能性がある.

表 5: 各組み合わせにおける結果

プロジェクト名	手法	結果			
		CCFinder	*CCFinderX*	*Simian*	*Scorpio*
Ant	佐野	C	C	C	C
	Krinke	C	C	C	C
	Lozano	N	N	C*	N
OpenYMSG	佐野	C	C	C	N
	Krinke	N	C	C	N
	Lozano	C*	C	N	C
EclEmma	佐野	N	N	N*	N
	Krinke	N	N	N	C*
	Lozano	N*	N	C*	N*
MASU	佐野	C	N*	C	—
	Krinke	C	C	C	—
	Lozano	C	C	C	—
TVBrowser	佐野	N	N	N	—
	Krinke	C	C	C	—
	Lozano	C	C	C	—

表 6: ソフトウェアの結果

ソフトウェア	結果	割合
Ant	重複コード	75%
OpenYMSG	重複コード	67%
EclEmma	非重複コード	83%
MASU	重複コード	89%
TVBrowser	重複コード	67%

7.2 異なる結果を出した原因

次にソフトウェアごとに, 異なる結果を出した組み合わせについて調査した.

7.2.1 Ant

Lozano らの手法では, 有意差のない結果も含めるといずれの検出ツールを用いた場合でも, *AC-Method* に比べて *NC-Method* の保守コストが高いという結果となった. これはメソッドの変更割合を示す評価指標 *likelihood* が, どの検出ツールにおいても *AC-Method* の方が高いためであった. また, *NC-Method* の中には存在期間の短いメソッドチェインが多く, これらの大多数が計測対象リビジョンのうち時期の早いリビジョンで存在していたが, このようなメソッドチェインの *likelihood* の値が特に高いと分かった. この原因は, 時期の早いリビジョンではメソッド数が少ないため, 第 5 節で示した式 (9) の分母にあたる, 変更のあったメソッド数の合計が少なかったと考えられる.

また, Ant に対してはバグの修正だけを用いているため, 計測対象リビジョンの間隔が開いていることが多い. Lozano らの手法では, リビジョン間でメソッドの同一性を判定するため, リビジョンの間隔が広いほどメソッドチェインが適切に生成できていない可能性がある.

このために Ant に対する Lozano らの手法は条件が適切ではなく, 他の手法と異なった結果が得られたと考えられる.

7.2.2 OpenYMSG

佐野らの手法では *Scorpio* を用いた場合のみ非重複コードの修正頻度が高いという結果となった. 計測対象リビジョン数, 各リビジョンのソースコードの行数はソフトウェアごとに一意なので, 検出ツールによって修正頻度の値に差の出る箇所は「総修正箇所数」「重複コードを含む総行数」の 2 点である. これらの値を重複コード検出ツールごとに表 7 に示す. この結果によると *Scorpio* での検出結果は重複コードの行数に比べて修正箇所数が少ないことが分かる. 更にソースコードを調べると, *Scorpio* を用いた場合のみ重複コードとして検出していないコードがいくつか存在した. その例を図 7.2 に示す. ここに示す 2 つのメソッドは同一クラスに存在し, あるリビジョンで$で示した行に対して修正が行われた. このメソッドは完全一致したコードが多く, 明らかに重複コードとして検出されるべき箇所である. しかし *Scorpio* を用いた検出では重複コードと判断されなかった. これは, このリビジョンにおいてこのクラスが構文エラーを含んでおり, プログラムの構造から重複コードを検出する *Scorpio* では, 構文の正しくないクラスにおいて重複コードの検出を行えなかったことが原因であった.

Krinke の手法では *Simian* を用いた検出結果で重複コードに対する修正割合が他と比べて

```
protected void receiveConfLogoff(YMSG9Packet pkt) {
$       YahooConference yc = getOrCreateConference(pkt);
        synchronized (yc) {
           if (!yc.isInvited()) {
              yc.addPacket(pkt);
           return;}}
        try {
$          SessionConferenceEvent se = new SessionConferenceEvent(
$               this, pkt.getValue("1"),
$               pkt.getValue("56"),
$               null,
$               yc
           );
           yc.removeUser(se.getFrom());
           if (!yc.isClosed())
              eventDispatchQueue.append(se, ServiceType.CONFLOGOFF);}
        catch (Exception e) {
           throw new YMSG9BadFormatException("conference logoff", pkt, e);}}
```

(a)

```
protected void receiveConfLogon(YMSG9Packet pkt) {
$       YahooConference yc = getOrCreateConference(pkt);
        synchronized (yc) {
           if (!yc.isInvited()) {
                yc.addPacket(pkt);
           return;}}
        try {
$          SessionConferenceEvent se = new SessionConferenceEvent(
$               this, pkt.getValue("1"),
$               pkt.getValue("53"),
$               null,
$               yc
           );
           yc.addUser(se.getFrom());
           if (!yc.isClosed())
                eventDispatchQueue.append(se, ServiceType.CONFLOGON);}
        catch (Exception e) {
           throw new YMSG9BadFormatException("conference logon", pkt, e);}}
```

(b)

図 7.2: *Scorpio* 以外のツールを用いたとき検出される重複コードの例

表 7: 修正箇所数の比較 —OpenYMSG—

検出ツール名	重複コード総行数	重複コードへの修正箇所数
CCFinder	77,463	299
CCFinderX	38,691	224
Simian	16,516	86
Scorpio	34,811	63

```
public interface SessionListener{
        public void newMailReceived(SessionNewMailEvent ev);
        public void notifyRecveived(SessionNotifyEvent ev);
        public void contactRequestReceived(SessionEvent ev);
        .
        .
}
```

(a) SessionListener.java

```
public class SessionAdapter implements SessionListener {
        public void newMailReceived(SessionNewMailEvent ev) {
        }
        public void notifyRecveived(SessionNotifyEvent ev) {
        }
        public void contactRequestReceived(SessionEvent ev) {
        }
        .
        .
}
```

(b) SessionAdapter.java

図 7.3: *Simian* でのみ検出される重複コードの例

高かった．この原因について調べたところ, 図 7.3 に示すように, Listener インターフェースのメソッド宣言と, それを実装する Adapter クラスにおける中身のないメソッド宣言が同じ順序であったため, 行単位で重複コードを検出する *Simian* でのみ, 重複コードとして検出されていた．また, あるリビジョンでこれらのメソッドの引数名が全て変更されるという修正が行われていた．しかし, Listener インターフェースに対する Adapter クラスはプログラムの記述を簡潔に表すために用いられるもので, 作業量に影響を与えるとは考えにくい．

Lozano らの手法でも *Simian* での検出結果が他の検出ツールを用いた場合と比べて値の分布が大きくずれているが, これも図 7.3 に示す重複コードが原因である．上記 2 つのファイルのメソッドが, *Simian* を用いた調査では *AC-Method*, 他のツールを用いた調査では *NC-Method* と判定されている．*Simian* での検出では重複コード含有率が少ないため, *AC-Method* の数も少ないが, 重複コードとして検出された箇所が複数のメソッドであったため, これらのメソッドのチェインが *AC-Method* の 60%以上を占めてしまった．これらメソッドチェインの, 変更割合の指標 *likelihood* はいずれも高いが, ともに変更されたメソッドの割合を示す指標 *impact* の値が他の *AC-Method* に含まれるメソッドチェインより低かったため, 保守コストの値が低く, 全体的に *AC-Method* の保守コストを下げる結果となっている．

7.2.3 EclEmma

EclEmma は 5 つの対象ソフトウェアの中で唯一非重複コードの修正が大きいという結果となっているが, *Scorpio* を用いた Krinke の手法, *Simian* を用いた Lozano らの手法でこれとは異なった結果が出ている．この原因について調査した．

Krinke の手法では, 図 6.6(b) より *Scorpio* を用いた検出で重複コードの削除とみなした割合が他の検出ツールを用いた場合と比べてやや高くなっている．そこで各計測対象リビジョンについて, 削除された重複コードの割合, 非重複コードの割合を調べた．*Scorpio* を用いた検出におけるこの割合の推移を図 7.4 に示す．削除された重複コードの割合を正の値, 削除された非重複コードの割合を負の値とし, 各リビジョンにおける値を棒グラフで表し, 更にその和を折れ線グラフで表示している．

この図によると, 後半のあるリビジョンで重複コードの削除割合が高いという結果になっており, この結果は他の検出ツールを用いた場合は見られなかった．ソースコードの修正履歴を調べると, *Scorpio* でのみ重複コードとみなされるコードを多く含むファイルが, このリビジョンで削除されたためであった．このコードの例を図 7.5 に示す．図中の%及び%%で示した箇所と, #及び##で示した箇所が互いに重複関係にあたり, 7.5(a) のファイルが削除されていた．

このようなウィジェットの記述は意味的に重複した記述が繰り返されたもので, 今回用いた重複コード検出ツールの中では *Scorpio* だけが検出できる重複コードの典型的な例である．図

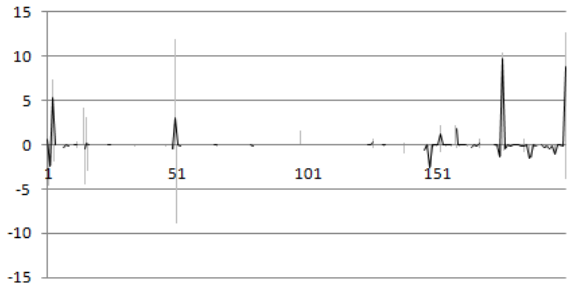


図 7.4: コードの削除割合の推移 —EclEmma, *Scorpio* を用いて—

7.5(a) の%%で示した箇所, ##で示した箇所は複数行にわたってはいるが, *Scorpio* 実行の際構築されるプログラム依存グラフではそれぞれ 1 つの要素を表しているため, widgetSelected メソッドのオーバーライドを行っている箇所の記述は, 直接的には重複コードとは関係がない. しかし Krinke の手法では, この箇所に含まれる全ての行を重複コードとみなしているため, 計測結果に差が出たと考えられる. このように, *Scorpio* を用いた場合, 重複コードが大きな単位で検出されることがあり, 行単位で計測を行う Krinke の手法との組み合わせは不適切である可能性が高い.

次に, Lozano らの手法適用の際, EclEmma に対して検出ツールごとで得られたメソッドチェイン数を表 8 に示す. これによると, 特に *Simian* を用いた検出では *AC-Method* に含まれるメソッドチェイン数が少ない. これはソフトウェアの規模が大きくないことと, 重複コードの含有率が低いことが原因である. メソッドチェイン数が余りに少ない場合グラフから傾向の判断がしづらく, また統計的にも差がないと出ている結果がほとんどであった. 今回この組み合わせで重複コードへの修正が高いと判断した理由は, *SC-Method* の期間 P_C における保守コストが P_N に比べてわずかに高かったためであり, これを実際の結果と結び付けるのは強引であると考えられる.

表 8: メソッドチェイン数 —EclEmma—

分類	検出ツール			
	CCFinder	*CCFinderX*	*Simian*	*Scorpio*
SC-Method	11	15	8	5
AC-Method	26	37	5	26
NC-Method	985	948	982	1,001

```
%       btnDefault = new Button(c, SWT.CHECK);
%       btnDefault.setText("Default:");
%%      btnDefault.addSelectionListener(new SelectionAdapter() {
%%              @Override
%%              public void widgetSelected(SelectionEvent e) {
%%                      ...
%%              }
%%      });
        ...
%       btnAdd = new Button(buttonBar, SWT.PUSH);
%       btnAdd.setText("Add");
%       btnAdd.addSelectionListener(new SelectionAdapter(){
%               ...
%       });
%       btnRemove = new Button(buttonBar, SWT.PUSH);
%       btnRemove.setText("Remove");
%       btnRemove.addSelectionListener(new SelectionAdapter(){
%               ...
%       });
```

(a) WildcardBlock.java

```
#       binariescheck = new Button(parent, SWT.CHECK);
#       binariescheck.setText(UIMessages.ImportSessionPage1Binaries_label);
##      binariescheck.addSelectionListener(new SelectionAdapter() {
##              @Override
##              public void widgetSelected(SelectionEvent e) {
##                      ...
##              }
##      });
        ...
#       Button buttonSelectAll = new Button(parent, SWT.PUSH);
#       buttonSelectAll.setText(UIMessages.SelectAllAction_label);
#       buttonSelectAll.addSelectionListener(new SelectionAdapter() {
#               ...
#       });
        ...
#       Button buttonDeselectAll = new Button(parent, SWT.PUSH);
#       buttonDeselectAll.setText(UIMessages.DeselectAllAction_label);
#       buttonDeselectAll.addSelectionListener(new SelectionAdapter()
#               ...
#       });
```

(b) SessionImportPage1.java

図 7.5: *Scorpio* でのみ検出される重複コードの例

7.2.4 MASU

CCFinderX を用いた佐野らの手法を適用した場合のみ, 他の検出ツール, 手法を用いた場合と異なる結果が出ている. これについて調査した.

重複コードへの修正箇所数の割合, 重複コードの行数の割合, 重複コードへの修正頻度を表 9 に示す. この結果によると, *CCFinderX* を用いた検出では, 修正行数の割合と修正箇所数の割合が同程度だが, 他のツールを用いた検出では修正箇所数の割合がわずかに高い. このために *CCFinderX* を用いた検出で, 重複コードへの修正頻度が低めとなっている. 次に計測対象リビジョンを 10 の期間に分割して, それぞれの期間における修正頻度を計測した. その結果を図 7.6 に示す. なお, この図では *CCFinder* を *C*, *CCFinderX* を *X*, *Simian* を *Si* と略している. この図によると, 2, 3, 4, 8, 10 番目の期間では全ての検出ツールにおいて重複コードへの修正頻度が高いことが分かる. また, 重複コードへの修正頻度が *CCFinderX* を用いた検出時のみ特別低いということもなく, この組み合わせは他の組み合わせの傾向に沿っていると言える. 今回は重複コードと非重複コードの修正頻度に大きな差がなかったため, 特別な要因は無いが結果に違いが出てしまった. このことから, 既存の手法のように組み合わせを限定せず, 複数の組み合わせから判断することも重要であると言える.

表 9: 重複コードに対する修正箇所数, 修正行数の割合 —MASU—

検出ツール名	修正箇所数の割合	行数の割合	修正頻度
CCFinder	27.99%	25.62%	18.22
CCFinderX	26.75%	26.49%	16.76
Simian	13.08%	11.31%	19.29

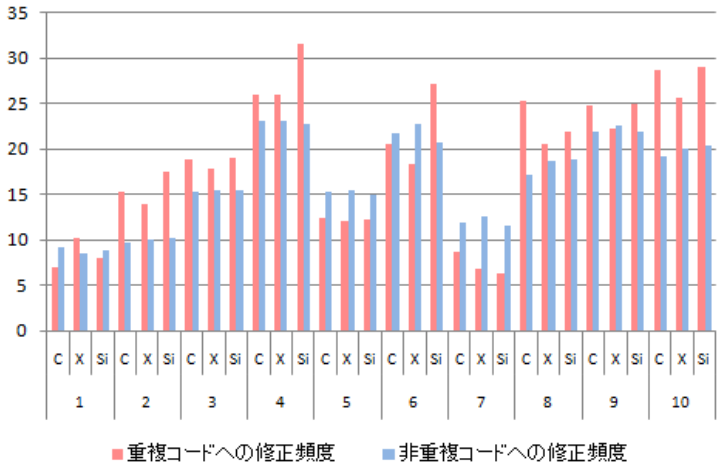


図 7.6: 修正頻度の推移 —MASU—

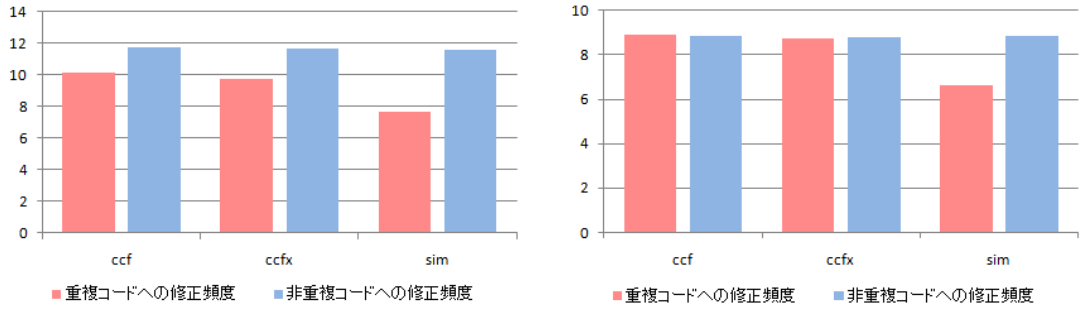
7.2.5 TVBrowser

TVBrowser に対する計測結果は, 各手法におけるツール間の違いはほとんど見られなかった. しかし佐野らの手法では非重複コードに対して, Krinke, Lozano らの手法では重複コードに対して修正が多いという結果となっている. この原因について調査した.

Krinke の手法では追加については無視して削除と修正行数の割合からのみ判断しているのに対して, 佐野らの手法では追加を含めた修正箇所数を計測している. このために違いが出たのではないかと考え, 佐野らの手法でも削除と修正箇所のみを対象として修正頻度を計測しなおした. その結果を図 7.7 に示す. この図より, ソースコードの削除と修正に関しては, *CCFinder* と *CCFinderX* を用いた検出では重複コードへの頻度, 非重複コードへの頻度に大きな差がなく, ソースコードの追加箇所が非重複コードに対する場合が多かったことが分かった. しかし, ソフトウェア開発に伴いソースコードは必ず追加されていくため, ソースコードの重複コード含有率が高くない限り必然的に非重複コードへの追加が多いと考えられる. ソースコードの修正内容にコードの追加も含めるならば, その内容も考慮するべきである.

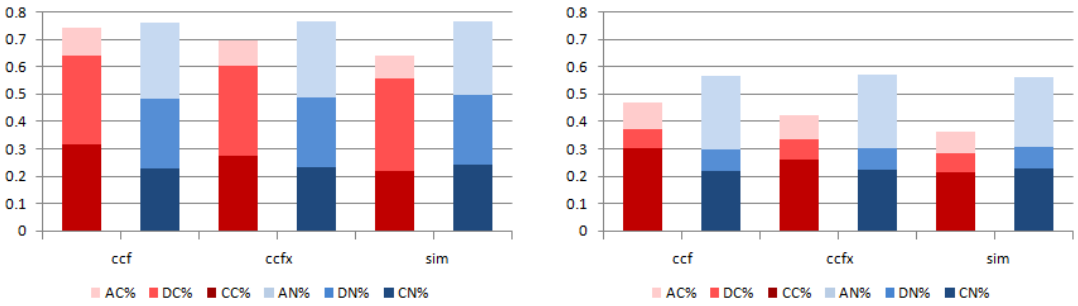
さらに, 佐野らの手法ではファイルの削除は 1 箇所の修正とみなし, ファイル中に重複コードがどれだけ多く含まれていても修正頻度に影響は出ない. 一方, Krinke の手法ではファイルの削除はそのファイルに含まれる全てのコードの削除とみなすため, 重複コードが多く含まれるファイルほど削除された場合に重複コードの修正量が多いという結果となるが, 実際にファイルの削除に伴う修正量がソースコードの行数に依存するとは考えにくい. そこで, Krinke の手法において, ファイルの削除を無視して修正行の割合を計測しなおした. その結果を図 7.8 に示す. この図より, ファイルの削除を除いた場合の削除されたソースコードは, 重複コードの割合, 非重複の割合に大きな差がないことが分かった. 修正も含めた変更割合を比較すると, *Simian* では非重複コードの変更割合がわずかに高いという結果が得られた.

このように, 修正量を表す値に大きく影響するような修正の内容が手法によって異なるため, 同じ重複コード検出ツールを用いた場合でも, 適用する手法で結果が異なる可能性がある.



(a) コードの追加を含む修正頻度　　(b) コードの追加を除く修正頻度

図 7.7: 佐野らの手法における計測結果 —TVBrowser—



(a) ファイルの削除を含む場合　　(b) ファイルの削除を除く場合

図 7.8: Krinke の手法における計測結果 —TVBrowser—

8　結果の妥当性

バグ修正の利用と Lozano らの手法

Ant に対してバグの修正だけを用いた場合の計測対象リビジョンは, 他の対象ソフトウェアと比較してリビジョンの間隔が大きく, Lozano らの手法ではメソッドチェインが適切に生成できていない可能性がある. メソッドチェインの定義は Lozano らの提案手法に沿ったものだが, リビジョンの間隔の大きいソフトウェアを対象にしていないため, メソッドチェイン生成に同じ手段を取ることに疑問が残る. 計測対象リビジョンをバグの修正の行われたリビジョンに限定する場合でも, ソースコードに修正の加えられた全てのリビジョンからメソッドチェインを定義する必要があると考えられる.

対象ソフトウェアの規模

本研究で実験を行った環境では, 大規模なソフトウェアに対して *Scorpio* を用いた重複コードの検出に大きな時間を要したため結果を出すことができなかった. そのため, 大規模なソフトウェアに対する *Scorpio* を用いた実験を行うと, 本研究で得られた結果とは違った結果が得られる可能性がある. また, 規模の小さいソフトウェアは定義されたメソッド数が少ないため, Lozano らの手法を適用した際に保守コストの値の分布が極端なものとなる傾向があり, 適切な比較が行えていない恐れがある.

計測対象

本研究で対象としたソフトウェアは 5 つであり, Java で開発されたオープンソースのソフトウェアに限定して調査を行った. このため, より多くのソフトウェアを対象として実験を行った場合や, 異なる言語で記述されたソフトウェアや商用ソフトウェアを対象とした場合は, 本研究で得られた結果と異なる結果が得られる可能性がある.

9 あとがき

本研究では, 重複コードがソースコードの修正に与える影響について, 計測の単位や評価指標, 重複コード検出ツールの違いによって結果にどのような違いが出るのか調査した. また, バグの修正のみを対象としたソフトウェアと全ての修正を対象とした複数のソフトウェアに対して実験を行った. その結果, バグの修正のみを対象としたソフトウェアについては, 重複コードに対する修正が多いという傾向が得られた. また, いずれのソフトウェアに対しても, 調査手法と重複コード検出ツールの全ての組み合わせで同じ結果を得ることはできなかったが, いくつかの組み合わせは常にソフトウェアの結果と一致する結果を得られた. 更に, 同じ調査手法を適用しても, 重複コードの検出単位が異なる場合に結果に差が出やすいという傾向が得られた.

今後の課題は以下の通りである.

- より多くのソフトウェアに対して実験を行う.
- 今回用いた重複コード検出ツール以外も比較対象に含める.
- 同一のソフトウェアに対して, バグ修正のみを対象とした場合と全ての修正を対象とした場合の比較を行う.
- 実際の保守コストと比較する.

謝辞

本研究を行うにあたり, 理解あるご指導を賜り, 常に励まして頂きました 楠本 真二 教授に心から感謝申し上げます.

本研究に関して, 有益かつ的確なご助言を頂きました 岡野 浩三 准教授に深く感謝申し上げます.

本研究の全過程を通し, 終始熱心かつ丁寧なご指導を頂きました 肥後 芳樹 助教に深く感謝申し上げます.

本報告を行うにあたり, 多大なるご助言, ご助力を頂きました, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 2 年の 西野 稔 氏に深く感謝申し上げます.

本研究に用いたツールの実装の大部分を担当してくださり, 多大なるご助言, ご助力を頂きました, 大阪大学大学院情報科学研究科コンピュータサイエンス専攻博士前期課程 1 年の 堀田 圭佑 氏に深く感謝申し上げます.

その他の楠本研究室の皆様のご助言, ご協力に心より感謝致します.

また, 本研究に至るまでに, 講義, 演習, 実験等でお世話になりました大阪大学基礎工学部情報科学科の諸先生方に, この場を借りて心から御礼申し上げます.

参考文献

[1] 肥後芳樹, 楠本真二, 井上克郎, コードクローン検出とその関連技術, 電子情報通信学会論文誌, Vol. J91–D, No. 6, pp. 1465–1481, 2008

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

[3] CCFinderX, http://www.ccfinder.net/ccfinderx-j.html

[4] Simian - Similarity Analyser, http://www.redhillconsulting.com.au/products/simian/

[5] 肥後芳樹, 楠本真二, コードクローン検出に必要な計算コストの削減を目的としたプログラム依存グラフ頂点集約手法の提案, ソフトウェアエンジニアリング最前線 2010(ソフトウェアエンジニアリングシンポジウム 2010 予稿集), pp. 127–134, Sep. 2010

[6] J.H. Johnson, Substring matching for clone detection tools, Proc. International Conference on Software Maintenance 94, pp. 120–126, Sep. 1994

[7] 佐野由希子, 肥後芳樹, 楠本真二, 重複コードと非重複コードにおける修正頻度の比較, 電子情報通信学会技術研究報告, Vol. 109, No. 456, pp. 43–48, Mar. 2010

[8] J. Krinke, Is Cloned Code more stable than Non-Cloned Code?, Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008

[9] A. Lozano and M. Wermelinger, Asessing the effect of clones on changeability, International Conference on Software Maintenance, pp. 227–236, 2008

[10] 斎藤晃, 吉田則裕, 松下誠, 井上克郎, 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, 電子情報通信学会技術研究報告, Vol. 110, No. 227, pp. 19–24, 2010

[11] Subversion, http://subversion.apache.org/

[12] ソースコード正規化ツール CommentRemover, http://www-sdl.ist.osaka-u.ac.jp/˜higo/cgi-bin/moin.cgi/commentremover

[13] 門田暁人, 佐藤慎一, 神谷年洋, 松本健一, コードクローンに基づくレガシーソフトウェアの品質の分析, 情報処理学会論文誌, Vol. 44, No. 8, pp. 2178–2188, 2003

[14] S. G. Eick and T. L. Graves and A. F. Karr and J.S. Marron and A. Mockus, Does code decay? assessing the evidence from change management data, IEEE TRANSCATIONS ON SOFTWARE ENGINEERING, Vol. 27, No. 1, pp. 1–12, Jan. 2001

[15] N. Göde, Evolution of Type-1 Clones, Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, pp. 77–86, 2009

[16] A. Lozano and M. Wermelinger and B. Nuseibeh, Evaluating the harmfulness of cloning: a change based experiment, IEEE Fourth International Workshop on Mining Software Repositories, 2007

[17] Clone Tracker, http://mcs.open.ac.uk/alr242/CloneTracker.htm

[18] 井上克郎 , 神谷年洋, 楠本真二, コードクローン検出法, コンピュータソフトウェア, Vol. 18, No. 5, pp. 47–54, 2001

[19] S. Bellon, Detection of software clones, Technical Report, Institute for Software Technology, University of Stuttugart, 2003. available at http://www.bauhaus-stuttgart.de/clones/

[20] S. Bellon and R. Koschke and G. Antniol and J. Krinke and E. Merlo, Comparison and evaluation of clone detection tools, IEEE Trans. Software Engineering, Vol. 31, No. 10, pp. 804–818, Oct. 2007

[21] I. Baxter and A. Yahin and L. Moura, M. Anna and L. Bier, Clone Detection Using Abstract Syntax Trees, Proc. of International Conference on Software Maintenance 98, pp.368–377, Mar. 1998

[22] S. Uchida and A. Monden and N. Ohsugi and T. Kamiya and K. Matsumoto and H. Kudo, Software Analysis by Code Clones in Open Source Software, Journal of Computer Information Systems, Vol. XLV, No. 3, pp. 1–11, Apr. 2005

[23] 肥後芳樹, 楠本真二, 実規模ソフトウェアへの適用を目的としたプログラム依存グラフに基づくコードクローン検出法, ソフトウェアエンジニアリング最前線 2009, Sep. 2009

[24] StrikeAMatch, htttp://www.catalysoft.com/articles/StrikeAMatch.html

[25] 三宅達也, 肥後芳樹, 楠本真二, 井上克郎, 多言対応メトリクス計測プラグイン開発基盤 MASU の開発, 電子情報通信学会論文誌 D, Vol. J92–D, No. 9, pp. 1518–1531, Sep. 2009

[26] SourceForge.net : Find and Develop Open Source Software, http://sourceforge.net/