Vol.2015-SE-190 No.4
2015/12/15

情報検索技術に基づく関数クローン検出を用いた変更管理システムの開発

佐野 真夢[1,a)]　吉田 則裕[2]　春名 修介[1]　井上 克郎[1]

概要：ソフトウェア保守における問題の 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る．あるコード片にバグがあると判明した場合, そのコードクローンにも同様のバグが存在する可能性がある．従って, 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わなければならない．この際, 構文上一致しているコード片だけでなく, 構文上は異なるが同様の機能を実装しているコード片も調査すべきと考えられる．しかし, こ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．この問題を解決するシステムとして, 修正が行われたコードのコードクローンを自動的に検出し, 通知する機能を持つコードクローン変更管理システムが存在する．しかし, このシステムには, 構文上異なる実装を行っているコードクローンを検出できないという問題がある．そこで, 本研究では, そのような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．また, 本システムにより, 実際にどのようなコードクローンが検出可能であるか調査を行った．その結果, 本システムが実際に構文上の違いを持つコードクローンに対する修正を検出できることを確認した．

キーワード：コードクローン，ソフトウェア保守，情報検索技術

Development of Change Management System Using an IR-based Detection of Function Clones

Manamu Sano[1,a)]　Norihiro Yoshida[2]　Shusuke Haruna[1]　Katsuro Inoue[1]

Abstract: Code clone has been known to be one of problems in software maintenance. It is a code fragment that is identical or similar to other code fragments in source code. If a code fragment has a bug, it is possible that other code clones have similar bugs. Therefore, developers must find all associated code clones to make consistent changes as necessary. They should find not only code clones that are the same syntactically, but also clones that are different syntactically and are implemented the same functionality. However, it is inefficient to do it manually whenever they make changes. In order to solve this problem, a change management system of code clones has been developed. However, this system has a problem that it is unable to detect code clones including syntactic differences. In this study, we developed a novel change management system of code clones that can detect such code clones. In addition, we evaluated what code clones our system can actually detect. Our evaluation shows that our system can actually detect code clones including syntactic differences.

Keywords: Code Clone, Software Maintenance, Information Retrieval Technique

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], [2], [5], [6], [10]．コー

[1] 大阪大学
Osaka University
[2] 名古屋大学
Nagoya University
[a)] m-sano@ist.osaka-u.ac.jp

© 2015 Information Processing Society of Japan

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, 定型処理の実装など様々な要因により生じる [2], [3].

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も同様のバグが存在する可能性が考えられる. 従って, バグ修正を行う際には, 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う必要がある [9], [10]. 性能改善や仕様変更など, 他の理由による修正においても同様のことが考えられる.

また, 一貫した修正を考慮すべきコード片は構文上一致しているコードクローンだけとは限らない. 修正の種類によっては, 構文上は異なるが同様の機能を実装しているコードクローンに対する一貫した修正も検討する必要があると考えられる. しかし, このような修正するコード片に対する全てのコードクローンを, 修正の度に手動で検出することは非常に手間がかかり, 非効率的である.

このような問題を解決するシステムとして, CloneNotifier[13] が挙げられる. CloneNotifier とは, 修正が行われたコード片に対するコードクローンを自動的に検出し, その結果を開発者に通知する機能を持つコードクローン変更管理システムのことである. CloneNotifier を利用すれば, 開発者は通知が行われた際に, 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ばよいので, 修正の際にコードクローンを意識する必要がなく, 開発者の負担を減らすことが可能である. 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[6] と呼ばれるコードクローン検出ツールを利用しているが, これには構文上は異なるが同様の機能を実装しているコードクローンを検出できないという欠点がある. 上述したように, このようなコードクローンも一貫した修正が必要な可能性は考えられるため, 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.

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, 構文上は異なるが同様の機能を実装しているコードクローンも検出できるコードクローン検出ツールを用いて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. また, 本システムを OSS の過去の開発履歴に対して適用し, 実際にどのようなコードクローンが検出されるか調査を実施した. 本調査の結果, 本システムが CloneNotifier では検出できないコードクローンの修正を検出できることが確認した.

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. 関連研究

2.1 コードクローン

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 [3]. 例えば, あるコード片にバグが見つかった場合, そのコードクローンにも同様のバグが含まれる可能性が考えられる. そのため, コードクローンの存在を知ることは重要であるが, 特に大規模ソフトウェアとなると, 開発者がソフトウェア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, 一般的には, 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 [6].

一般的に, 互いにコードクローンとなるコード片の対をクローンペアと呼び, 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をクローンセットと呼ぶ.

コードクローンは, いくつかのタイプに分類することができる. Roy らは, コードクローンを以下の 4 つのタイプに分類している [12].

タイプ 1 空白・コメントの有無, レイアウトなど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, 変数名等のユーザ定義名, 変数の型など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・削除・変更などが行われたコードクローン.

タイプ 4 同一の処理を実行するが, 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.

コードクローンの検出手法は主に, 構文の類似性に着目した検出手法 [2], [6], [10] と意味的な類似性 [4], [8], [14] に着目した検出手法に分類できる. 両者共そのアプローチは様々であるが, 構文の類似性に着目した手法では主にタイプ 1,2,3 を, 意味的な類似性に着目した手法では全タイプのコードクローンを検出できる.

2.2 コードクローン検出ツール

コードクローンの検出のために様々な手法が考案され, それを実装した検出ツールが開発されている. 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を紹介する.

2.2.1 CCFinder

CCFinder[6] は, 構文の類似性に着目した手法を用いたコードクローン検出ツールの 1 つであり, ソースコード中にある同一文字列を字句単位で検索することでコードクローン検出を行う. CCFinder は高いスケーラビリティを有しており, 大規模なソフトウェアに対しても実用的な時間で

© 2015 Information Processing Society of Japan

コードクローンを検出できる．CCFinder は実際に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 [11]．しかし，検出可能なコードクローンはタイプ 1, 2 のみであり，タイプ 3 以上は検出できない．

2.2.2 山中らのツール

山中らのツール [14] は，意味的な類似性に着目した手法を用い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．具体的には，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，登場するユーザ定義名が似ていれば，関数クローンとなる．山中らのツールは全タイプのコードクローンの検出が可能であり，また，他の意味的な類似性に着目した手法を用いた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

2.1 節で述べたように，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，全タイプのコードクローンに適用できる可能性が存在している．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．

上述の問題を解決するために，CloneNotifier[13] を利用することが考えられる．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 のコードクローンを管理できるシステムといえる．

しかし，一貫した修正はタイプ 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はある．**図 1** 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 1 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．

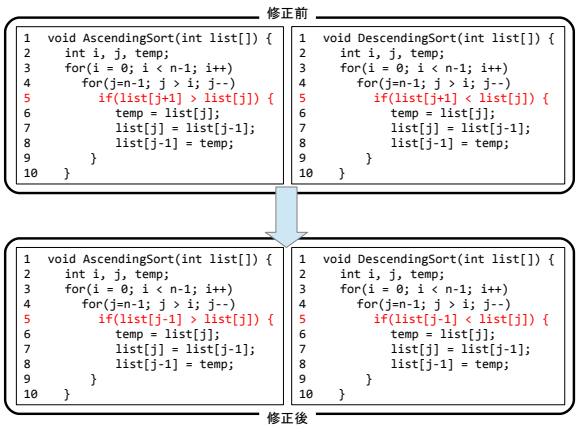

図 1 タイプ 3 コードクローンの一貫した修正の例
Fig. 1 Example of consistent changes for type-3 code clones.

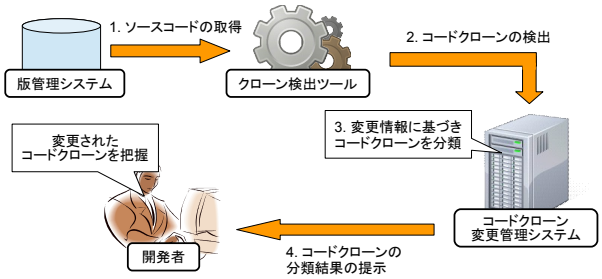

図 2 CloneNotifer の通知プロセス
Fig. 2 Notification process of CloneNotifier.

図 2 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．

© 2015 Information Processing Society of Japan

(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．

3. 関数クローン検出を用いた変更管理システムの開発

本研究では, 2.3 節で説明した CloneNotifier をベースにして, コードクローン検出部分を CCFinder から山中らのツール（2.2.2 節参照）に置き換える形で関数クローン検出を用いた変更管理システムを開発した．山中らのツールを利用することで, 全タイプのコードクローンの変更を検出, 通知することが可能になると期待できる．本節では, 開発した変更管理システムの実装について説明する．

本システムの通知プロセスは, 図 2 に示した CloneNotifier の手順とほとんど同様であり, 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) 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．
(5) 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．

基本的な部分は CloneNotifier と同様であるため, 特定の 2 バージョンを指定した比較も可能である．

本研究では, CloneNotifier のコードクローン検出部分に着目している．従って, 手順 2. のコードクローン検出以外の機能に関しては, CloneNotifier のソースコードを流用することが可能である．コードクローン検出に関しては, 使用するツールを 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る．

本システムの出力情報には, コードクローンの位置情報（開始・終了行番号, 列番号）が含まれる．CloneNotifier においては, CCFinder の検出結果にコードクローンの位置情報が存在していたため, その情報を用いることができた．しかし, 山中らのツールは関数クローンの関数名やそれを含むファイル名は出力するが, その位置情報は出力しない．そのため, 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．すなわち, 手順 2. の実施後にクローンとなっている関数名とそれを含むファイルの情報から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, アクセス修飾子等）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．

4. 評価実験

本研究では, 開発したコードクローン変更管理システムの有用性を評価するための調査を行った．本節では, 実施した調査の内容と結果, そこから得られる考察について説明する．

4.1 実験内容

本実験では, 開発したコードクローン変更管理システムの有用性に関する評価を行う．具体的には, 評価対象となるシステムの過去の開発履歴に対して実際に本システムを適用することで, どのような結果が得られるかを調査する．

本システムの主な目的は, CloneNotifer では検出できないタイプ 3, タイプ 4 のクローンセットの修正を見つけ出すことにある．このようなクローンセットを検出できなければ, 本システムが有用であるとは言えない．ゆえに, 本実験では, CloneNotifer では検出できないクローンセットの修正に着目し, これらがどの程度, 検出できているかを評価する．

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． 期間は 2005/01/01〜2005/06/30 までの 6 ヶ月間とし, 1 週間単位で全 26 期間の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 [7], [10], 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, 2005/01/01〜2005/01/07 の期間を例に説明を行う．

(1) 本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,

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

© 2015 Information Processing Society of Japan

表 1　本システムで検出された PostgreSQL の修正クローンセットの数
Table 1　The number of modified clone sets in postgreSQL using the proposed system.

notCN	その他	合計
126	219	345

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（2005/06/25〜2005/06/30）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．

4.2　実験結果と考察

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 で検出できないと考えられる修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, CloneNotifier で検出不可能な修正クローンセットが約 4 割検出されていることがわかるが,

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．

本システムでは, 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, 本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, 本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, 本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 3** に示す．図 3 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, 2005/03/12〜2005/03/18 の期間に行われたものであり, 2 つの関数は共に `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 3 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 3 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 4** 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, 2005/04/30〜2005/05/06 の期間に行われたものであり, 2 つの関数は共に `pg_ctl.c` に含まれている．図 4 のコード片は, 2 つの関数のエラー処理に関する部分である．`do_restart` の修正前における 10 行目の呼出し関数にあたるコードが `do_stop` 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 4 の修正では, `do_restart` に対してのみ 7 行目（修正後）の if 文を追加しており, `do_stop` にはこの修正が

© 2015 Information Processing Society of Japan

表 2　CloneNotifier で検出できない修正クローンセットの内訳
Table 2　Classification of modified clone sets that CloneNotifier cannot detect.

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

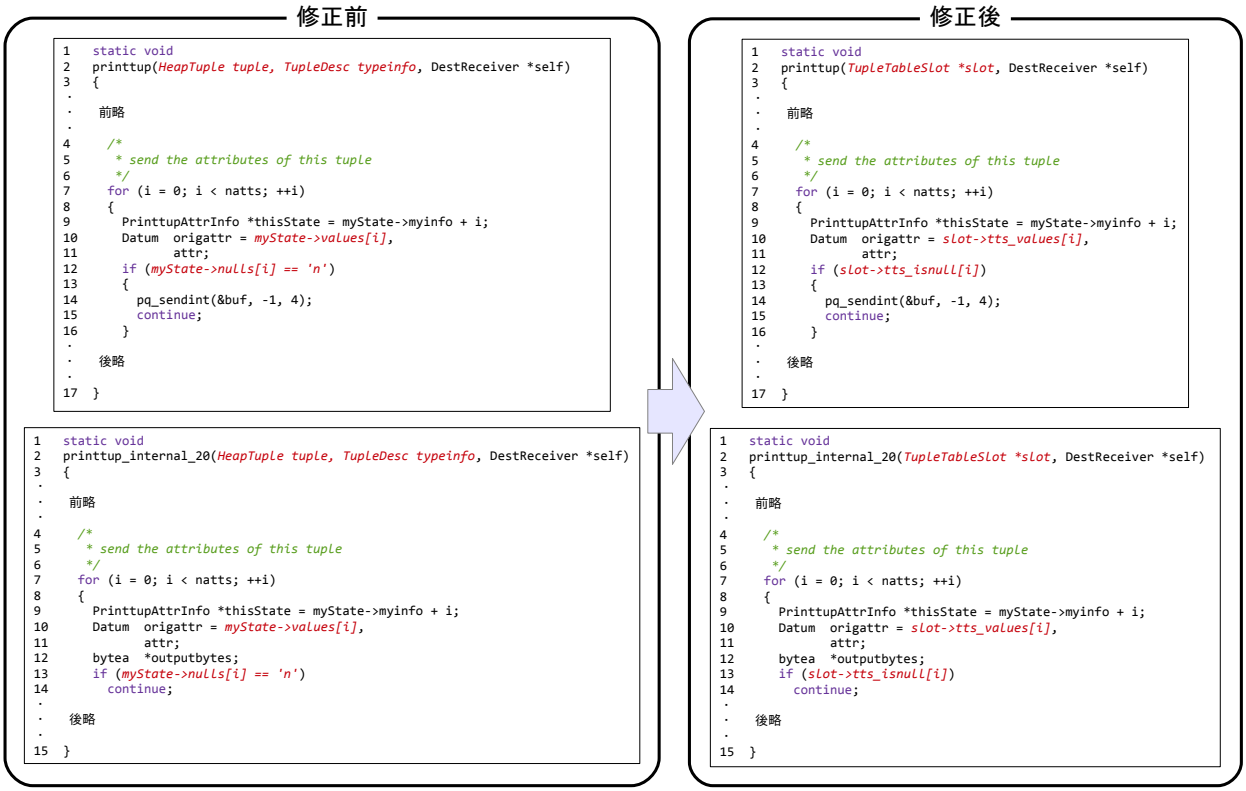

図 3　本実験における有用なクローンセットの検出事例 (タプル出力に関するクローン)
Fig. 3　Example of usefull cloneset in our evaluation (clones on printing tuples).

行われていない. 従って, この事例では, 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検出事例の 1 つであるといえる.

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, 本システムが 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, 表 2 の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, 本システムが, 図 3, 図 4 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, 本システムが CloneNotifier より有用な点を持つことは確実であると考えられる.

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

© 2015 Information Processing Society of Japan

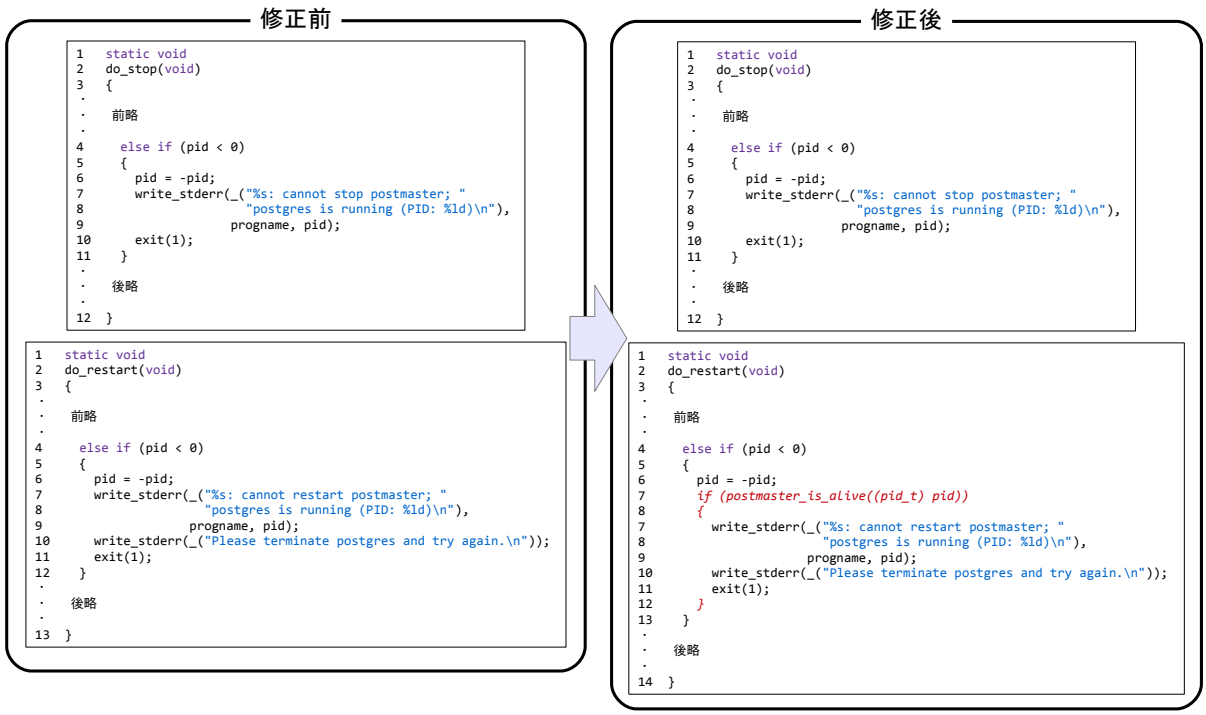

図 4　本実験における有用なクローンセットの検出事例 (データベース制御に関するクローン)
Fig. 4　Example of usefull cloneset in our evaluation (clones on database control).

プ 1〜4 の全てのコードクローンに対応した新しいコードクローン変更管理システムを開発した．また, PostgreSQL に本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．

今後の課題として, CloneNotifier との比較評価を行う必要があると考えられる．現時点では, CloneNotifier が検出できず, 開発したシステムが検出できるコードクローンを確認したのみであり, その逆に, 開発したシステムが検出できず, CloneNotifier のみが検出できるコードクローンが存在する可能性もある．また, 厳密な議論を行うためには, 定量的な評価も必要あると考えている．その他, 本システムの対応言語の拡張も必要と考えられる．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, 本システムでは, 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　本研究は JSPS 科研費 25220003，26730036 の助成を受けたものです．

参考文献

[1] Baker, B. S.: Finding Clones with Dup: Analysis of an Experiment, *IEEE Transactions on Software Engineering*, Vol. 33, No. 9, pp. 608–621 (2007).
[2] Baxter, I. D., Yahin, A., Moura, L., Sant'Anna, M. and Bier, L.: Clone detection using abstract syntax trees, pp. 368–377 (1998).
[3] 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).
[4] 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).
[5] Jiang, L., Misherghi, G., Su, Z. and Glondu, S.: DECKARD: Scalable and accurate tree-based detection of code clones.
[6] Kamiya, T., Kusumoto, S. and Inoue, K.: CCFinder: A multilinguistic token-based code clone detection system for large scale source code, *IEEE Transactions on Software Engineering*, Vol. 28, No. 1, pp. 654–670 (2002).
[7] 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).
[8] Kim, H., Jung, Y., Kim, S. and Yi, K.: MeCC: memory comparison-based clone detector, pp. 301–310 (2011).
[9] Laguë, B., Proulx, D., Mayrand, J., Merlo, E. M. and Hudepohl, J.: Assessing the Benefits of Incorporating Function Clone Detection in a Development Process, pp. 314–321 (1997).
[10] Li, Z., Lu, S., Myagmar, S. and Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code, *IEEE Transactions on Software Engineering*, Vol. 32, No. 3, pp. 176–192 (2006).

© 2015 Information Processing Society of Japan

[11] 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).
[12] Roy, C. K., Cordy, J. R. and Koschke, R.: 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).
[13] 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).
[14] 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).

© 2015 Information Processing Society of Japan