修士学位論文

題目

関数クローン変更管理システムの開発と評価

指導教員
井上 克郎 教授

報告者
佐野 真夢

平成 28 年 2 月 9 日

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 ソフトウェア工学講座

平成 27 年度 修士学位論文

関数クローン変更管理システムの開発と評価

佐野 真夢

内容梗概

ソフトウェア保守を困難にする大きな要因の 1 つとしてコードクローンが指摘されている. コードクローンとは, ソースコード中に存在する互いに一致または類似した部分を持つコード片のことであり, 既存コードのコピーアンドペーストによる再利用等が原因で生じる. また, コードクローンは, コーディングスタイルを除いて構文上完全に一致するものや, 構文上は異なる実装を行っているが同様の処理を実行しているものなど, いくつかの種類に分類することができる.

あるコード片にバグが存在することが判明した場合, そのコードクローンにおいても同様のバグが存在している可能性が考えられる. 従って, 1 つのコード片のバグを修正する際には, それに関連する全てのコードクローンを調査し, 必要に応じて一貫した修正を行う必要があると考えられる. また, バグ修正以外にも, 同じコード片の最適化や共通の仕様変更といった修正に対しても, 一貫した修正が必要な可能性があると考えられる. ゆえに, コード片の修正を行う際には, それに対するコードクローンを調査することが望ましいと考えられるが, そのような調査を修正の度に手動で実施することは非効率的であると考えられる.

この問題を解決するために, 修正が行われたコード片と構文上一致するコードクローンを自動的に検出し, それを開発者に通知する機能を提供するコードクローン変更管理システムが開発されている. このシステムを利用すれば, 構文上一致するコードクローンの調査の手間を省くことができると考えられる.

しかし, 一貫した修正の対象となるコードクローンが必ずしも構文上一致しているとは限らない. 例えば, 仕様変更による修正の場合, 構文上は異なる実装を行っているが同様の処理を実行しているコードクローンも一貫した修正の対象になりうると考えられる. また, 完全に一致しておらず, ごく一部分が異なるコード片であっても, 修正が必要な部分が一致していれば一貫した修正の対象となりうる. 前述のコードクローン変更管理システムでは, このようなコードクローンを検出することはできない.

また, 前述のコードクローン変更管理システムは検出粒度がコード片単位であり, 詳細にコードクローンの調査を行うことができるが, その分, 一貫した修正とは無関係な修正事例も数多く検出してしまうという問題も存在する.

そこで, 本研究では, 一部の差異を持つコードクローンや, 構文上は異なる実装を行っているコードクローンも検出できる新たなコードクローン変更管理システムの開発を行った. このような種類のコードクローンを検出するために, 本研究では, 情報検索技術に基づいた既存のコードクローン検出ツールを利用した. このツールは関数単位でコードクローンを検出するため, より機能的に類似し, 一貫した修正が必要になる可能性の高いクローンを検出できるようになることが期待できる. また, 開発したシステムを評価するために, 実際にどの程度のコードクローンが検出可能であるかの調査と, 従来の構文上一致するコードクローンの変更管理システムとの比較評価を行った. 前者の調査では, オープンソースソフトウェアの過去の開発履歴を対象に, 本システムが実際に検出したコードクローン修正事例を調査することで, 従来の変更管理システムでは検出できない多くのコードクローン修正事例を検出していることを確認した. また, 後者の比較評価では, 従来システムと開発したシステムのそれぞれをオープンソースソフトウェアの過去の開発履歴に適用し, 一貫した修正が必要かどうかと, クローン追跡ができているかという 2 つの視点を尺度に, 検出したクローンセット修正事例からランダムに選択した各システム 50 件のクローンセットにおける有用なクローンセットの割合を比較することで, 開発したシステムが従来システムより有用であることを確認した.

主な用語

コードクローン
ソフトウェア保守
情報検索技術

目 次

1 まえがき

ソフトウェア保守工程における大きな問題の 1 つとしてコードクローンが指摘されている [1, 3, 8, 9, 14]. コードクローンとは, ソースコード中に存在する互いに一致または類似した部分を持つコード片のことであり, 既存コードのコピーアンドペーストによる再利用や, 定型処理の実装, 最適化を目的とした意図的なコードの繰り返しなど様々な要因により生じる [3, 6]. また, コードクローンは, コーディングスタイルを除いて構文上完全に一致するものや, 構文上は異なる実装を行っているが同様の処理を実行しているものなど, いくつかの種類に分類することができる [18].

一般的に, あるコード片にバグが存在することが判明した場合, そのコードクローンにおいても同様のバグが存在している可能性が考えられる. 従って, 1 つのコード片のバグを修正する際には, それに関連する全てのコードクローンを調査し, 必要に応じて一貫した修正を行う必要がある [13, 14]. また, バグ修正以外にも, 同じコード片の最適化や共通の仕様変更といった修正に対しても, 一貫した修正が必要な可能性があると考えられる. ゆえに, コード片の修正を行う際には, それに対するコードクローンを調査することが望ましいと考えられるが, そのような調査を修正の度に手動で実施することは非効率的であると考えられる.

この問題を解決するために, CloneNotifier[20] と呼ばれるシステムが開発された. CloneNotifier は, 修正が行われたコード片とコードクローンを自動的に検出し, それを開発者に通知する機能を提供するコードクローン変更管理システムである. このシステムを利用すれば, 開発者は通知されたコードクローン情報に基づいて一貫した修正の検討を行えるため, 構文上一致するコードクローンの調査の手間を省くことが可能である. また, システムが自動的にコードクローンを検出するため, 一貫した修正が必要なコードクローンを見逃す可能性が大幅に減ると考えられる.

CloneNotifier は検出部に CCFinder[9] と呼ばれるコードクローン検出ツールを利用している. CCFinder は, 字句解析ベースのコードクローン検出ツールの 1 つであり, コーディングスタイルを除いて構文上一致するコードクローンに加えて, 変数などの識別子の名前が異なるコードクローンも検出することができる.

しかし, 一貫した修正の対象となるコードクローンが必ずしも構文上一致しているとは限らない. 例えば, 仕様変更による修正の場合, 構文上は異なる実装を行っているが同様の処理を実行しているコードクローンも一貫した修正の対象になりうると考えられる. また, 完全に一致しておらず, 演算子やごく一部の文が異なるコード片であっても, 修正が必要な部分が一致していれば一貫した修正の対象となりうる. CCFinder は, このような種類のコードクローンの検出には対応していないため, CloneNotifier でもこれらを検出することはできない. もしこれらを検出できれば, 一貫した修正を見逃す可能性のさらなる削減が期待できる.

また, 前述のコードクローン変更管理システムは検出粒度がコード片単位であり, 詳細にコードクローンの調査を行うことができるが, その分, 一貫した修正とは無関係な修正事例も数多く検出してしまうという問題も存在する. 関数単位など, より大きな粒度のコードクローンであれば, 機能的に類似し, 一貫した修正が必要になる可能性がより高くなると考えられる. また, 粒度が大きくなれば検出数自体は減少するため, より一貫した修正が必要になる可能性が高いコードクローンを厳選して検出できることが期待できる.

そこで, 本研究では, 構文上一致しているコードクローンだけでなく, 一部の差異を持つコードクローンや, 構文上は異なる実装を行っているコードクローンも検出できる新たなコードクローン変更管理システムの開発を行った. このような種類のコードクローンを検出するために, 本研究では, 山中らのツール [21] と呼ばれる情報検索技術に基づくコードクローン検出ツールを利用した. 山中らのツールは関数単位でコードクローン検出を行っており, 上述の粒度の問題も解決することが期待できる. また, 開発したシステムを評価するために, 実際にどの程度のコードクローンが検出可能であるかの調査と, 従来の構文上一致するコードクローンの変更管理システムとの比較評価を行った. 前者の調査では, オープンソースソフトウェアの過去の開発履歴を対象に本システムを適用し, 実際に検出されたコードクローン修正事例を調査することで, 従来の変更管理システムでは検出できない多くのコードクローン修正事例を検出していることを確認した. また, 後者の比較評価では, CloneNotifier と開発したシステムのそれぞれをオープンソースソフトウェアの過去の開発履歴に適用し, 一貫した修正が必要かどうかと, クローン追跡ができているかという 2 つの視点を尺度に, 検出したクローンセット修正事例からランダムに選択した各システム 50 件のクローンセットにおける有用なクローンセットの割合を比較することで, 開発したシステムが CloneNotifier より有用であることを確認した.

以降, 2 節では本研究に関連する研究について説明する. また, 3 節では本研究で開発したコードクローン変更管理システムについて説明し, 4 節では本研究で行った評価実験について述べる. そして, 5 節で本研究のまとめと今後の課題について述べる.

2　背景

本節では, 本研究の背景として, コードクローンとその検出ツール, コードクローン変更管理システムについて説明する.

2.1　コードクローン

コードクローン (Code clone) とは, ソースコード中に存在する互いに一致または類似した部分を持つコード片のことである. また, 互いにコードクローンとなるコード片の対はクローンペア (Clone pair), クローンペアにおいて推移関係が成り立つコードクローンの集合はクローンセット (Clone set) と呼ばれている.

一般的に, コードクローンの存在はソフトウェア保守を困難にすると考えられている [6]. 例えば, あるコード片にバグが存在することが判明した場合, そのコード片に対するコードクローンにも同様のバグが含まれている可能性が考えられる. そのため, バグを修正する際には, 修正するコード片に対する全てのコードクローンを対象に, 一貫した修正が必要か否かを検討する必要がある. また, バグ修正だけでなく, 同様のコードに対する最適化や共通の仕様変更など, コードクローンに対して一貫した修正を検討すべき様々な変更が考えられる. そのため, コードクローンの存在を知ることは, ソフトウェア保守において重要である. しかし, 特にソフトウェアの規模が大きい場合, 開発者がソフトウェア中に含まれる全てのコードクローンを見つけることは非現実的である. 従って, 一般的に, ソフトウェア開発では自動検出ツールを用いてコードクローンの検出を行う [9].

コードクローンに対する保守作業としては, 集約を行うことも考えられる. 集約とは, 同じクローンセットに含まれるコードクローンを 1 つのメソッドやクラスなどにまとめることであり, 集約を行うことで, 保守の対象となるコードクローンを除去することできる. しかし, 集約が必ずしも実施可能とは限らない. 最適化やプロジェクトにおけるコーディング規約などの理由で意図的にコードクローンとして残している場合や, いくつかの差異を持つコードクローン（2.1.2 節を参照）である場合, 集約を実施することは困難であると考えられる. このようなコードクローンは, 集約を行わずにコードクローンのまま管理し, 必要に応じて一貫した修正を行うことで保守していく必要がある. このような保守作業を効率良く実施するためには, コードクローンを自動的に管理し, 一貫した修正の実施を手助けするコードクローン変更管理システムの存在が重要であると考えられる.

2.1.1　コードクローンの発生原因

コードクローンがソースコード中に発生する原因には様々なものが考えられる. 以下に, 代表的なコードクローンの発生原因を挙げる [4, 9].

既存コードのコピーアンドペーストによる再利用

一からソースコードを書くよりも, 同様の処理を行う既存コードを流用し, 必要に応じて部分的な変更を加えた方が信頼性は高い. そのため, 実際には, コピーアンドペーストによる既存コードの再利用が数多く存在する.

定型処理

定義上簡単で, 頻繁に用いられる処理はコードクローンになる傾向がある. 例として, キューの挿入処理や, データ構造へのアクセス処理などが挙げられる.

プログラミング言語における適切な機能の欠如

プログラミング言語が抽象データ型やローカル変数に対応していない状況において, 同じようなアルゴリズムを持つ処理を繰り返し書かなければならない場合がある.

パフォーマンスの改善

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, 特定のコードを意図的に繰り返し記述することでパフォーマンスの改善を図ることがある.

コード生成ツールによる自動生成

コード生成ツールは, あらかじめ定められたコードをベースにして自動的にコードを生成する. そのため, 目的の処理が類似している場合, 識別子などを除いて類似したコード片が生成される.

複数のプラットフォームへの対応

複数の OS や CPU に対応しているソフトウェアでは, 各プラットフォーム用のコード中に重複した処理が存在する傾向がある.

偶然の一致

偶然, 開発者が同一のコードを書いてしまう場合がある.

2.1.2 コードクローンの定義

コードクローンには様々な検出方法が存在するが, そのどれもが異なったコードクローンの定義を持つ. そのため, 厳密で普遍的なコードクローンの定義は存在しない. Roy らは, コードクローンを以下のような 4 つのタイプに分類している [18].

タイプ 1

空白やコメントの有無, インデントなどのコーディングスタイルの違いを除いて完全に一致するコードクローン.

タイプ 2

タイプ 1 の違いに加えて, 変数名や関数名などのユーザ定義名, 及び変数の型などの一部の予約語が異なるコードクローン.

タイプ 3

タイプ 2 の違いに加えて, 文の挿入や削除, 変更が行われているコードクローン.

タイプ 4

類似する処理を実行するが, 文の並び替えなど構文上の実装が異なるコードクローン.

本研究では, 上記に挙げた Roy らによる分類に基づいて, コードクローンを定義している. 以降, 本稿では, コードクローンのタイプは Roy らによる分類に従ったものとして説明を行う.

2.2 コードクローン検出ツール

現在までに, コードクローンを検出するための様々な手法が考案され, それを実装した検出ツールも開発されている. 例えば, ソースコードの字句解析に基づく手法 [2, 9, 14, 17] では, ソースコード中にある同一の文字列を検索することでコードクローンの検出を行う. また, 特徴メトリクスに基づく手法 [12, 15] では, クラスや関数, ファイルといったプログラム中のある種の単位ごとに特徴メトリクスを定義・算出し, それらのメトリクス値が類似したものをコードクローンとして抽出する. その他にも, 多くの検出手法が存在している [5, 7, 8, 11].

本節では, 本研究に関連する 2 つのコードクローン検出ツールと, その検出手法について紹介する.

2.2.1 CCFinder

CCFinder[9] は, 構文の類似性に着目したコードクローン検出ツールの 1 つであり, ソースコードの字句解析に基づく手法を用いてコードクローン検出を行っている. 具体的には, 字句解析によりソースコードをトークン列に変換し, 閾値以上の長さを持つ同一のトークン列をコードクローンとして検出している. また, 変数名や関数名などのユーザ定義名を同一のトークンに置き換えることで, ユーザ定義名の違いを吸収することを可能としている. 従って, タイプ 1,2 のコードクローンを検出することができる.

CCFinder は C/C++, Java, COBOL などの多くのプログラミング言語に対応している. また, 高いスケーラビリティも有しており, 大規模なソフトウェアに対しても実用的な時間でコードクローンを検出することできる. 実際に, CCFinder は様々な大規模ソフトウェアに適用され, その有用性が確認されている [16].

2.2.2 山中らのツール

山中らのツール [21] は, 意味的な類似性に着目したコードクローン検出ツールの 1 つであり, 情報検索技術を用いることで関数単位のコードクローン（以下, 関数クローンと呼ぶ）の検出を行う. 簡単に言うと, 関数中で用いられているユーザ定義名に基づいて, 関数同士の類似度を求めることでコードクローンを検出している. すなわち, 出現するユーザ定義名が似ていれば, 関数クローンとして検出される.

山中らのツールにおける, 具体的なコードクローン検出手順は, 以下の通りである.

1. ソースコードから, 各関数ごとにワード（変数名や関数名などのユーザ定義名, 予約語）を抽出する.

2. 手順 1. で抽出した各ワードに対して, その出現頻度やプロジェクト全体に対する非一般性（すなわち, そのワードが全ての関数で汎用的に使用されるものかどうかを表す尺度）を元に重み付けを行う. また, それに基づいて, 各関数の特徴ベクトルを算出する.

3. 手順 2. で求めた特徴ベクトルに基づいて, 各関数をクラスタリングする.

4. 手順 3. で求めた関数の各クラスタ内において, 特徴ベクトルの類似度を計算し, 閾値以上の類似度を持つものをコードクローンとして検出する.

上記の手法を用いることで, 山中らのツールはタイプ 1 から 4 の全てのコードクローンを検出することが可能である. また, 全てのタイプのコードクローンに対応した他の検出ツールよりも, 高速にコードクローンを検出することができる.

2.3 クローン変更管理システム CloneNotifier

2.1 節で説明したように, あるコード片にバグが存在することが判明した場合, そのコード片に対するコードクローンにも同様のバグが存在する可能性が考えられる. また, 性能改善や仕様変更といった他の修正においても, 同様の修正を複数のコードクローンに対して適用しなければならない可能性がある. そのため, 修正を行う際には, 修正が加えられるコード片に対する全てのコードクローンを調査し, 必要に応じて一貫した修正を行わなければならない. このような一貫した修正は, どのタイプのコードクローンでも必要になる可能性があると考えられる. しかし, 修正するコード片に対するコードクローンを修正の度に検出し, 一貫した修正の必要性があるか検討することは非効率的であると考えられる.

上記の問題を解決するために, CloneNotifier[20] というシステムが利用できる. CloneNotifier とは, コードクローンの変更を管理するシステムであり, 修正されたコード片がコードクローンを持つ場合, そのクローンセットを一貫した修正を検討すべきものとして開発者に

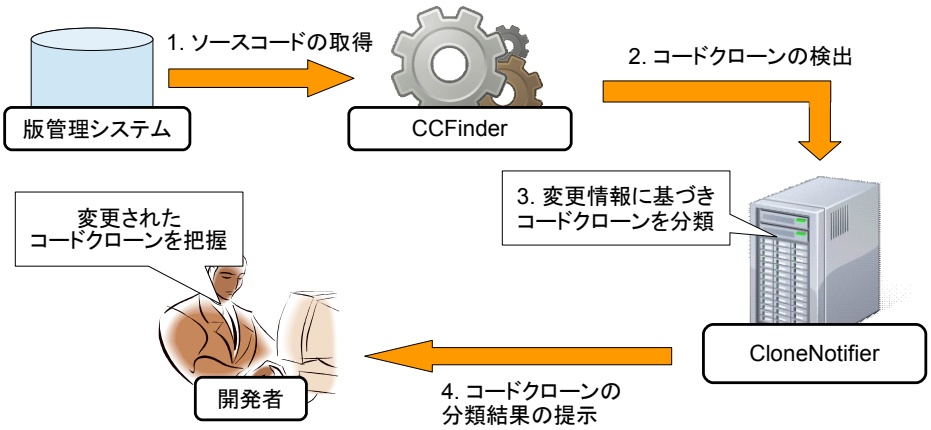


図 1: CloneNotifer の通知プロセス

自動で通知する機能を持つ. コードクローンの検出には, 2.2.1 節で説明した CCFinder が利用されている. 従って, CloneNotifier はタイプ 1, 2 のコードクローンの変更を管理できるシステムといえる.

図 1 は, CloneNotifier において, 修正されたコードクローンを開発者に通知するまでのプロセスを示した図である. 通知プロセスは, 下記のような手順で通知が行われる.

1. 版管理システムから最新のソースコードを取得する. この際, 以前のバージョンのソースコードを移動して保持する.

2. 取得した最新のソースコードと, 以前のバージョンのソースコードからコードクローンの検出を行う.

3. コードクローンの検出結果と, 変更情報に基づいてコードクローンを分類する. 分類は修正されたか否かの他, 新たにコードクローンとして追加された, コードクローンではなくなった, 異なるコード片とクローンセットを構成するようになった場合も考慮される.

4. コードクローンの分類結果を開発者に通知する. 通知方法は電子メール（テキスト）, CSV, html 形式の 3 種類が存在する.

また, 特定の 2 バージョンを指定した比較も可能であり, 過去の開発履歴から修正されたクローンセットを検出することも可能である. また, 単に修正されたクローンセットだけで

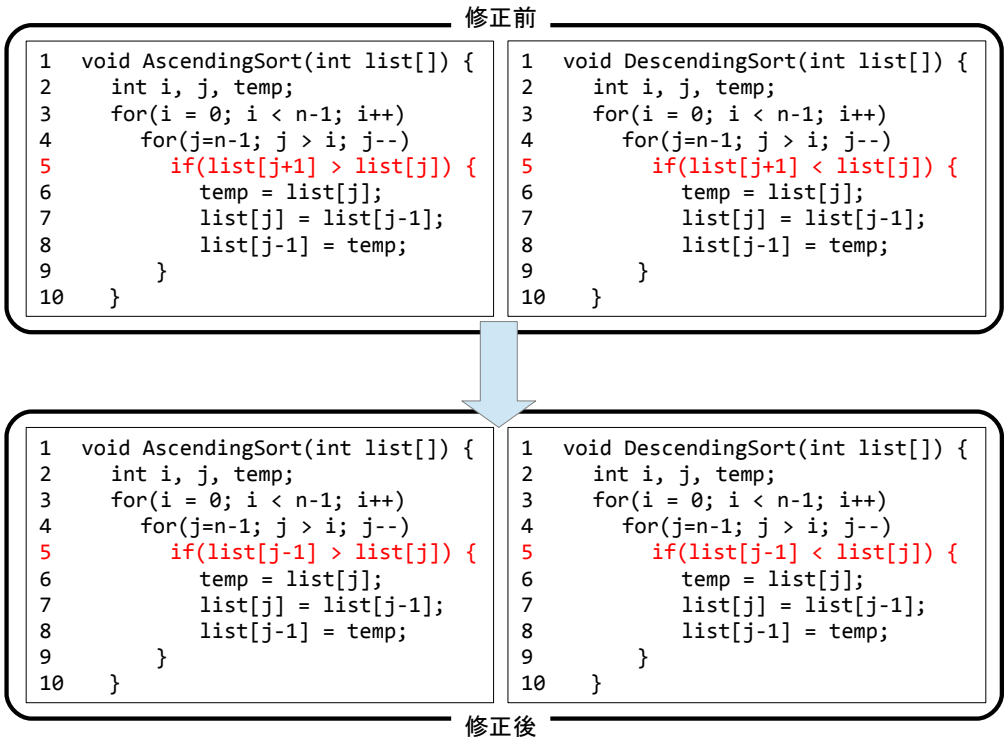


図 2: タイプ 3 コードクローンの一貫した修正の例

なく, 新たに増えたクローンセットや, 削除されたクローンセットも検出できる. 検出されるクローンセットの分類は以下の通りである.

New そのバージョン間において, 新たに出現したクローンセット.

Changed そのバージョン間において, 修正が行われたクローンセット, および, そのクローンセットを構成するコードクローンが削除・追加されたクローンセット.

Deleted そのバージョン間において, 消滅したクローンセット.

Stable そのバージョン間において, 一切の変化が起きなかったクローンセット.

一貫した修正はタイプ 1, 2 のコードクローンだけに生じるとは限らず, タイプ 3, 4 のコードクローンにも適用しなければならない可能性がある. 図 2 は, 一貫した修正が行われたコードクローンの例である. 図 2 の関数 `AcendingSort()`,`DescendingSort` はそれぞれ昇順, 降順のバブルソートを行う関数であり, 5 行目の比較において `list[j-1]` と比較すべきところ

を, `list[j+1]` と比較するという共通のバグを含んでいる. 従って, 一方を修正した際に, 一貫した修正の対象として CloneNotifier はこのコードクローンを通知すべきであると考えられる. しかし, このコードクローンは 5 行目の比較演算子が異なるためタイプ 3 コードクローンに分類される. それゆえ, CloneNotifier ではこのコードクローンを検出できない. その原因は, CCFinder がタイプ 3, 4 のコードクローンを検出できないことに起因する. コードクローンの変更管理においては, 全てのタイプを検出可能なコードクローン検出ツールを利用する方が望ましいと考えられる.

そこで, 本研究では, 2.2.2 節で説明した全てのタイプのコードクローンを検出できる山中らのツールを利用した, 全てのタイプに対応したコードクローン変更管理システムを提案する. このシステムがあれば, CloneNotifier が対応していないタイプ 3, 4 のコードクローンの修正を検出することができるため, より多くのコードクローンに対する一貫した修正漏れを防ぎ, より信頼性の高いシステムを開発するために役立つと期待できる.

3　提案する関数クローン変更管理システム

本研究では, 2.2.2 節で説明した山中らのツールを利用して, 全てのタイプのコードクローンに対応した関数クローン変更管理システムを開発した. 開発においては, 2.3 節で説明した CloneNotifier をベースにし, コードクローン検出部分を CCFinder（2.2.1 節参照）から山中らのツールに置き換える形で実装を行った. 本節では, 開発した関数クローン変更管理システムについて説明する.

3.1　関数クローン変更管理システムの概要

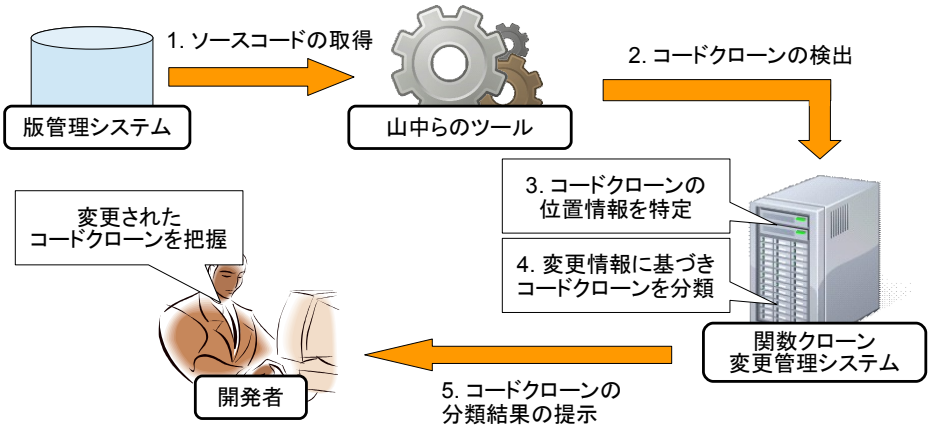


図 3: 提案システムの通知プロセス

開発した関数クローン変更管理システム（以降, 提案システムと呼ぶ）が, 変更された関数クローンを開発者に通知するプロセスの概要を図 3 に示す. 図 3 に示すように, 提案システムによる通知は以下のような手順で実施される.

1. 版管理システムから最新のソースコードを取得する. この際, 以前のバージョンのソースコードを移動して保持する.
2. 取得した最新のソースコードと, 以前のバージョンのソースコードから関数クローンの検出を行う.
3. 関数クローンの位置特定を行う.

4. 関数クローンの検出結果と, 変更情報に基づいて関数クローンを分類する. 分類は修正されたか否かの他, 新たに関数クローンとして追加された, 関数クローンではなくなった, 異なるコード片とクローンセットを構成するようになった場合も考慮される.

5. 関数クローンの分類結果を開発者に通知する. 通知方法は電子メール（テキスト）, CSV, html 形式の 3 種類が存在する.

図 1 と比較すれば分かるように, 提案システムと CloneNotifier の通知プロセスはほとんど同様である. また, 提案システムは CloneNotifier をベースに開発しているため, CloneNotifier と同様, 特定の 2 バージョンを指定した比較も可能である.

提案システムと CloneNotifier の実装において, 大きく変更が必要な部分は, 異なる検出ツールへの対応が必要となるコードクローン検出部である. 言い換えれば, コードクローン検出以外の機能（ソースコードの取得や, 結果の通知）に関しては, CloneNotifier のソースコードを概ね流用することが可能である. コードクローン検出部においては, 前述したように使用する検出ツール CCFinder から山中らのツールに置き換える必要がある. また, CCFinder と山中らのツールでは入出力仕様や出力される情報が異なるため, 他の手順の実装も適宜修正を加える必要があった. 提案システムと CloneNotifier の相違点の詳細は, 3.2 節で説明する.

3.2 提案システムと CloneNotifier の相違点

提案システムと CloneNotifier の大きな違いは, 使用する検出ツールのみである. しかしながら, CCFinder と山中らのツールの間にある違いから, 実装や仕様にいくつかの違いが生じる.

実装における, 提案システムと CloneNotifier の相違点（すなわち, 提案システムの開発における, CloneNotifier からの変更点）は以下の通りである.

コードクローン検出ツールの入出力仕様

当然ながら, CCFinder と山中らのツールのそれぞれが要求する入力（コマンドライン引数）や, 出力される情報のフォーマットは異なる. 従って, 提案システム開発においては, コードクローン検出ツールの実行部分, および, 検出結果の読み込みを行う部分を変更する必要があった.

コードクローンの位置情報の特定

CloneNotifier では, コードクローンの位置情報（開始・終了行番号, 列番号）を利用した処理がいくつか存在し, また, 通知するコードクローン情報にも, この位置情報が含まれている. この位置情報は, コードクローンが変更されたかどうか判断する際にも必要なものであり, 欠かすことができない. しかし, CloneNotifier では, CCFinder の検

出結果に含まれるコードクローンの位置情報を利用していた．山中らのツールは，位置情報の代わりに関数クローンの関数名やそれを含むファイル名を出力するため，新たにコードクローンの位置特定を行う機能を実装する必要があった．そのため，提案システムでは，コードクローンの位置特定を行うプロセスが追加されている（図 3）．提案システムでは，関数クローンの関数名とそれを含むファイルの情報に基づいて位置情報を特定する．具体的には，提示されたファイルから該当する関数の宣言部を探索し，その宣言の開始トークン（返り値の型，アクセス修飾子など）をコードクローンの開始位置，終了トークン「}」をコードクローンの終了位置として特定を行う．

仕様における，提案システムと CloneNotifier の相違点は以下の通りである．

ID	分類	LEN	POP	NIF	RAD	RNR	TKS	LOOP	COND	McCabe
92	CHANGED	50	4	3	6	0.860000	15	0	1	1
64	CHANGED	175	10	7	8	0.662857	17	0	0	0
59	CHANGED	240	6	7	8	0.312500	17	0	0	0
60	CHANGED	220	2	2	6	0.781818	25	2	4	6
61	CHANGED	213	7	7	8	0.629108	17	0	0	0
62	CHANGED	191	4	3	6	0.916230	19	0	2	2
63	CHANGED	189	3	3	6	0.915344	19	0	2	2
65	CHANGED	169	2	2	6	0.887574	23	2	2	4
66	CHANGED	155	17	18	8	0.567742	17	0	0	0
67	CHANGED	146	2	2	6	0.931507	22	2	2	4
68	CHANGED	138	13	7	8	0.673913	17	0	0	0
69	CHANGED	121	2	4	5	0.917355	18	0	3	3
70	CHANGED	120	17	7	8	0.625000	17	0	0	0
71	CHANGED	72	2	2	5	0.944444	16	0	1	1
72	CHANGED	69	2	2	5	0.971014	24	1	1	2
73	CHANGED	67	2	4	6	0.656716	13	0	0	0

図 4: CloneNotifier のメトリクス情報

メトリクス情報の有無

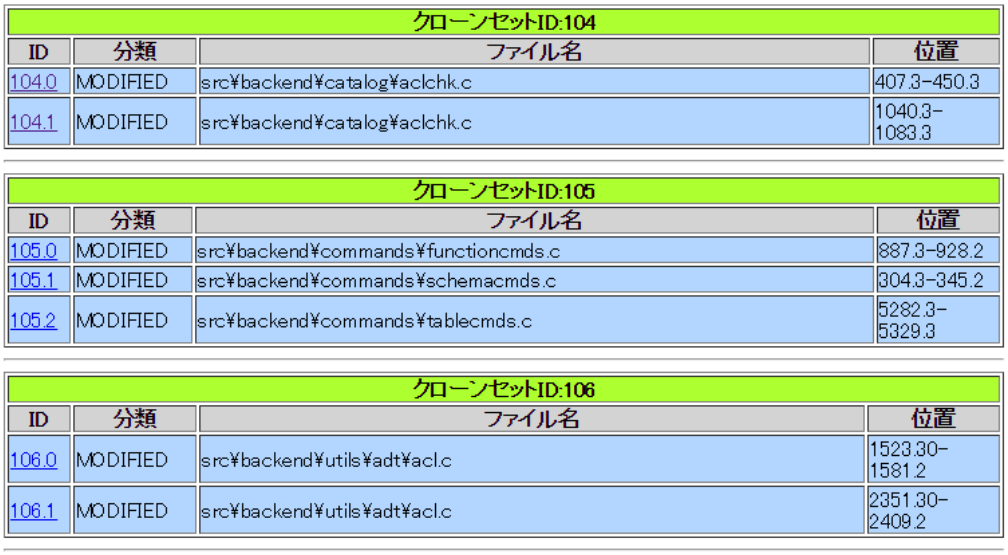
CloneNotifier では，図 4 のようなクローンセットに関するメトリクスの情報が通知される．このメトリクス情報は，CCFinder が出力する結果に含まれているものである．しかし，山中らのツールの出力結果には，メトリクス情報が存在しない．そのため，提案システムでは，クローンセットに関するメトリクス情報は通知されない．

なお，クローンセットに関するメトリクス情報の多くは，再度ソースコードを分析することで算出することは可能である．しかしながら，「全てのタイプのコードクローンの

変更を検出する」という本研究の目的とは直接関係はしないと考えられるため, 今回は実装を行わなかった.

関数名の情報の有無

提案システムは, 関数単位でコードクローンを検出する. そのため, 開始・終了行番号, 列番号だけでなく, 該当する関数名も通知される方が, 開発者がどの部分が変更されたコードクローンとして検出されているのか理解しやすくなると考えられる. 提案システムが利用する山中らのツールの出力結果には, コードクローンとなっている関数名が含まれており, 通知情報に関数名を追加することは容易であったため, 提案システムではコードクローンとなっている関数名の情報も通知するようにしている.



クローンセットID:104

ID	分類	ファイル名	位置
104.0	MODIFIED	src¥backend¥catalog¥aclchk.c	407.3-450.3
104.1	MODIFIED	src¥backend¥catalog¥aclchk.c	1040.3-1083.3

クローンセットID:105

ID	分類	ファイル名	位置
105.0	MODIFIED	src¥backend¥commands¥functioncmds.c	887.3-928.2
105.1	MODIFIED	src¥backend¥commands¥schemacmds.c	304.3-345.2
105.2	MODIFIED	src¥backend¥commands¥tablecmds.c	5282.3-5329.3

クローンセットID:106

ID	分類	ファイル名	位置
106.0	MODIFIED	src¥backend¥utils¥adt¥acl.c	1523.30-1581.2
106.1	MODIFIED	src¥backend¥utils¥adt¥acl.c	2351.30-2409.2

図 5: CloneNotifier が通知するクローンセット情報

図 5 は, CloneNotifier において, html 形式で通知されるクローンセット情報の一例である. 図 5 に示されるように, CloneNotifier では, 各クローンセットに含まれるコードクローンの ID, 分類, ファイル名, 位置に関する情報が表示される. また, 図 6 は, 提案システムにおいて, html 形式で通知されるクローンセット情報の一例である. 図 6 を見ると, 提案システムでは, CloneNotifier と同様, 各クローンセットに含まれるコードクローンの ID, 分類, ファイル名, 位置に関する情報を表示するのに加えて, メソッド名の情報が表示されていることがわかる.

クローンセットID:9

ID	分類	ファイル名	位置	メソッド名
9.0	MODIFIED	src¥backend¥optimizer¥path¥costsize.c	140.1-174.1	cost_seqscan
9.1	MODIFIED	src¥backend¥optimizer¥path¥costsize.c	620.1-646.1	cost_tidscan
9.2	STABLE	src¥backend¥optimizer¥path¥costsize.c	652.1-678.1	cost_subqueryscan
9.3	MODIFIED	src¥backend¥optimizer¥path¥costsize.c	684.1-709.1	cost_functionscan

クローンセットID:10

ID	分類	ファイル名	位置	メソッド名
10.0	MODIFIED	src¥backend¥optimizer¥path¥costsize.c	534.1-569.1	cost_bitmap_and_node
10.1	MODIFIED	src¥backend¥optimizer¥path¥costsize.c	577.1-614.1	cost_bitmap_or_node

クローンセットID:11

ID	分類	ファイル名	位置	メソッド名
11.0	MODIFIED	src¥backend¥optimizer¥path¥pathkeys.c	431.1-472.1	compare_pathkeys
11.1	STABLE	src¥backend¥optimizer¥path¥pathkeys.c	492.1-520.1	compare_noncanonical_pathkeys

図 6: 提案システムが通知するクローンセット情報（C 言語）

各項目の詳細な意味は以下の通りである.

ID 各コードクローンに割り当てられた通し番号である. “x.y” の形式で表され, x には, そのコードクローンが属しているクローンセットの ID, y には出力順に割り当てられた連番が入る.

分類 システムを適用したバージョン間におけるコードクローンの状態を表している. 分類には, 以下の 5 種類がある.

STABLE 適用バージョン間において, 一切の変更が起きなかったコードクローン.

ADDED 修正前のバージョンではコードクローンとして検出されないが, 修正後のバージョンではコードクローンとして検出されたコードクローン.

DELETED 修正前のバージョンではコードクローンとして検出されたが 修正後のバージョンではコードクローンとして検出されなかったコードクローン.

MOVED 修正前後ともにコードクローンとして検出されるが, 所属するクローンセットが修正前後で異なるコードクローン.

MODIFIED 適用バージョン間において変更があり, かつ, 所属するクローンセットに変化がないコードクローン.

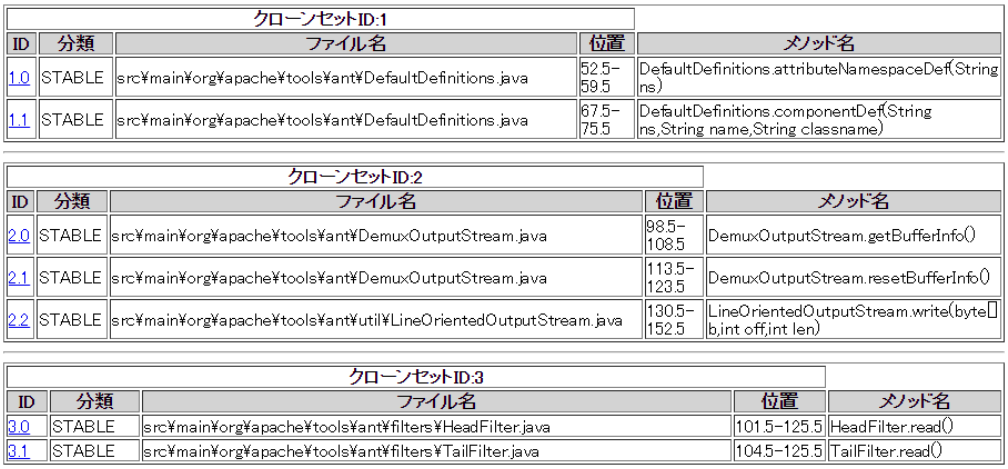
ファイル名 そのコードクローンを含んでいるソースファイルを表す. ファイル名は, “src” ディレクトリからの相対パスで表示される.

位置 「ファイル名」に記載されたソースファイルにおいて, そのコードクローンが存在する位置を表す. “x.y-z.w” の形式で表され, x がコードクローンの開始行, y が開始列, z が終了行, w が終了列を表す.

メソッド名 提案システムにおいて, コードクローンとなっている関数の名前を表す. C 言語では, 図 6 のように, 関数名のみが表示される. Java の場合は, 図 7 のように, その関数が定義されているクラスや, 引数も表示している. 前者は, 内部クラスなどにより, 同じファイル中に同じ名前の関数が宣言されている場合, そのことが一目でわかるように考慮したものであり, 後者は, オーバーロードを考慮したものである.

対応するプログラミング言語

提案システムが対応しているプログラミング言語は, Java と C のみである. 一方で, CloneNotifier は Java と C だけでなく, COBOL や C#などの多数のプログラミング言語に対応している. この対応するプログラミング言語の違いは, 使用するコードクローン検出ツールの対応状況に依存している. すなわち, 山中らのツールが現状, Java



クローンセットID:1

ID	分類	ファイル名	位置	メソッド名
1.0	STABLE	src¥main¥org¥apache¥tools¥ant¥DefaultDefinitions.java	52.5-59.5	DefaultDefinitions.attributeNamespaceDef(String ns)
1.1	STABLE	src¥main¥org¥apache¥tools¥ant¥DefaultDefinitions.java	67.5-75.5	DefaultDefinitions.componentDef(String ns,String name,String classname)

クローンセットID:2

ID	分類	ファイル名	位置	メソッド名
2.0	STABLE	src¥main¥org¥apache¥tools¥ant¥DemuxOutputStream.java	98.5-108.5	DemuxOutputStream.getBufferInfo()
2.1	STABLE	src¥main¥org¥apache¥tools¥ant¥DemuxOutputStream.java	113.5-123.5	DemuxOutputStream.resetBufferInfo()
2.2	STABLE	src¥main¥org¥apache¥tools¥ant¥util¥LineOrientedOutputStream.java	130.5-152.5	LineOrientedOutputStream.write(byte[] b,int off,int len)

クローンセットID:3

ID	分類	ファイル名	位置	メソッド名
3.0	STABLE	src¥main¥org¥apache¥tools¥ant¥filters¥HeadFilter.java	101.5-125.5	HeadFilter.read()
3.1	STABLE	src¥main¥org¥apache¥tools¥ant¥filters¥TailFilter.java	104.5-125.5	TailFilter.read()

図 7: 提案システムが通知するクローンセット情報（Java）

と C のみに対応しているため, 提案システムでもこの 2 つのプログラミング言語のみを対象とした.

4　評価実験

本研究では, 開発した関数クローン変更管理システム（提案システム）の有用性を評価するための 2 つの評価実験を行った. 本研究で実施する評価実験は, 以下の通りである.

1. 提案システムにより実際に検出されるコードクローンを調査することで, 提案システムが有用な修正クローンセットをどの程度検出できるかを評価する. もし, 提案システムを実際に適用しても, 一貫した修正が行われた, あるいは, 一貫した修正が必要な可能性がある（特に, タイプ 3, タイプ 4 の）有用なクローンセットをあまり検出できないのであれば, 提案システムの有用性は低くなってしまう. そのため, どの程度の数の有用なクローンセットを検出できるのか, 実際に確認する必要がある.

2. CloneNotifier と提案システムの各々が検出する修正クローンセットを比較し, どちらがより有用であるか評価する. もし, CloneNotifier より有用な結果が得られることが証明できれば, 提案システムのみを利用すればよいことを提示することができるかもしれない. 逆に, CloneNotifier より有用な結果が得られなければ, 提案システムの有用性が低くなってしまう可能性がある. そのため, 2 つのシステムを比較した評価が必要である.

以降, 本節では, 上記に述べた 2 つの評価実験の詳細と結果, そこから得られる考察について説明する.

4.1　提案システムにより検出できるコードクローンの調査

4.1.1　実験内容

提案システムの主な目的は, 一貫した修正が行われた, あるいは, 一貫した修正が必要な可能性があるタイプ 3, タイプ 4 のクローンセットを検出することにある. もし実際に適用してもこのようなクローンセットが検出されない, あるいは, ごく少数しか存在しないのであれば, 提案システムが有用であるとは言えない.

そこで, 本実験では, 実際のプロジェクトの過去の開発履歴を対象に, 実際に提案システムを適用して得られる結果に基づいて, 提案システムの有用性に関する評価を行う.

評価対象には, PostgreSQL[1] と呼ばれる C 言語で記述されたデータベース管理システムを使用する. 提案システムを適用する期間は 2005/01/01 から 2005/06/30 までの 6ヶ月間とし, 1 週間単位で全 26 期間に対する修正クローンセットの検出を行った. PostgreSQL を評価対象にした理由として, 開発リポジトリが公開されており, 過去の開発履歴を容易に取得でき

[1] http://www.postgresql.org/

表 1: 提案システムで検出された PostgreSQL の修正クローンセットの数

notCN	その他	合計
126	219	345

ることが挙げられる．また, PostgreSQL は様々なコードクローン研究で利用され [10, 14], ある程度の量のコードクローンが存在していることが判明していることも理由である．対象システムにコードクローン自体が存在しなかった場合, 本実験は意味を成さないため, ある程度のコードクローンが存在する対象を選択する必要があった．

実際の評価手順は以下のようになる．ここでは, 2005/01/01 から 2005/01/07 までの期間に適用する場合を例に説明を行う．

1. 提案システムを適用する 2 バージョンのソースコードを入手する．2005/01/01 時点のソースコードは, 2005/01/01 の 0 時 0 分 0 秒時点でのソースコードとする（以降, これを旧バージョンと呼ぶ）．すなわち, 取得するのは 2005/01/01 の 0 時 0 分 0 秒以前における最終バージョンとなる．2005/01/07 時点のソースコードは, 2005/01/08 の 0 時 0 分 0 秒以前の最新ソースコードとする．すなわち, 取得するのは 2005/01/07 の最終バージョンとなる（以降, これを新バージョンと呼ぶ）．なお, 最終期間（2005/06/25 から 2005/06/30 まで）は 7 日間ではないが, 2005/06/30 時点を新バージョンとする．

2. 検出結果に基づいて, 1 つでも修正された関数クローンを含むクローンセットを抽出する．

3. 抽出したクローンセットを以下の 3 種類に分類する．

   - 全ての関数クローンが一貫した修正を受けた．
   - 修正をされていない場合を含み, 一貫した修正を受けていない関数クローンが存在する．しかし, 修正を受けていない関数クローンにも, その修正を適用できる可能性がある．
   - 一貫した修正を受けていない関数クローンが存在し, その修正を適用することはできない．または, 不要であることが明らかにわかる．

4.1.2 結果と考察

本実験により得られた, PostgreSQL において修正されたクローンセットの数を表 1 に示す．表 1 において, “notCN” は CloneNotifier で検出できない（タイプ 3, 4）と考えられる修

表 2: CloneNotifier で検出できない修正クローンセットの内訳

consistent	inconsistent	comment and forming	その他	合計
73	16	32	5	126

正クローンセットである．“その他”は, CloneNotifier で検出可能な修正クローンセットの他, そのバージョンで新たに追加された関数クローンなど, コード自体の修正以外の理由で検出されたものも含まれている．表 1 から, 提案システムで検出された修正が行われたクローンセットのうち, 約 4 割が CloneNotifier では検出不可能な（タイプ 3, 4）クローンセットであることがわかる．

また, CloneNotifier で検出できない修正クローンセットを分類すると表 2 のようになる．表 2 において, “consistent”はクローンセット中の全てのクローンが一貫した修正を受けたものの数を示している．“inconsistent”は一貫した修正を受けていないクローンが存在し, なおかつ, そのクローンに一貫した修正の一部でも適用できる可能性があるものを示している．“comment and forming”はコード整形やコメントの修正のみ行われた場合である．この場合, 一貫しているか否かは考慮していない．ドキュメント作成やコーディング規約の遵守の観点から見ると, 整形やコメント修正が行われたクローンセットは有用であるとも考えられる．しかし, バグの修正漏れ等のコードに対する一貫した修正の問題とは本質が異なると考えられるため, 本実験では別途数えている．“その他”は一貫した修正を受けていないクローンが存在し, 修正を適用することができない, あるいは, 適用する必要が無いことがコードから明らかに読み取れるものである．

提案システムでは, コードクローンの修正を自動的に検出し, それを開発者に通知することで, コードクローンに一貫した修正が行われたか, あるいは, 行う必要があるかを効率良くチェックできるようにすることを目的としている．従って, 提案システムにとって有用なクローンセットは, 実際に一貫した修正が行われたクローンセット（表 2 の “consistent”）, 及び, 一貫した修正が適用可能なクローンセット（表 2 の “inconsistent”）であると考えられる．表 2 より, 提案システムで検出可能だが, CloneNotifier で検出不可能な修正クローンセットのうち, 約 7 割が有用なクローンセットであるとわかる．従って, 本システムでのみ検出可能なコードクローンの修正の多くが有用なクローンセットであり, これらを検出できることは, 提案システムの大きな強みであると考えられる．

本実験において実際に検出されたクローンセットの例を図 8, 図 9 に示す．図 8, 図 9 は, 本実験において実際に検出された有用な修正クローンセットの 1 つである．修正は, 2005/03/12 から 2005/03/18 までの期間に行われたものであり, 図 8 が修正前（2005/03/12 時点）の, 図 9 が修正後（2005/03/18 時点）のソースコードを示している．これらの図に記載された

```
1   static void
2   printtup(HeapTuple tuple, TupleDesc typeinfo, DestReceiver *self)
3   {
·
·    前略
·
4     /*
5      * send the attributes of this tuple
6      */
7     for (i = 0; i < natts; ++i)
8     {
9       PrinttupAttrInfo *thisState = myState->myinfo + i;
10      Datum  origattr = myState->values[i],
11             attr;
12      if (myState->nulls[i] == 'n')
13      {
14        pq_sendint(&buf, -1, 4);
15        continue;
16      }
·
·    後略
·
17  }
```

```
1   static void
2   printtup_internal_20(HeapTuple tuple, TupleDesc typeinfo, DestReceiver *self)
3   {
·
·    前略
·
4     /*
5      * send the attributes of this tuple
6      */
7     for (i = 0; i < natts; ++i)
8     {
9       PrinttupAttrInfo *thisState = myState->myinfo + i;
10      Datum  origattr = myState->values[i],
11             attr;
12      bytea  *outputbytes;
13      if (myState->nulls[i] == 'n')
14        continue;
·
·    後略
·
15  }
```

図 8: 有用な修正クローンセットの例 (タプル出力に関するクローン:修正前)

```
1   static void
2   printtup(TupleTableSlot *slot, DestReceiver *self)
3   {
·
·    前略
·
4     /*
5      * send the attributes of this tuple
6      */
7     for (i = 0; i < natts; ++i)
8     {
9       PrinttupAttrInfo *thisState = myState->myinfo + i;
10      Datum  origattr = slot->tts_values[i],
11             attr;
12      if (slot->tts_isnull[i])
13      {
14        pq_sendint(&buf, -1, 4);
15        continue;
16      }
·
·    後略
·
17  }
```

```
1   static void
2   printtup_internal_20(TupleTableSlot *slot, DestReceiver *self)
3   {
·
·    前略
·
4     /*
5      * send the attributes of this tuple
6      */
7     for (i = 0; i < natts; ++i)
8     {
9       PrinttupAttrInfo *thisState = myState->myinfo + i;
10      Datum  origattr = slot->tts_values[i],
11             attr;
12      bytea  *outputbytes;
13      if (slot->tts_isnull[i])
14        continue;
·
·    後略
·
15  }
```

図 9: 有用な修正クローンセットの例 (タプル出力に関するクローン:修正後)

2 つの関数は共にソースファイル printtup.c に含まれている．なお, 行番号は説明のために再度割り当てたものであり実際とは異なる．2 つの関数 printup, printup_internal_20 は共にデータベースのタプルの情報出力に関する関数であり, 多くの共通処理を含む関数クローンとなっている．図 8, 図 9 から, 2 つの関数は, printup_internal_20 が独自のローカル変数を定義している（printup_internal_20 の修正前後共に 12 行目）, printup にのみ関数呼出しが存在する（printup の修正前後共に 14 行目）などの差分を含むため, この関数クローン（及び図のコード片）は, タイプ 3 であることがわかる．従って, CloneNotifier では検出できない．しかし, 修正された内容（図 9 の赤字部分）を見ると, その修正が同様のものであることがわかる．従って, これら 2 つの関数クローンに対する修正は一貫した修正であり, これらは有用なクローンセットの 1 つであるといえる．

また, 図 10, 図 11 はデータベースサーバの制御に関する関数クローンである．修正は, 2005/04/30 から 2005/05/06 の期間に行われたものであり, 2 つの関数は共にソースファイル pg_ctl.c 中に含まれている．図 10, 図 11 に示されているコード片は, 2 つの関数のエラー処理に関する部分である．図 10 より, 関数 do_restart の 10 行目にある関数呼出しを行うコードが, 関数 do_stop 中には存在しないため, 2 つの関数はタイプ 3 のコードクローンであるといえる．しかし, 図 11 を見ると, 関数 do_restart に対してのみ修正（7 行目の if 文を追加）が加えられており, 関数 do_stop には, この修正が行われていない．従って, この事例では, 関数 do_stop に対して同様の一貫した修正を加えるべきかどうか検討すべきであると考えられる．ゆえに, この関数クローンに対する修正は, 有用なクローンセット修正事例の 1 つであるといえる．

4.2 提案システムと CloneNotifier の比較評価

4.2.1 実験内容

4.1 節の評価により, 提案システムを用いることで多くの有用なクローンセット修正事例を検出できることが確認できた．しかし, CloneNotifier と提案システムは異なる手法のコードクローン検出ツールを利用しているため, 提案システムの結果が, CloneNotifier の結果を全て内包しているとは限らない．そのため, どちらのシステムがより有用なクローンセットを検出できるのかを比較する必要がある．そこで, 本実験では, CloneNotifier と提案システムのそれぞれを同じプロジェクトに適用し, 有用なクローンセットの割合を比較することで評価を行う．

評価対象のプロジェクトは, 4.1 節と同様にした．すなわち, PostgreSQL の 2005/01/01 から 2005/06/30 までの 6ヶ月間の開発リポジトリを対象に, 1 週間単位で全 26 期間に対する修正クローンセットの検出を行う．PostgreSQL を評価対象に選択した理由は, 4.1.1 節で説

明した通りである.

本実験の具体的な手順は, 以下の通りである.

1. PostgreSQL の対象期間（2005/01/01 から 2005/06/30 を 1 週間単位で区切った 26 期間）のすべてに対して, CloneNotifier を適用し, 各期間で修正されたクローンセットの一覧を得る. 同様に, 提案システムを PostgreSQL の対象期間に適用し, 各期間で修正されたクローンセットの一覧を得る.

2. 手順 1. で得られた CloneNotifier の結果から, 50 件の修正クローンセットをランダムに選択する. また同様に, 提案システムの結果からも, 50 件の修正クローンセットをランダムに選択する. 各々のシステムから 50 件のクローンセットを選択する理由は, 評価の際に手作業でクローンセットの各コードクローンを確認する必要があったためである. 特に, CloneNotifier においては, 合計で 1000 件を超える結果を出力するため, これを手作業で全て確認することは困難であると考えられた. そのため, 本実験では, 手作業で確認できることが見込める数として, 各システムで 50 件ずつ, 合計で 100 件を選択した.

3. 手順 2. で選択した各 50 件の修正クローンセットのうち, 有用なクローンセットの数を各システムごとに算出する. 本実験では, 2 つの異なる視点で, 有用なクローンセットの判定を行う. そのため, ここで得られる有用なクローンセットの数は, 2 つのシステム（CloneNotifier と提案システム）と, 2 つの観点に対して 4 種類となる. 2 つの観点それぞれにおける, 有用なクローンセットの定義は以下の通りである.

   (a) 一貫した修正, あるいは, それが必要な可能性のあるクローンセットを検出できているか. すなわち, 以下のクローンセットを有用なクローンセットとみなす.

   - クローンセット中の全てのクローンに対して, 一貫した修正が行われているもの.
   - クローンセット中の一部に一貫した修正が行われており, 残りはその修正が行われていない. しかし, 残りにも同様の修正が適用できる可能性があると外見上で判断できるもの.

   有用なクローンセットであるかどうかの確認は手作業で実施した. その際, 以下のような基準を元にして作業を実施した.

   - 2 つのコードの修正された部分だけを見て, それが修正前後共にタイプ 1, あるいはタイプ 2 のコードクローンの定義を満たしている場合. すなわち, 同じ実装のコードを同じように修正している場合は, 一貫した修正とみなす. ま

た, 修正前のみタイプ 1, あるいはタイプ 2 のコードクローンの定義を満たす場合は, 一貫した修正が適用可能と判断する.

- 2 つのコードに対して, 三項演算子の解消や, null チェックの方法の変更など, 明らかに同じ内容の修正が行われている場合は, 一貫した修正と見なす. また, 同様の修正を適用できる箇所があれば, 一貫した修正が適用可能と判断する.
- 2 つのコードに対して, タイプ 1, あるいは, タイプ 2 のコードクローンとなるコードが追加された場合は, 一貫した修正とみなす. また, 一方の追加されたコードの前後を見て, 前後の少なくともどちらか一方と同じコードが他方（コード追加が行われていないもの）にも存在している場合, その直後（あるいは, 直前）に同じコードが追加できる可能性があると考え, 一貫した修正が適用可能と判断する.
- 2 つのコードに対して, タイプ 1, あるいは, タイプ 2 のコードクローンとなるコードが削除された場合は, 一貫した修正とみなす. また, 一方で削除されたコードと同じものが, コード削除が起きていない他方にも存在している場合, 一貫した修正が適用可能と判断する.
- 修正が行われていない場合（クローンセットを構成するコードクローンが追加された場合など）は, この評価において有用かどうかの判定はできない. しかし, そのようなクローンセットが有用かどうかは別に議論が必要になると考えられる. 本評価では, その議論は対象としないため, これらのクローンセットは除外して考える.

(b) その修正クローンセットが検出されたバージョン間において, クローンを追跡できているか. すなわち, 修正クローンセットとして検出された以降もコードクローンとして変更管理を継続できる場合に有用であるとみなす. この観点においては, 以下のものを有用なクローンセットとみなす.

- “Changed” クローンセットであり, かつ, それを構成する全てのクローンの状態が Deleted 以外であるもの. 状態 Added のコードクローンが存在しない場合は, 完全に同じクローンセットを追跡できているため有用とみなせる. また, 状態 Added のコードクローンが存在している場合も, 追跡すべき新たなコードクローンを検出できているため有用と考えられる.
- “Deleted” クローンセットであるが, その全てのコードクローンを内包する “Changed”, あるいは. “New” クローンセットが存在するもの. この場合, システムのクローンセットの等価判定の都合上 “Deleted” クローンセット, す

表 3: 一貫した修正検出の観点から見た有用な修正クローンセットの数

	有用である	部分的に有用	有用でない	除外	合計
CloneNotifier	17	5	17	11	50
提案システム	21	4	10	15	50

なわち消滅したクローンセットの扱いになっているが, 実際には, 新たに別のコードクローンとともに追跡が継続していることを意味する. この場合は, 追跡そのものは継続できているため有用であるとみなす.

- そのバージョンで新たに出現したクローンセット（構成する全てのコードクローンの状態が “Added” であるような “New” クローンセット）は, その時点から追跡が開始されたものであり, 追跡の “継続” は評価できない. また, コード自体が消滅したことによりコードクローンではなくなった場合は, 追跡の継続は明らかに不可能ではあるが, 有用かどうかは別の議論が必要と考えられる. 本評価では, その議論は対象としないため, これらのクローンセットは除外して考える.

4. それぞれのシステムと観点において, 母集団（50 件から各除外対象を取り除いたもの）に対する有用なクローンセットの割合を求め, 2 つのシステム間で比較を行う. 絶対数ではなく, 割合で評価を行った理由としては, CloneNotifier と提案システムの検出する修正クローンセットの数が大きく異なる可能性があることが挙げられる. CloneNotifier はコード片単位, 提案システムは関数単位のコードクローンを検出粒度とするため, 一般的に, CloneNotifier の方が個数としては多くのクローンセットを検出することになると考えられる. そのため, 必然的に, CloneNotifier の方が有用なクローンセットの絶対数が多くなる可能性が高い. また, 本実験では, 作業の困難さから検出結果の一部を選択して評価を実施している. このことから, 絶対数よりも割合で比較する方が, 提案システムとの比較評価には適していると考えられる.

4.2.2 結果と考察

表 3 に, CloneNotifier と提案システムのそれぞれにおける, 一貫した修正検出の観点から見た場合（4.2.1 節の手順 3-(a)）の有用な修正クローンセットの数を示す. また, 表 4 には, 一貫した修正検出の観点から見た有用な修正クローンセットの割合（分母は 50 件から除外対象を取り除いた数）を示している. 表 3, 表 4 において, 縦軸は各システムを, 横軸は各システムの検出結果からランダムに選択した 50 件の修正クローンセットの分類を示している.

表 4: 一貫した修正検出の観点から見た有用な修正クローンセットの割合

	有用である (%)	部分的に有用 (%)	有用でない (%)
CloneNotifier	43.6	12.8	43.6
提案システム	60.0	11.4	28.6

また, “有用である”はクローンセット中の全てのクローンに対して一貫した修正が行われているか, あるいは, 修正が適用できる可能性があると判断できるクローンセットの個数を意味しており, “有用でない”は一貫した修正が全くなく, 適用も難しいと判断できるクローンセットの個数を表している. “部分的に有用”は, クローンセットに行われた修正の一部が有用と判断できるものの数である. すなわち, クローンセット中の全てではない複数のクローンに対して, 修正内容の一部において一貫した修正が行われているか, あるいは, 修正が適用できる可能性があると判断できるクローンセットのことを指す. 最後に, “除外”とは, 有用であるか否かの判断ができないクローンセットのことである. ここでは, コードに対する修正が行われていないクローンセットのことを意味する. このようなクローンセットが生じる理由は, CloneNotifier, および, 提案システムが検出する修正されたクローンセットには, クローンセットを構成するクローンが変化したものを含んでいるためである. また, コード自体が新たに追加された場合や完全に消去された場合も, それは厳密にはコードクローンの “修正”とは異なる種類の変更のため, 一貫した修正が必要か否かの判断が難しい. このようなクローンセットも, 本実験では, “除外”に分類している.

表 4 の結果から, CloneNotifier で検出された一貫した修正の観点で有用な修正クローンセットの数は 43.6%, 提案システムでは 60.0%であり, 提案システムの方が多くの有用なクローンセットを検出しているという結果になった. また, “部分的に有用”なものを含めても, CloneNotifier は 56.4%, 提案システムは 71.4%であり, 提案システムの方が多くの有用なクローンセットを検出しているという結果になる. このような結果になった理由として, 粒度の違いが考えられる. CloneNotifier はコード片単位であり, 提案システムは関数単位でコードクローンを検出する. そのため, CloneNotifier の方が絶対数としては多くのコードクローンを検出することになるが, 粒度が小さいため, 一貫した修正が不要な機能的には異なるクローンや, ある程度, 汎用的に出現する可能性のあるクローンを検出する可能性も高い. 一方で, 関数は一般的に機能単位でまとまっており, 関数クローンであれば類似の機能や処理を行っている可能性が高くなる. そのため, 一貫した修正が必要なクローンセットの割合がより高くなったのではないかと考えられる.

一方で, 提案システムは CloneNotifier に比べて多くの “除外”のクローンセット, すなわち, コード自体への修正が無いクローンセットを検出している. これは, 提案システムが利

用している山中らのツールのコードクローン検出手法に起因しているのではないかと考えられる．CloneNotifier で用いられている CCFinder の検出手法では, トークン列が一致しているものをクローンとしている．そのため, トークン列そのものに変化が無ければ常にコードクローンとして検出されるため, CloneNotifier で検出される “除外”のクローンセットは, 通常はコード自体が新たに追加されたものや完全に消去されたもののみとなる．一方で, 山中らのツールの手法は, 類似度の計算の過程でプロジェクトの全てのソースコードが関係するため, ある修正が全ての関数の類似度に影響する可能性を持つ．そのため, 修正が一切起きていない場合でも, クローンセットを構成するコードクローンが変化し, “除外”のクローンセットとして検出される可能性があると考えられる．

しかし, フィッシャーの正確確率検定を用いて, 表 3 から “除外”を除いた 2×3 分割表に対して有意差検定を行った結果, 有意水準 5%で有意差無しという結果になった．ただし, 本評価のサンプル数が少なかった可能性や, ランダム選択に偏りがあった可能性は否定できない．有意な結果を出すには, より多くの対象に関する評価が必要であると考えられる．

表 5: クローン追跡の観点から見た有用な修正クローンセットの数

	有用である	有用でない	除外	合計
CloneNotifier	16	18	16	50
提案システム	30	12	8	50

表 6: クローン追跡の観点から見た有用な修正クローンセットの割合

	有用である (%)	有用でない (%)
CloneNotifier	47.1	52.9
提案システム	71.4	28.6

表 5 に, CloneNotifier と提案システムのそれぞれにおける, クローン追跡の観点から見た場合（4.2.1 節の手順 3-(b)）の有用な修正クローンセットの数であり, 表 6 は, その割合（分母は除外対象を取り除いた数）を示したものである．表 5, 表 6 において, 縦軸は各システムを, 横軸は各システムの検出結果からランダムに選択した 50 件の修正クローンセットの分類を示している．これらの表において, “有用である”とは各クローンセットが検出されたバージョン間において, その追跡が継続しているものを示しており, “有用でない”は, そのバージョン以降で正しく追跡できなくなるものを意味する．また, “除外”は追跡が継続しているか否かを分類できないクローンセットを示している．具体的には, コード自体が消滅したことで追跡不可能となった場合および, そのバージョンから追跡を開始するクローンセットが

これに当たる.

表 6 から, CloneNotifier のクローン追跡の観点から見た有用なクローンセットは 47.1%, 提案システムは 71.4%であり, 提案システムの方が正しく追跡できているクローンセットの割合が大きいことがわかる. また, 有用でない, すなわち, 正しく追跡できないクローンセットの割合も, CloneNotifier が 52.9%に対して, 提案システムは 28.6%と低くなっており, 提案システムの方が優れた結果を示しているといえる. このような結果になった理由としては, 2 つのシステムが対応するコードクローンのタイプの差が考えられる. CloneNotifier はタイプ 2 までのコードクローンには対応しているが, 少しの一貫しない修正が入っただけで追跡は不可能となってしまう. 一方で, 提案システムはタイプ 3 のコードクローンにも対応している. そのため, 少し一貫しない修正が入った程度ではクローンの関係は壊れず, 継続した追跡が可能となる. タイプ 1, 2 のコードクローンが修正によりタイプ 3 になることも多いため [19], このようなクローンセットでも継続して追跡できる提案システムは, CloneNotifier より優れていると考えられる.

また, 表 5 から “除外” を除いた 2×2 分割表に対して, フィッシャーの正確確率検定を用いて有意差検定を行った結果, 有意水準 5%で有意差ありと言う結果になった. 従って, 追跡性能に関しては, 提案システムは有意に優れた結果を示しているといえる.

以上の評価では, 一貫した修正とクローン追跡のどちらにおいても, 提案システムの方がより有用なクローンセットの割合が高いという結果になった. 有用なクローンセットの割合が高いということは, 言い換えれば, 誤検出が少ないと言うことである. 従って, 本実験の結果からは, 提案システムは CloneNotifier より優れていると結論付けることができる. また, 提案システムは, 検出粒度の違いから CloneNotifier より検出される修正クローンセット数が少なくなる傾向が強い. 本実験でも, CloneNotifier が検出した修正クローンセットの合計は 1216 件に対し, 提案システムでは 340 件であった. ゆえに, もし仮に本実験の結果が一般化できるのであれば, 提案システムは少ない検出数で多くの割合の有用なクローンセットを検出できることになる. もしそうなれば, 提案システムは CloneNotifier より高い精度で修正クローンセットを検出できるといえるだろう.

本実験の結果から, 提案システムが有用であり, クローン変更管理に利用すべきであることがいえる. しかし, 提案システムの検出結果が, CloneNotifier の結果を内包するとは限らない. ゆえに, 提案システムが検出できず, CloneNotifier が検出できるコードクローンが少なからず存在する. 特に, 提案システムが利用している山中らのツールは, 識別子名を利用してコードクローンを検出しているため, 潜在的にタイプ 2 のコードクローン検出に弱いという欠点を持つため, タイプ 2 コードクローンにおいては, CloneNotifier の方が優れている可能性が高い. また, 本実験において選択された各システム 50 件の修正クローンセットにおいても, 全く同じ修正箇所を示すクローンセットは存在しなかった. これらのことを考慮す

ると, 提案システムのみを利用すればよいと言うことはできないが, 修正の内容に応じて提案システムと CloneNotifier をうまく使い分けることができれば, さらに精度の高いクローン変更管理が可能になるかもしれない. 例えば, ほんの数行程度の計算式の修正などであれば, 粒度も小さく, 識別子名が異なる同じ計算式を検出できる CloneNotifier を用い, 仕様変更や関数の多くの部分に渡る修正, ロジックの修正などを行った場合には, 粒度が大きく, ある程度の違いを許容する提案システムを用いるようにすれば, より適切なクローン変更管理が可能になるだろう.

4.3 妥当性への脅威

本実験では, 評価対象として PostgreSQL の特定の期間を選択したが, 期間に応じて行われる修正の量や種類は異なる. 従って, 異なる期間を選択すれば, 本実験とは異なる結果が得られる可能性が考えられる. また, 評価対象に異なるシステムを選択した場合も, 同様に異なる結果が得られる可能性がある. どちらの場合も, タイプ 3 のコードクローンに対する修正が含まれる場合は, 量に差はあれど CloneNotifier には検出できない修正クローンセットが検出できると考えられる. しかし, タイプ 3 のコードクローンに対する修正が存在しない場合, 提案システムが CloneNotifier より有用であるといえる結果が得られないだろう.

また, 本実験では, 修正クローンセットの分類を手作業で実施している. 従って, 分類にヒューマンエラーが含まれる可能性を否定できない. もし誤りがあれば, 本実験の結果は異なるものになると考えられる. しかし, クローンセットに行われた修正が一貫したものであるか否かを判断するには, 最終的には人間がコードを確認しなければならず, 完全な自動化は難しい. よって, 正確な結果を得るためには, より多くの研究者や, 評価対象の開発者の協力を得て, 分類を実施する必要があるだろう. しかし, 提案システムが, 図 8, 図 10 といった CloneNotifier には検出できないタイプ 3 の修正クローンセットを検出できることは, 実験結果から明らかである. ゆえに, 提案システムが CloneNotifier より有用な点を持つことは確実であると考えられる.

4.2 節の実験では, 各システムから 50 件の修正クローンセットをランダムに選択し, 評価を実施した. そのため, 選択されたサンプルが異なれば, 異なる結果になっていた可能性がある. また, 本実験の結果を一般化するには, 各システム 50 件の修正クローンセットでは足りないと考えられる. 一般的に, 提案システムが CloneNotifier より有用であるかどうかを調べるには, より多くのプロジェクトから多数の修正クローンセットを検出し, 調査する必要があるだろう.

5　まとめと今後の課題

本研究では, 修正されたコードのコードクローンを自動的に検出・通知するコードクローン変更管理システム CloneNotifier をベースに, そのコードクローン検出部を CCFinder から山中らのツールに置き換えることで, タイプ 1 から 4 の全てのコードクローンに対応した関数クローン変更管理システムを開発した. また, PostgreSQL に提案システムを実際に適用し, CloneNotifier では検出できない多くの有用なコードクローンを検出できていることを確認した. さらに, CloneNotifier との比較評価を行うことで, 提案システムが CloneNotifier よりも優れた結果を示すことを確認できた.

今後の課題としては, 本研究のさらなる一般化のために, より多くのプロジェクトに対して提案システムを評価することが必要であると考えられる. 4.3 節でも触れたが, 本研究の評価だけでは, 提案システムが一般的にも有用であると述べることは難しい. より多くのプロジェクトに提案システムを適用し, 優れた結果が得られれば, 提案システムの有用性により確信を持てることになるだろう. また, 本実験では精度や性能を視点に評価を実施したが, 実際に開発現場で利用することを考えた場合, 結果の提示方法や操作性といったユーザインタフェースの評価・改善も必要になるだろう. その場合, 実際にユーザに利用してもらうユーザ調査などで, インタフェースに対するアンケートや, 問題を解くなどの作業にかかった時間などに基づいた評価が必要になると考えられる.

さらに, 対応言語の拡張といった, 提案システム自体の改善も必要であると考えられる. CloneNotifier は Java や C, COBOL や C#など数多くのプログラミング言語に対応しているが, 提案システムでは, そのうち Java と C にしか対応していない. これは, 各々が使用するコードクローン検出ツールの対応言語に起因している. ただし, 山中らのツールで用いられている関数中のユーザ定義名の類似度に基づくクローン検出手法はプログラミング言語にほとんど依存せず適用できる. そのため, 関数やそれに類する概念のあるプログラミング言語であれば, 対応は容易と考えられる.

謝辞

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 井上克郎 教授には, 御多忙の中，常に適切な御指導及び御助言を賜りました. 井上 教授の適切な御指導のおかげで, 本論文を完成させることができました. 井上 教授のもとで研究生活を送ることができたこと, 本論文を執筆できたことに厚く御礼申し上げます.

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 松下誠 准教授には, 研究における問題点の提示など, 多くの御助言を賜りました. それらの御助言は, 本研究を遂行する上で非常に役立ちました. 多くの御指導及び御助言を頂いた 松下 准教授に心より深く感謝いたします.

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 石尾隆 助教には, 研究に関してだけではなく, 発表における細かな誤りなど, 様々な御指導を頂きました. それらの御指導は, 非常に勉強になりました. 様々な御指導及び御助言を頂いた 石尾 助教に心より深く感謝いたします.

名古屋大学大学院情報科学研究科附属組込みシステム研究センター/情報システム学専攻 吉田則裕 准教授には, 本研究の構成から, 本論文の執筆に至るまで, 終始適切な御指導及び御助言を頂きました. 本論文を執筆することができたのは, 吉田 准教授の御指導のおかげであると, 心より深く感謝しております.

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 春名修介 特任教授には, 企業在籍時の経験など，様々な観点から本研究に対し適宜適切な御指導及び御助言を頂きました. 貴重な御指導及び御助言を頂いた 春名 特任教授に心より深く感謝いたします.

大阪大学大学院国際公共政策研究科国際公共政策専攻 崔恩瀞 助教には, 学部生の頃から, 研究における様々な段階で御協力を頂きました. また, 研究生活においても, 様々な場面で支えて頂きました. 研究生活を有意義に過ごすことができたのは, 崔 助教のおかげであると心より深く感謝しております.

最後に，様々な御指導，御助言等を頂き, 研究生活を支えてくださった, 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井上研究室の皆様に深く感謝いたします.

参考文献

[1] B. S. Baker. Finding clones with Dup: analysis of an experiment. *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 608–621, 2007.

[2] H. A. Basit and S. Jarzabek. Detecting higher-level similarity patterns in programs. In *Proceeding of the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 156–165, 2005.

[3] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. pp. 368–377, 1998.

[4] I. Baxter, A. Yahin, L. Moura, M. Anna, and L. Bier. Clone detection using abstract syntax trees.

[5] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, Vol. 18, No. 1, pp. 37–58, 2006.

[6] 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.

[7] 肥後芳樹, 楠本真二. プログラム依存グラフを用いたコードクローン検出法の改善と評価. 情報処理学会論文誌, Vol. 51, No. 12, pp. 2149–2168, 2010.

[8] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones.

[9] 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. 1, pp. 654–670, 2002.

[10] 川口真司, 松下誠, 井上克郎. 版管理システムを用いたクローン履歴分析手法の提案. 電子情報通信学会論文誌, Vol. J89-D, No. 10, pp. 2279–2287, 2006.

[11] H. Kim, Y. Jung, S. Kim, and K. Yi. MeCC: memory comparison-based clone detector. pp. 301–310, 2011.

[12] K. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Experiment on the automatic detection of function clones in a software system using metrics. *Automated Software Engineering*, Vol. 3, pp. 77–108, 1996.

[13] B. Laguë, D. Proulx, J. Mayrand, E. M. Merlo, and J. Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. pp. 314–321, 1997.

[14] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192, 2006.

[15] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. pp. 244–253, 1996.

[16] 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.

[17] L. Prechelt, G. Malpohl, and M. Philippsen. Finding plagiarisms among a set of programs with jplag. *Journal of Universal Computer Science*, Vol. 8, No. 11, pp. 1016–1038, 2002.

[18] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Science of Computer Programming*, Vol. 74, No. 7, pp. 470–495, 2009.

[19] R. K. Saha, C. K. Roy, K. A. Schneider, and D. E. Perry. Understanding the evolution of type-3 clones: an exploratory study. In *Proc. of MSR 2013*, pp. 139–148, 2013.

[20] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎, 佐野建樹. コードクローン変更管理システムの開発と実プロジェクトへの適用. 情報処理学会論文誌, Vol. 54, No. 2, pp. 883–893, 2013.

[21] 山中裕樹, 崔恩瀞, 吉田則裕, 井上克郎. 情報検索技術に基づく高速な関数クローン検出. 情報処理学会論文誌, Vol. 55, No. 10, pp. 2245–2255, 2014.

```
1   static void
2   do_stop(void)
3   {
·
·    前略
·
4     else if (pid < 0)
5     {
6       pid = -pid;
7       write_stderr(_("%s: cannot stop postmaster; "
8                      "postgres is running (PID: %ld)\n"),
9                    progname, pid);
10      exit(1);
11    }
·
·    後略
·
12  }
```

```
1   static void
2   do_restart(void)
3   {
·
·    前略
·
4     else if (pid < 0)
5     {
6       pid = -pid;
7       write_stderr(_("%s: cannot restart postmaster; "
8                      "postgres is running (PID: %ld)\n"),
9                    progname, pid);
10      write_stderr(_("Please terminate postgres and try again.\n"));
11      exit(1);
12    }
·
·    後略
·
13  }
```

図 10: 有用なクローンセットの例 (データベース制御に関するクローン：修正前)

```
1   static void
2   do_stop(void)
3   {
·
·    前略
·
4     else if (pid < 0)
5     {
6       pid = -pid;
7       write_stderr(_("%s: cannot stop postmaster; "
8                      "postgres is running (PID: %ld)\n"),
9                    progname, pid);
10      exit(1);
11    }
·
·    後略
·
12  }
```

```
1   static void
2   do_restart(void)
3   {
·
·    前略
·
4     else if (pid < 0)
5     {
6       pid = -pid;
7       if (postmaster_is_alive((pid_t) pid))
8       {
7         write_stderr(_("%s: cannot restart postmaster; "
8                        "postgres is running (PID: %ld)\n"),
9                      progname, pid);
10        write_stderr(_("Please terminate postgres and try again.\n"));
11        exit(1);
12      }
13    }
·
·    後略
·
14  }
```

図 11: 有用なクローンセットの例 (データベース制御に関するクローン：修正後)