行単位の依存関係を用いたテスト選択手法の提案

藤原 勇真[1]　神田 哲也[1]　嶋利 一真[2]　肥後 芳樹[1]

概要：テスト実行のコストを削減するための手法の１つとして，テストケース選択が活用されている．既存研究では，ファイルやメソッドレベルでプロダクトコードとテストの依存関係を取得し，テスト選択を行うことで，テスト実行のコストを大幅に削減している．しかし，メソッド単位で依存関係を取得した場合でも，分岐等によって実際に実行されない命令が存在する場合があり，不要なテストケースが含まれている可能性がある．そこで本研究では，プロダクトコードの行とテストの依存関係を利用したテストケース選択手法を提案する．

1. はじめに

回帰テストは，ソフトウェアの変更によって既存機能の動作に影響がないかを確認するために重要である．変更のたびに全てのテストを再実行することはコストが大きいため，障害分析に影響が出ないように実行するテストケースの数を削減するテスト選択技術が活用されている．Gligoric らは，ファイルレベルのテストケースとプロダクトコードの依存関係を利用し，変更されたファイルに関連するテストを選択する動的テスト選択技術，Ekstazi [1] を提案し，エンドツーエンド時間を 32 %短縮することを示している．テスト選択にファイルやメソッドレベルの依存関係を用いた場合，分岐等によって実際に実行されない命令が存在する場合があり，不要なテストが含まれている可能性がある．細かい粒度での依存関係を用いることで解析時間が増大するが，選択するテスト数の削減が期待できる．

本研究では，テストケースとプロダクトコードの行単位での依存関係と変更時のソースコード差分を用いることで，ソースコードの更新時に実行されるテストを選択する手法を提案する．提案手法と既存手法を比較し，細かい粒度依存関係を用いたテスト選択手法の有用性を評価する．

2. 提案手法

本研究では，既存手法でテストケースに対する依存関係として用いられているファイルやメソッドレベルより細かいソースコードの行単位を用いたテスト選択手法を提案する．本研究において，テストケース T とプロダクトコードのある行 L に依存関係があるとは，あるテスト実行 T において，プロダクトコードのある行 L が実行されていることを意味する．提案手法の全体像を図 1 に示す．

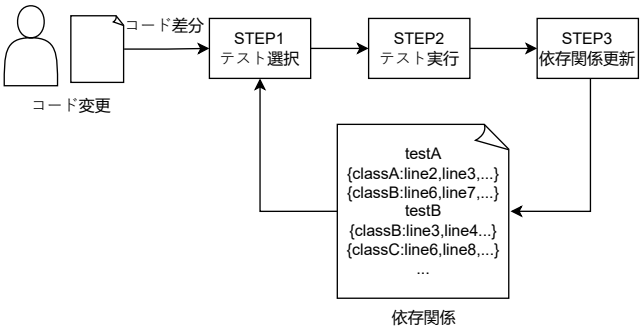

図 1　提案手法全体図

STEP1 では，コードの差分と既存の依存関係から，テストケースを選択する．メソッド内にコードの追加があった場合，追加されたコードの直前行と依存関係のあるテストケースを選択する．メソッド内にコードの削除があった場合，削除されたコードと依存関係のあるテストケースを選択する．フィールドに変更があった場合，該当のフィールドと依存関係のあるテストケースを選択する．新たにファイル単位での追加があった場合，全テストを選択する．なお，既存の依存関係が存在しない場合は全てのテストを実行し，依存関係を構築する．STEP2 では，選択したテストケースを実行し，実行時の情報を取得する．本研究では，SELogger [2] を利用し，テスト単位で実行時の命令や行番号を取得する．STEP3 では，STEP2 で得た実行時情報及びコードの差分を用いて，依存関係を更新する．テストケースとプロダクトコードの依存関係については，テスト単位で実行するクラス名及び行番号を管理する．フィールド変数の依存関係については，フィールド変数名単位で同

[1] 大阪大学
Osaka University
[2] 奈良先端科学技術大学院大学
Nara Institute of Science and Technology

一のフィールド変数を参照するクラス名及び行番号を管理する．また，コードの差分からテストケースとプロダクトコードの依存関係内の行番号を更新する．これは，import 文の削除による行番号のずれのような，テストケースに無関係なコードの変更が行われた場合を考慮するためである．

3. 評価

提案手法の有用性を Ekstazi と比較し評価する．これは，Shin らが既存のテスト選択技術を比較評価した結果，Ekstazi が最も優れていると結論づけているためである [3]．

3.1 評価手法

既存研究と同様に，OSS の各リビジョンに対してテスト選択技術を適用し，性能を評価する．初回のリビジョンに対しては，既存の依存関係が存在しないため，全テストを実行し，依存関係を構築する．2 番目に古いリビジョン以降から，テスト選択が行われる．実験対象の OSS を表 1 に示す．本研究では，テスト削減率と実行時間削減率の側面から評価を行う．テスト削減率は，テスト選択を利用しない場合，つまり全てのテストケースを実行した場合に対して，テスト選択を適用した場合に，実行するテストをどの程度削減できたかを示す指標である．実行時間の削減率は，全てのテストケースを実行する場合に対して，テスト選択を適用した場合にテスト実行及びテスト選択が費やす時間の合計をどの程度削減できたかを示す指標である．Ekstazi の場合，ビルドコマンドを実行することで，テスト選択からテスト実行，依存関係の更新まで行われるため，ビルドコマンドの実行時間をエンドツーエンド時間として計測する．提案手法の場合，コードの差分の取得，テスト選択，テスト実行，依存関係の更新，はそれぞれ手動で行う必要があるため，各フェーズの実行時間を計算し，合計することで，実行時間を計測する．選択されたテストケースがない場合，Ekstazi と同様にテスト実行はスキップし，ビルドのみ行う．人為的なミスを避け，再現性を確保するために，実験全体を自動化して行った．

3.2 結果

結果を表 2 に示す．テスト数削減率に関しては，3 つのプロジェクトで Ekstazi を上回ったが，2 つのプロジェクトでは下回った．これは，提案手法ではインデントの修正やコメントの編集をファイルの変更有と判断している一方，Ekstazi で変更無と判断しており，この点で提案手法が多くのテストケースを選択しているからである．Ekstazi ではクラスファイルのチェックサムを用いてファイルの変更有無を判断しているため，インデントの変更やコメントの編集を無視できるが，提案手法ではコードの差分を利用しているため，無視できない．

実行時間削減率に関しては，3 つのプロジェクトでは，

表 1 実験対象のプロジェクト一覧

プロジェクト	リビジョン数	テストクラス数	テストメソッド数	コード行数
commons-cli	100	29	438	16,932
commons-jxpath	99	61	411	48,491
jacksonXML	33	140	348	26,639
NuProcess	51	11	34	9,899
commons-net	80	63	316	65,791

表 2 結果

プロジェクト	実行時間 [s]	テスト数削減率 [%]		実行時間削減率 [%]	
		提案手法	Ekstazi	提案手法	Ekstazi
commons-cli	8.75	**98.00**	94.02	**6.40**	3.94
commons-jxpath	20.36	**97.07**	96.85	1.92	23.37
jacksonXML	24.99	83.05	83.32	**25.60**	13.70
NuProcess	78.10	**96.16**	80.70	**91.15**	76.02
commons-net	109.99	93.42	93.47	69.8	80.70

Ekstazi を上回ったが，その他の 2 つに関しては，Ekstazi を下回った．特に，commons-jxpath では，Ekstazi を大きく下回った．commons-jxpath は依存関係が非常に複雑であり，テストケース単位で実行したクラス名及び行番号を記録した依存関係ファイルのサイズが，commons-net は 852KB に対し，commons-jxpath は 92MB もあった．依存関係が複雑である場合，依存関係の解析や更新に時間がかかってしまい，実行時間が大きくなってしまう．反対に，依存関係が単純で，テスト実行時間が長い NuProcess に関しては，細かい依存関係を利用することで，実行するテストケース数が削減でき，実行時間の削減が達成できた．jacksonXML に関しては，選択されたテストケースがない場合に，提案手法の方が Ekstazi よりもビルド時間が短くなったため，実行時間削減率が高い値になっている．

4. まとめ

プロダクトコード行単位とテストケースの依存関係を利用したテスト選択手法を提案し，依存関係が単純でテスト実行時間が長いプロジェクトにおいて，提案手法の有用性を確認した．今後は，差分や依存関係の解析方法の洗練による実行時間の削減や，プロジェクトの規模や変更内容に応じた適切な依存関係の粒度選択を検討していく．

謝辞 本研究は栢森情報科学振興財団，JSPS 科研費（JP19K20239, JP20H04166，JP21K18302，JP21K11829，JP21H04877，JP22H03567，JP22K11985）の助成を受けたものです．

参考文献

[1] Gligoric, M., Eloussi, L. and Marinov, D.: Practical regression test selection with dynamic file dependencies, *Proc. ISSTA 2015*, pp. 211–222 (2015).

[2] Shimari, K., Ishio, T., Kanda, T. and Inoue, K.: Near-omniscient debugging for java using size-limited execution trace, *Proc. ICSME 2019*, pp. 398–401 (2019).

[3] Shin, M. K., Ghosh, S. and Vijayasarathy, L. R.: An empirical comparison of four Java-based regression test selection techniques, *J. Syst. Softw.*, Vol. 186 (2022).