Feature Locationベンチマークの現状と課題

林　　晋　平[†1]

本稿では，Feature Location 手法評価のためのベンチマークとして改版履歴に基づくものを取り上げ，その特徴や問題点を分析しながら，ベンチマーク作成の課題について議論する．

Issues and Challenges in Feature Location Benchmarks

Shinpei Hayashi[†1]

This paper discusses issues and challenges in feature location benchmarks using an existing benchmark based on revision histories.

1. はじめに

これまでに提案されてきた様々な Feature Location (FL)[3),4)] 手法を比較・検討するため，FL 手法の評価基盤としてベンチマーク（FL ベンチマーク）の整備が望まれている．FL ベンチマークは，FL 手法の入力に用いるソフトウェアのソースコードやソフトウェアが実装している機能の一覧などに加え，FL 手法の望ましい出力（正解）として各機能の実装箇所も含んでいる．既存の著名な FL ベンチマークとして Dit らのもの[1)] が存在する．しかし，その実際の利用には課題も多い．本稿では該当 FL ベンチマークの課題を整理し，FL ベンチマークの要件を議論するための準備とする．

2. 改版履歴に基づく FL ベンチマーク

Dit らは Java で実装された 5 つのオープンソースソフトウェア（OSS）の改版履歴をもとにした FL ベンチマークを公開している[1)]．この FL ベンチマークは，各 OSS に対して，対象バージョンのソースコードおよび該当バージョンで実装されている機能の一覧を含んでいる．また，各機能に対して，機能名，機能記述，該当機能を実行したシナリオにおけるメソッド実行系列および，正解として機能を実装しているメソッド一覧が用意されている．

このベンチマークは，以下のように構築されている．まず，各 OSS で対象とするバージョンを定める．次に，該当 OSS の問題管理システムから，機能の追加要求とみなせるチケットを選定する．ここで，チケットは該当バージョンで実現済みの（該当バージョンの出荷より前に閉じられた）ものに限定する．各チケットのタイトルを機能名，報告者の記述を機能記述として抽出する．次に，各チケットが表現する機能に対応するファイル群を特定する．これは，改版履歴において，機能に対応するチケット番号をログに含んだコミットで該当の機能が実装されたとみなし，該当のコミットで変更されたファイル群（変更セット）を抽出することにより行う．最後に，各ファイルの変更前後の抽象構文木を比較することにより，追加・変更されたメソッド（コンストラクタを含む）の集合を特定し，これを機能を実装しているメソッド一覧とする．その他，動的解析に基づく FL 手法用に，該当機能を実際に起動するよう定めたシナリオを実行し，得られたメソッド実行系列を添付している．以上により得た機能群を，該当バージョンにおける機能の一覧としてまとめる．

改版履歴に基づく FL ベンチマークは，「機能の導入時に変更されたモジュールが該当機能を実装している（可能性が高い）」という仮定のもと，OSS の活動履歴から全自動で構築されている．そのため，量産可能であり，またベンチマーク作成者の主観を可能な限り排除して構築できるという特徴がある．「ある機能が実装されているモジュールはどれか」という問いは曖昧であり，該当ソフトウェアの熟練者であっても回答が分かれることもあるため，実際の修正に基づいたアプローチは客観さに優れている．

3. 課　　題

前述したように改版履歴に基づく FL ベンチマーク

†1 東京工業大学
Tokyo Institute of Technology

は有益であるが，利用する改版履歴の特徴などにより，得られたメソッド一覧が正しい機能の実装箇所としてふさわしくないものも存在する．誌面の都合上具体例を省略するが，以下に問題点の一部を列挙する．また，解決のための方策を議論する．

なお，改版履歴から導出された正解と，機能記述から想像される実装には多くの場合乖離があることも付記しておく．機能追加によりソースコードのどこが修正されるかは追加前のソースコードに強く依存するものの，機能記述は追加前のソースコードの実装からの差分のみを必ずしも表現していないからである．この問題は根が深いが，本稿では取り扱わない．

3.1 改版記録の誤り

たとえばSubversionでは，あるファイルの修正が，実際には該当ファイルの置換として記録されてしまうことがある．これを，該当ファイル中の全メソッドの変更とみなしてしまうと，該当機能に無関係のメソッドが正解に多く含まれてしまうことになる．この問題は，変更セットからのメソッド集合の抽出方法を工夫することにより解決できるかもしれない．

3.2 単純削除

機能の導入により削除されたコード片のみを含むメソッドを，該当機能を実装済みのソースコード上の分析から発見することは難しい．これは，削除されたコード片は該当機能の特徴ではなく，対象ソフトウェアの過去の実装の特徴であり，現在のソースコード上には表現されていないからである．単純削除のみに関わるメソッドを正解から除外するか，あるいはこういった修正を変更セットに含む機能をベンチマークから除外する必要があるかもしれない．

3.3 複数の意図の変更の混在

1つの変更セットで，該当の機能追加だけでなく，他の意図の変更も実施してしまっている場合，正解のメソッド集合が不要に大きくなる．典型的なものは，該当機能の追加前あるいは追加後に行われたリファクタリングである．Rajlichの示すソフトウェア変更プロセス[3]においても，FLの後に行うソフトウェア変更の前後でリファクタリングを行うことを許容している．この場合，一連の変更を分けずにコミットすれば，変更セットにリファクタリングに関連する修正が含まれることになる．

解決には変更セットからのリファクタリング影響の除去技術[5]，変更セットの分割技術などの適用が検討できるが，ベンチマークの品質に関連技術の精度が影響してしまう問題も生じる．混在した変更セットはベンチマークから除外すべきかもしれない．

3.4 後段の変更の影響

機能の実装後，該当機能の実装箇所に他の変更が行われると，得られたメソッド集合と実際の正解に差が生じてしまう．たとえば，後段の変更により機能の実装が他のメソッドに移ってしまうと，抽出されたメソッドと該当機能との関係が薄くなる．また，後段の変更でメソッド名や所属クラスの変更，メソッドの削除や他のメソッドとの統合が起こると，正解が含むメソッドがソースコード上に存在しなくなる．名前変更や移動のリファクタリングは，その前後における同一のモジュールの特定を難しくしてしまう（要素マッチング問題[2]）．

この問題は，機能の実装直後のソースコードをFLの入力として用いることにより回避することはできる．しかし，利用しやすいベンチマークの提供という観点からは，ソースコードの分析回数を抑えるためにも，単一のソースコードに複数の機能が実装され，それらを特定するという構成が望ましい．後段の修正の依存関係について分析し，誤りを除去する取り込みの導入が必要かもしれない．

4. おわりに

本稿では改版履歴に基づく既存のFLベンチマークの問題点について概観した．ベンチマーク作成において，人力での精査・修正の是非やそのコストと品質とのトレードオフ，多くのFL手法に適したベンチマークの形式などについて議論が進むことを期待する．

謝辞 議論にご参加頂いた大阪大学の石尾隆助教，日本電信電話株式会社の風戸広史博士，大島剛志氏に感謝する．

参考文献

1) Dit, B., Revelle, M., Gethers, M. and Poshyvanyk, D.: Feature Location in Source Code: A Taxonomy and Survey, *J. Softw. Maint. Evol.: Res. Pract.* (2011). DOI: 10.1002/smr.567.
2) Kim, M. and Notkin, D.: Program Element Matching for Multi-Version Program Analyses, *Proc. MSR*, pp.58–64 (2006).
3) Rajlich, V.: *Software Engineering: The Current Practices*, CRC Press (2012).
4) Rajlich, V. and Wilde, N.: The Role of Concepts in Program Comprehension, *Proc. IWPC*, pp.271–278 (2002).
5) Thangthumachit, S., Hayashi, S. and Saeki, M.: Understanding Source Code Differences by Separating Refactoring Effects, *Proc. APSEC*, pp.339–347 (2011).