不具合修正履歴の理解に向けたソースコードの変更解析

畑　秀明[†1]　森井亮介[†1]
水野修[†2]　菊野亨[†1]

ソースコードの変更解析は，プログラム理解などの重要な技術であるが，構文変化を指摘するにとどまっている．不具合の修正前後で何を対象とした修正が行われたのかを理解するには，構文変化が何に影響するかを人目で見て判断する必要があった．本論文ではプログラム依存グラフを用いて，各データが何に使われるか，構文変化がどのデータに影響するか，データに対する処理の流れはどう変わるかといった分析を行い修正対象を分析した．修正対象として，メソッドコールに限定して，オープンソースプロジェクトの開発履歴を調査した結果，よく修正されるメソッドがあることを明らかにした．

Change Analysis Toward an Understanding of Fixing Bugs

Hideaki Hata,[†1] Ryosuke Morii,[†1] Osamu Mizuno[†2]
and Tohru Kikuno[†1]

Investigating bug fix changes that occur across several versions in software repositories has been a key issue in software analysis. We propose a PDG (Program Dependency Graph)-based source code change analysis technique. To clarify fixing targets, we conducted data-centric analysis. We analyzed which data is affected by a change and what operations the data is used. With this technique, we conducted an experiment with an open source project. Fixing number of method calls are counted and it revealed that there are project-specific method calls that induce bugs.

1. はじめに

ソフトウェア開発において，低コストで信頼性を確保するためにはツールによる支援が必要である．我々は，開発履歴中の不具合修正の情報をもとに，似たような不具合の混入を早期に防止するシステムの開発を目指している．これまでの研究では，モジュールに不具合がありそうか否かの判定を行うことを目指し，一定の成果を得た [1]．また，プロジェクトごとに不具合混入モジュールに頻発するメソッドコールや変数名などがあることを明らかにした．

発展的課題として，どこを修正すべきかという修正対象の提示を目指している．課題実現の方針として，まず，(i) 不具合修正履歴からの修正対象分析を行い，(ii) 開発中のモジュールと過去のモジュールとのパターンマッチから修正対象の候補を出力したいと考えている．本論文では (i) についてのこれまでの成果を述べる．

これまでのソースコードの変更解析は，構文の変更を指摘するにとどまっている．不具合の修正における変更解析を行った研究でも，*if* 条件の変更などがよく起こる構文変更だと明らかにしたのみである [2]．我々が目指すのは，修正対象を明らかにすることであり，例えば *if* 条件の変更においては，その条件変更がどの処理に影響するかまで分析することである．

2. 提案手法

修正対象を分析するためのキーアイディアは，どこで使われるデータの処理が修正されたかを明らかにするということである．このために，以下のことを分析する．

- 構文変化は，どのデータに影響するか
- 各データは何に使われるか（データの依存関係）
- データに対する処理の流れはどう変わるか（処理のパス，制御の依存関係）

データの依存関係，制御の依存関係を把握するため，ソースコードをメソッド単位でプログラム依存グラフ（Program Dependency Graph, PDG）として表し，分析

†1 大阪大学 大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University
†2 京都工芸繊維大学 大学院工芸科学研究科
Graduate School of Science and Technology, Kyoto Institute of Technology

```
String format( Object obj ) {
    String str = "";
    str = obj.toString( );
    return str;
}
```

(a) 修正前

```
String format( Object obj ) {
    String str = "";
    if ( obj != null ) {
        str = obj.toString( );
    }
    return str;
}
```

(b) 修正後

図 1　修正前後のソースコード
Fig. 1　Source code before and after a bug fix

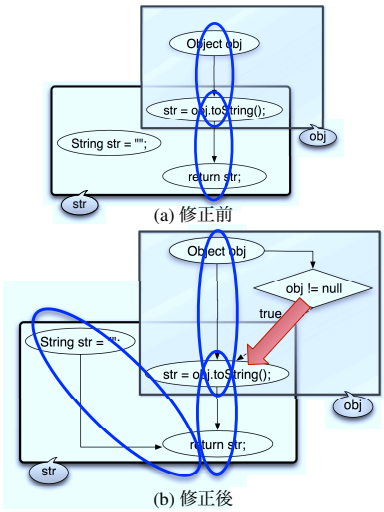

図 2　ソースコードからの PDG
Fig. 2　Constructed PDGs

を行う．修正対象を特定する分析を，図 1 に示した修正前後のソースコードを用いて説明する．これは引数 *obj* を *String* へ変換するメソッドの修正である．

図 2 は，図 1 のソースコードをそれぞれ PDG へ変換し，関係するデータごとに頂点をまとめたものである．またデータが到達するまでのパスを楕円で，制御条件を太矢印でそれぞれ示している．図 2 の差分から，影響するパスの到達点の操作を修正対象として特定する．この例では，メソッドコール *toString* と返り値を対象とした修正であったと決定する．

ただし，この手法では変更が影響する対象を全て修正対象として特定するため，リファクタリングなどの不具合修正以外も含めてしまう．そのため，修正内容のさらなる分析が必要である．

3. 不具合履歴の調査

前述したように提案手法はさらなる改善が必要である．しかし，これまで行われていない，修正対象の自動的な分析が可能である．そこで，修正対象をメソッドコールに限定して不具合履歴の調査を行った．調査対象はオープンソースプロジェクトの Eclipse Web Tools Platform で，開発言語は Java である．開発履歴から不具合修正前後のソースコードを取得し，メソッドコールを対象とする修正に関し，呼び出されるメソッドごとに修正回数を測定した．

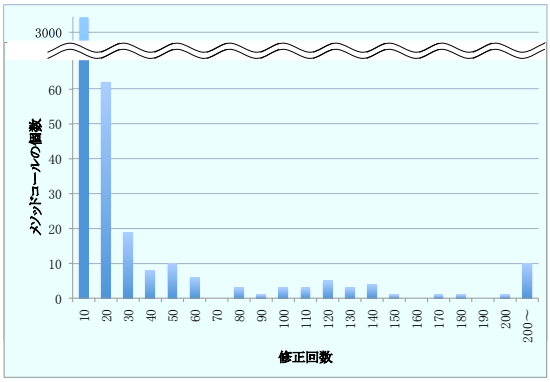

図 3　メソッドコールごとの修正回数のヒストグラム
Fig. 3　Histogram of fixing numbers for method calls

図 3 は，修正対象ごとの修正回数をヒストグラムとして描いたものである．95% 以上のメソッドコールは，修正回数が 10 以下であった．それに対し，一部のメソッドコールは何度も修正対象となっていることが分かった．本調査から，メソッドの呼び出し側において，何度も修正されているメソッドがあることが分かった．

4. おわりに

本論文では，不具合修正履歴における不具合対象を特定する手法を提案した．不具合対象をメソッドコールに限定した調査では，よく修正される典型的なメソッドコールがあることを明らかにした．今後は修正内容の分析を行い，適切な修正対象を提示する手法について検討を行う．

参考文献

1) Hata, H., Mizuno, O. and Kikuno, T.: Fault-prone module detection using large-scale text features based on spam filtering, *Empirical Software Engineering* (to appear).
2) Pan, K., Kim, S. and Whitehead, Jr., E. J.: Toward an understanding of bug fix patterns, *Empirical Software Engineering*, Vol.14, No.3, pp.286–315 (2009).