修士学位論文

題目

テスト-コード間の行単位の動的な依存関係に基づくテスト選択手法

指導教員
肥後 芳樹 教授

報告者
藤原 勇真

令和 6 年 2 月 1 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

令和５年度 修士学位論文

テスト-コード間の行単位の動的な依存関係に基づくテスト選択手法

藤原 勇真

内容梗概

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．しかし，近年，ソフトウェアは大規模・複雑化しており，実行されるソフトウェアテストの量や頻度も大きく，テスト実行には膨大な時間を要する．

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ソース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，テストとソースコード間の行単位の動的な依存関係を用いたテストケース選択手法を提案する．本手法により，実行経路を考慮でき，不要なテストを削減することができる．

評価実験では，テスト数削減率やエンドツーエンド時間削減率の観点から，提案手法を既存手法と比較することで，有用性を調査した．その結果，一部のプロジェクトでは既存手法を上回り，有用性を確認できた．適合率の評価を行ったところ，いくつかの変更パターンには対応できていないものの，基本的な変更パターンには対応できていることが分かった．また，プロジェクトの特徴と提案手法の有用性の関連性について調査を行ったところ，提案手法が有用に働くプロジェクトの特徴は見つからなかったが，バイトコード命令の実行回数が多いほど，提案手法が有用に働きにくい傾向が分かった．

主な用語

テスト選択
実行トレース
動的解析

目 次

1　はじめに

ソフトウェアテストは，ソフトウェア製品，アプリケーションが想定どおりに機能することを評価および検証するプロセスである．ソフトウェアテストは，バグの防止や開発コストの削減，パフォーマンスの向上の実現に寄与するため，ソフトウェア開発において重要なプロセスである．ソフトウェアテストには，システム全体を検証する受け入れテストや，システムがどれくらいの負荷に耐えられるかを検証するストレステストなど，様々なテストがある．

ソフトウェアテストの１つである回帰テストは，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．しかし近年，ソフトウェアは大規模，複雑化しており，実行されるソフトウェアテストの量や頻度も大きく，テスト実行には膨大な時間を要する [6]．例えば，Google の TAP は，平均して 1 日に 1 万 3000 行以上のコードプロジェクトを統合し，80 万回のビルドと 1 億 5000 万回のテスト実行を必要としている [17]．また，既存の研究によると，ソフトウェア開発の全体のテスト時間において，回帰テストの時間は約 80%を占めている [4]．

ソフトウェアの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，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，テスト選択技術が活用されている．テスト選択とは，障害分析に影響が出ないように，テストケースの数を削減することで，テスト実行のコストを削減する手法である．テスト選択には，類似したテストケースの片方のみを選択して実行する手法や，ソフトウェアの変更に関連するテストケースのみを選択して実行する手法など，様々な手法がある．ソフトウェアの変更に関連するテストケースのみを選択する手法として，ファイルレベルやメソッドレベルの依存関係を利用し，変更されたファイルやメソッドに関連するテストケースを選択する手法がいくつか提案されている [1,5,11,12]．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，障害分析に影響が出ない不要なテストが含まれている可能性がある．ファイルやメソッドレベルよりも細かい粒度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．

そこで，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の依存関係を用いたテスト選択手法を提案する．本手法により，既存の手法に比べてより多くのテストケースを削減でき，テスト実行時間を削減できる可能性がある．

提案手法の有用性を調査するため，テスト数削減率とエンドツーエンド時間削減率の観点から，既存手法と比較評価を行った．その結果，評価実験対象の全 7 プロジェクトにおいて，提案手法のテスト数削減率が既存手法を上回り，7 プロジェクト中 4 プロジェクトにおいて，

提案手法のエンドツーエンド時間削減率が既存手法を上回った．提案手法の適合率についても調査を行った．その結果，いくつかの変更パターンの際に適切なテストケースを選択できていなかったが，よくある変更パターンについては基本的に対応できていた．また，提案手法の有用性とプロジェクトの特徴の関連性の調査を行った．その結果，提案手法が有用に働くようなプロジェクトの特徴は得られなかったが，バイトコード命令の実行回数が多いプロジェクトに対しては，提案手法が有用に働かない傾向にあることが分かった．

以降 2 章では，背景について述べる． 3 章では提案手法について述べ， 4 章ではその評価実験について述べる． 5 章では，妥当性の脅威について述べる．最後に 6 章では本研究のまとめと今後の課題を述べる．

2　背景

2.1　回帰テスト

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，テストを実行して検証する作業であり，重要な品質保証手法である．Hetzel らの調査によると，プログラムの修正中にエラーが発生する確率は 50%から 80%であり [10]，プログラムの修正の過程において回帰テストの実施が不可欠であることを示している．

近年，ソフトウェアは大規模・複雑化しており，実行されるソフトウェアテストの量や頻度も大きく，テスト実行には膨大な時間を要する [6]．また，ソフトウェア業界では，従来よりもはるかに頻繁で迅速なペースでソフトウェアの変更がリリースされ，アジャイルで継続的なデリバリーが進んでいると指摘されている [16, 19]．例えば，Google の TAP は，平均して 1 日に 1 万 3000 行以上のコードプロジェクトを統合し，80 万回のビルドと 1 億 5000 万回のテスト実行を必要としている [17]．また，既存の研究によると，ソフトウェア開発の全体のテスト時間において，回帰テストの時間は約 80%を占めている [4]．このように，開発者は回帰テストのコストに悩まされている．全てのテストを実行すると回帰テストのコストが膨大になるため，今日では，回帰テストのコストを削減するために多くの研究者がテスト選択やテスト優先度付け，テスト最小化について研究を行っている [24]．

2.2　テスト選択

テスト選択とは，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手法である．テスト選択には，類似したテストケースの片方のみを選択して実行する手法やソフトウェアの変更に関連するテストケースのみを選択して実行する手法などがある．

2.2.1　類似したテストの片方を選択する手法

テストケースの中には類似したものが含まれるため，回帰テストのようなテストにおいては，類似したテストを取り除くことで効率的に故障検出を行うことができると考えられる．Chen らは，半教師付きクラスタリングを用いて，類似したテストの片方を選択する手法を提案し，手法の有用性を示している [2]．Mondal らは，テストケースの多様性を最大化するようなテスト選択手法を提案し，高い故障検出率を示している [18]. 嶋利らは，バイトコード命令単位での実行頻度をもとに大量のテストから小規模なテストケースを抽出する方法を提案し，実企業のシステムにおいて有用性を検証している [22]．これらの手法は，類似したテストケースを省くなどして効率的なテストのサブセットを構築する手法である．

2.2.2　変更に関連するテストを選択する手法

回帰テストにおいては，小さなソフトウェアの変更毎にテストを実行するため，変更に関連するテストのみを実行することで，効率的に故障検出を行うことができると考えられる．ソフトウェアの変更に関連するテストケースを選択する手法として，Gligoric らは，ファイルレベル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ケースを選択する動的テスト選択技術，Ekstazi [5] を提案している．他にも，クラスレベルの分析を活用する静的テスト選択技術，STARTS [12] や, クラスレベルとメソッドレベルを組み合わせた動的テスト選択技術，HyRTS [11], ファイルレベルの分析を活用し，実行コードのみを考慮する動的テスト選択技術，OpenClover [1] などが提案されており，Shin らは，これらのテスト選択技術の性能を評価している [23]．Shin らは，上記 4 つのテスト選択技術の性能を，エンドツーエンド時間，選択したテストケース数，安全性，精度，欠陥検出能力の 5 つの観点から評価している [23]．結論として，Ekstazi が最も優れたパフォーマンスを発揮し，特にプログラムのサイズが 100KLOC を超える場合において優れていた．

Ekstazi について詳しく説明する．Ekstazi [5] は，ファイルレベルの依存関係を利用する，動的テスト選択技術である．Ekstazi には，分析フェーズ，実行フェーズ，収集フェーズの 3 つのフェーズがある．分析フェーズでは，変更されたファイルを特定し，変更されたファイルに関連するテストケースを選択する．具体的には，まず，変更後の各ファイルのチェックサムを計算し，変更前の各ファイルのチェックサムと比較する．変更前後でファイルのチェックサムが異なる場合，そのファイルは変更されたと判断される．次に，ファイルとテストケースの依存関係を用いて，変更されたファイルと依存関係のあるテストケースを選択する．実行フェーズでは，ビルドシステムからテストの実行を開始し，分析フェーズで選択されたテストケースを実行する．収集フェーズでは，各テストケースとファイルの依存関係を収集する．具体的には，テストとテスト対象のコードの実行を監視し，各エンティティの実行中にアクセスされるファイルのセットを収集し，各ファイルのチェックサムを計算し，対応する依存ファイルに記録する．Gligoric らは，Ekstazi を使用することで，エンドツーエンド時間を平均 32 %短縮することを示している．

2.2.3　変更に関連したテスト選択の例

テスト選択の例を，図 1, 図 2 を用いて示す．図 1 はプロダクトコード及びその編集を表しており，赤で色付けした箇所が緑で色付けした箇所に変更されている．図 2 は図 1 のプロダクトコードに対するテストコードを示している．テストクラス T1,T2 では，クラス A のメソッド m1 を呼び出し，テストクラス T3 では，クラス A のメソッド m2 を呼び出し，テストクラス T4 では，クラス B のメソッド m3 を呼び出している．ここで，コードの変更と

関連のあるテストケースを選択するために，ファイルレベルおよびメソッドレベルの依存関係を考える．ファイルレベルの依存関係を考えると，テストクラス T1，T2，T3 がクラス A に依存し，テストクラス T4 がクラス B に依存していることになる．ファイルレベルの依存関係を用いる Ekstazi を今回の編集に適用した場合，図 1 ではクラス A に編集が加えられているため，クラス A と依存関係のあるテストクラス T1，T2，T3 が選択される．メソッドレベルの依存関係を考えると，テストクラス T1，T2 がメソッド m1，テストクラス T3 がメソッド m2，テストクラス T4 がメソッド m3 に依存していることがわかる．従って，クラスレベルとメソッドレベルの依存関係を組み合わせて用いている HyRTS では，図 1 ではメソッド m1 に編集があると判別し，メソッド m1 と依存関係のあるテストクラス T1，T2 を選択する．

しかし，図 1，図 2 の例の場合，条件分岐によって実行されない命令も存在しており，ファイルレベルやメソッドレベルでのソースコードとテストケースの依存関係を用いた場合に不要なテストケースが含まれている．実際に編集箇所を通過するテストはテストクラス T2 のみであるため，本来必要なテストは T2 のみである．しかし，ファイルレベル，メソッドレベルでの依存関係を用いた場合，不要なテストクラス T1 及び T3 を選択している．より細かい解析を行うことで，実行経路を考慮することができ，不要なテストを削減することができると考えられる．不要なテストを削減することで，テスト選択の性能を向上させる可能性がある．

2.3 テスト優先度付け

テスト優先度付けとは，欠陥の早期検出などのテストの目的のために，テストの実行順序をスケジューリングする技術である [20]．テスト実行順序のスケジューリングの基準は様々である．例えば，実行される条件分岐の数が基準の場合，条件分岐の数が多い順にテストをスケジューリングし，実行する．カバレッジが基準の場合，未だカバーされていない箇所を通過するようなテストを次に実行するようにスケジューリングし，全体のカバレッジが徐々に向上するようにテストを実行する．一般的なテスト優先度付けでは，スケジューリングしたテストを逐次的に実行する [7, 14, 15]．最近では，スケジューリングしたテストを並列化して実行する，並列テスト優先度付け [25] が提案されている．

テスト優先度付けでは，欠陥の早期検出などの目的のためにスケジューリングしたテストを全て実行するものであり，実行するテスト数を削減し，テスト実行コストの削減が目的であるテスト選択とは異なる．

```
1  class A{
2      public int m1(int x, int y)
           {
3          if(x > y){
4              return x - y ;
5          }else{
6              return x + y;
7              return y - x;
8          }
9      }
10     public int m2(int x, int y)
           {
11         return x / y;
12     }
13 }
14 class B{
15     public int m3(int x, int y)
           {
16         return x * y ;
17     }
18 }
```

図 1: プロダクトコード例

```
1  class T1 {
2      @Test  public void t1() {
3          assertEquals(2, new A()
               .m1(5,3));
4      }
5  }
6  class T2 {
7      @Test public void t2() {
8          assertEquals(2, new A()
               .m1(3,5));
9      }
10 }
11 class T3 {
12     @Test public void t3(){
13         assertEquals(2, new A()
               .m2(6,3));
14     }
15 }
16 class T4 {
17     @Test public void t4() {
18         assertEquals(6, new B()
               .m3(2,3));
19     }
20 }
```

図 2: テストコード例

2.4 テストスイート最小化

テストスイート最小化とは，冗長なテストケースを特定し，それらをテストスイートから削除して，テストスイートのサイズを減少させる技術である [8]．最小のテストスイートを見つけることは NP 完全であり，ヒューリスティックな手法がいくつか提案されている [3,9]．

テストスイート最小化は，テスト設計時にテストスイートのサイズを減少させ，永続的に回帰テストのコスト削減を行っているものであり，ソフトウェアの変更の度に回帰テストのコスト削減を試みるテスト選択とは異なる．

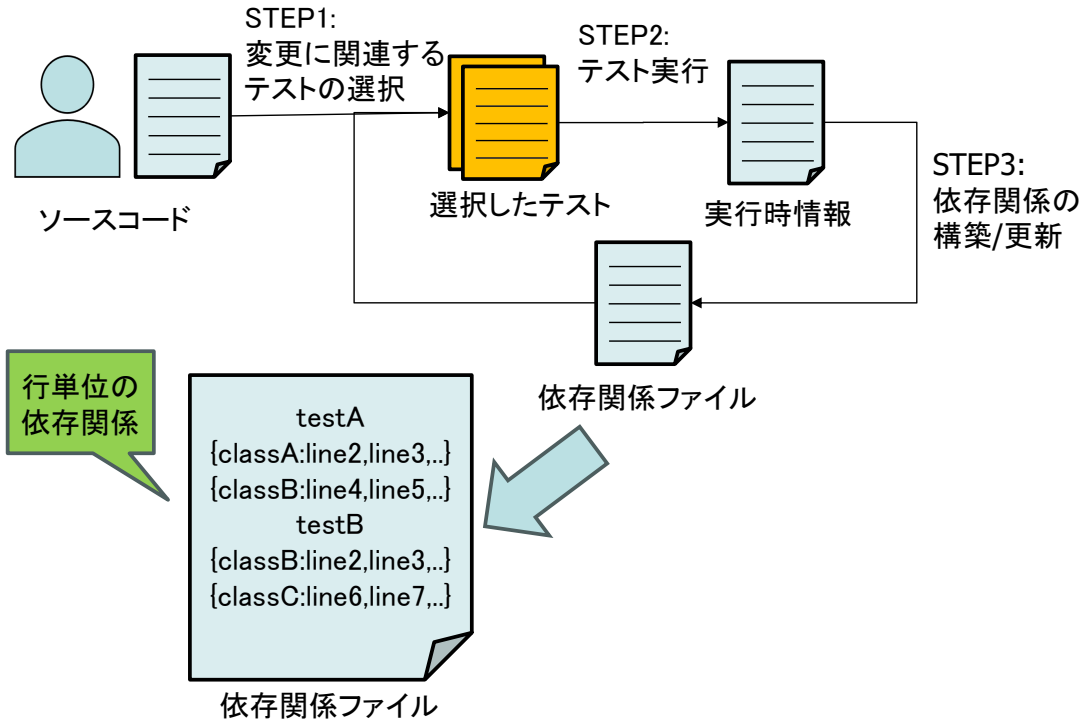

図 3: 提案手法の概要

3 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の依存関係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．また，テストケース T とフィールド変数 F に依存関係があるとは，あるテスト実行 T において，フィールド変数 F が参照されていることを意味する．提案手法の全体像について図 3 に示す．

提案手法の全体の流れは，既存手法の Ekstazi [5] と同様であり，図 3 に示す 3 つの STEP から成る．STEP1 では，コードの変更箇所を特定し，既存の依存関係を用いて，変更に関連するテストケースを選択する．STEP2 では，選択したテストケースを実行するとともに，実行時の情報を取得する．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．以下，各 STEP の詳細について述べる．

3.1 STEP1: 変更に関連するテストの選択

ソースコードに変更があった場合，ソースコードのどの箇所に変更があったかを特定し，変更箇所の動作を検査するテストを選択する必要がある．従って STEP1 では，ソースコードの差分から，変更箇所を特定し，テストケースとソースコードの依存関係を用いて，ソースコードの変更に関連するテストケースを選択する．変更箇所の特定と関連するテストの選択についてそれぞれ説明する．

3.1.1 変更箇所の特定

まず，コードの差分を受け取り，変更のあったソースファイルを特定する．今回は，git diff を用いてコードの差分を取得し，入力として与える．次に，変更のあったソースファイルに対し，字句解析を行い，トークン列に変換する．変更前のソースファイルのトークン列と，変更後のソースファイルのトークン列を比較することで，変更箇所を特定する．字句解析を行い，トークン列で比較する理由は，インデントの変更やコメントの編集の変更などのプログラムの実行に影響のない変更を無視するためである．コードの差分をそのまま利用すると，インデントの変更やコメントの編集なども，コードの変更として認識され，障害分析に必要のないテストを選択してしまう．字句解析を行い，トークン列で差分をとることで，上記の問題を解決することができる．

初回の実行に関しては，コードの差分は存在しないため，全てのソースファイルに対して字句解析を行い，各ソースファイルのトークン列をファイルに記録する．２回目以降は，変更のあったソースファイルに対して字句解析を行い，変更のあったソースファイルのトークン列を更新する．

3.1.2 関連するテストの選択

次に，変更箇所に関連するテストを，既存のテストケースとソースコードの依存関係から選択する．Ekstazi では，テストクラス単位でテストを選択するが，提案手法では，テストメソッド単位でテストを選択する．プロダクトコードの変更を，メソッド内の追加，メソッド内の修正/削除，フィールド変数の変更と分類し，それぞれ下記の通り，異なる方法でテストを選択する．

メソッド内の追加

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の実行に影響を与える恐れがあるため，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．

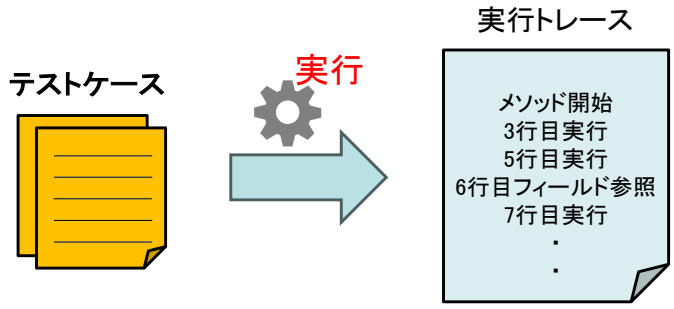

図 4: 実行トレース収集のイメージ

メソッド内の修正/削除

修正/削除されたコードと依存関係のあるテストケースの実行に影響を与える恐れがあるため，修正/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．

フィールド変数の変更

変更されたフィールドと依存関係のあるテストケースの実行に影響を与える恐れがあるため，変更されたフィールドと依存関係のあるテストケースを選択する．

テストコードに変更があった場合は，変更箇所と依存関係のあるテストケース及び変更のあったテストクラスを選択する．テストクラスの追加があった場合は，新たなテストクラスとプロダクトコードの依存関係を構築する必要があるため，追加があったテストクラスを選択する．既存の依存関係が存在しない場合，つまり初回の実行に関しては，全てのテストケースを選択する．

3.2 STEP2: 実行トレースの収集

テストケースとソースコードの依存関係を構築するために，テスト実行時の情報が必要であるため，SELogger [21] を有効にして，STEP1 で選択したテストケースを実行する．SELogger とは，Omniscient Debugging [13] の実装の 1 つであり，プログラムの実行開始から実行終了までに実行された全てのバイトコード命令と，それによる値の変化を実行トレースとして記録する．SELogger は Java 仮想マシン内で Java エージェントとして動作し，プログラムに組み込み実行することで，実行トレースを収集できる [21]．SELogger のオプションをいくつか設定し，選択したテストを実行することで，図 4 のように，テストメソッド単位で実行した行番号及びフィールドアクセス情報を収集する．

3.2.1 SELogger のオプション設定

SELogger で設定可能なオプションと，提案手法に必要な最小限の情報を取得するための設定について詳しく説明する．

実行トレースのデータ形式を指定する．デフォルトでは，各バイトコード位置のタイムスタンプとスレッド ID を含む最新 k 回（k はユーザーが指定可能，デフォルトは k=32）のイベントデータを記録する．ほかに，すべてのイベントデータを記録するモードや，バイトコード命令の実行された場所とその頻度だけを記録する軽量なモードがある．本手法では，実行トレースの収集を高速化するために，バイトコード命令の実行された場所とその頻度だけを記録する最も軽量なモードで実行する．

記録するイベントデータを指定する．記録可能なイベントは，メソッドの実行，メソッド呼び出し，フィールドアクセス，配列アクセス，synchronized ブロック，オブジェクト操作，ローカル変数，制御フローイベント，行番号イベント，その他のイベント（例外オブジェクトのキャッチ）であり，デフォルトではこれらの全てのイベントを記録する．本手法では，実行トレースの収集を高速化するために，提案手法に必要な最小限のイベントを記録するように設定する．具体的には，メソッドの実行，行番号イベント，フィールドアクセスの 3 つのイベントのみを指定し，記録する．メソッドの実行は，テストメソッドの実行単位でイベントを記録するために必要である．行番号イベントは，実行されたソースコードの箇所を特定するために必要であり，フィールドアクセスは，フィールド変数へのアクセスを特定するために必要である．

特定の間隔毎に実行トレースを記録するように指定する．SELogger では，クラス名やメソッド名，命令のイベントタイプを指定することで，特定の間隔毎に実行トレースを記録することができる．本手法では，テストメソッド単位での実行トレースが必要なため，全てのテストメソッド名及びイベントタイプとしてメソッドの開始/終了を指定し，各テストメソッドの実行開始から実行終了の間隔毎に実行トレースを収集する．

特定のユーティリティ，ライブラリを記録対象から除外するように設定する．提案手法では，変更されうるソースコードの情報が必要であるため，外部ライブラリの行番号等の情報は不要である．従って，実行トレースの記録対象から外部ライブラリを除外することで，実行トレースの収集の高速化を図る．

3.3 STEP3: 依存関係の構築・更新

ソースコードの差分からテストを選択するためには，変更前のソースコードとテストケースの依存関係が必要となるため，STEP3 では，依存関係の構築，更新を行う．初回の実行では，テストケースとソースコードの依存関係及び，フィールド変数の依存関係は存在しな

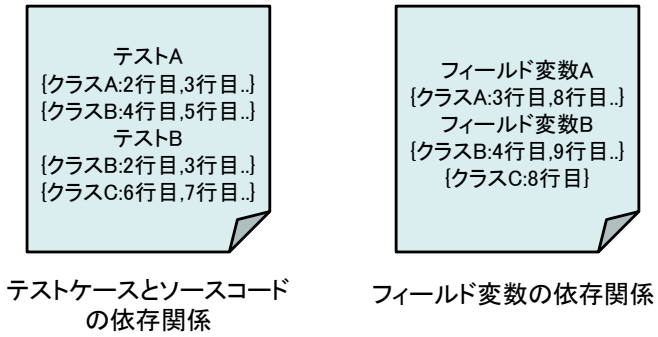

図 5: 依存関係のイメージ

いため，各依存関係を一から構築する．2 回目以降の実行では，STEP2 で取得した実行トレースを用いて，各依存関係を部分的に更新する．

まず，ソースコードの差分から STEP1 で更新されなかったテストケースとソースコードの依存関係の行番号を更新する．STEP1 で選択されたテストに関しては，依存関係自体の更新により行番号が更新される．しかし，変更内容によっては関連するテストケースが存在しない場合が存在するため，行番号の更新をここで別途行う必要がある．例えば、不要な import 文の書かれた 1 行を削除した場合などがこれにあたる，この変更に関連するテストケースが選択されないため，依存関係の更新が行われない．しかし，削除された行以降の行番号は 1 ずれるため，行番号を更新しなければ，今後のテストケース選択に影響を及ぼす．このように，テストケースが選択されないソースコードの変更が存在するため，ソースコードの差分からテストケースとソースコードの依存関係内の行番号を更新する．

次に，STEP2 で取得した実行トレースを用いて，テストケースとソースコードの依存関係及び，フィールド変数の依存関係の構築・更新を行う．依存関係のイメージを図 5 に示す．テストケースとソースコードの依存関係については，テストケース単位で，実行するクラス名及び行番号を記録する．フィールド変数の依存関係については，フィールド変数単位で，同一のフィールド変数を参照するクラス名及び行番号を記録する．

4　評価

提案手法の有用性を評価するため，以下の 3 つの RQ に答える．

- RQ1: 提案手法は既存手法 Ekstazi を上回るか？
- RQ2: 提案手法の適合率はどの程度か？
- RQ3: 提案手法が有用に働くプロジェクトの特徴は何か？

RQ1 では，提案手法の性能を既存手法と比較し，有用性を評価する．Shin らは最新のテスト選択技術を比較しており，Ekstazi が最も優れていると結論づけている [23] ため，Ekstazi と比較評価を行う．RQ2 では，必要なテストが提案手法で選択できているかを明らかにするため，適合率の調査を行う．RQ3 では，提案手法が有用に働くプロジェクトの特徴を明らかにするために，プロジェクトの静的，動的なメトリクスを分析する．

4.1　実験環境

本実験は，4core 2.80 GHz Intel(R) Xeon(R) CPU，32GB RAM のマシンを用いて，Ubuntu Linux 22.04 LTS の OS 上で行った．利用した Java の環境は，OpenJDK 64-Bit Server, バージョン 17.0.6 である．本研究で使用した SELogger はバージョン 0.5.1，Ekstazi(Java Agent) はバージョン 5.3.0 である．

4.2　評価対象プロジェクト

評価で用いたプロジェクトのリストを表 1 に示す．2 章で紹介した既存研究 [1,5,11,12,23] の評価実験で用いられている OSS を評価対象の候補として選択した．加えて，ビルドシステムとして Maven を利用，スター数 50 以上，コミット数 100 以上である，GitHub 上の OSS を評価対象の候補として選択した．選択した合計約 200 個の評価対象候補の OSS に対し，提案手法及び Ekstazi を適用した．最終的に，提案手法及び Ekstazi を適用し，双方ともビルド及び全テストに成功した OSS7 個を対象のプロジェクトとして選定した．提案手法は単一プロジェクトのみに対応しているため，jitWatch はサブプロジェクトの内，core モジュールのみを対象とした．対象としたリビジョンは，java ファイルに変更のあったリビジョンのうち，各手法を適用し，ビルド及びテストに成功した，最大 50 リビジョンである．

各プロジェクトにおいて，対象となった最も古いリビジョン ID，対象となったリビジョン数，プロダクトコードのクラス数，テストクラス数，テスト数，論理 LOC を表 1 に示す．各コードメトリクスは，対象としたリビジョンのうち，最古のリビジョンでのメトリクスで

表 1: 評価対象プロジェクトとコードメトリクス

プロジェクト	最古のリビジョン ID	リビジョン数	クラス数	テストクラス数	テスト数	論理 LOC
commons-cli	591af95	50	26	34	432	6,203
hilbertCurve	1d7c9f8	24	14	37	59	8,283
commons-jxpath	42da558	33	250	82	411	24,927
jacksonXML	b311677	22	56	451	334	15,638
jitWatch(core)	f28deaa	13	123	43	251	58,939
NuProcess	7dadfb1	20	31	23	22	4,965
commons-net	0827578	50	155	62	309	28,305

ある．各クラス数には，インナークラスも含まれる．論理 LOC とは，空行やコメント行は除き，実際の処理部分のみの行数の合計である．

4.3 実験の設定

4.3.1 RQ1: 提案手法は既存手法 Ekstazi を上回るか？

RQ1 の実験設定について説明する．RQ1 では，提案手法と Ekstazi の性能を比較し，提案手法の有用性を評価する．具体的には，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．本研究では，テスト選択を利用しない場合，提案手法を適用した場合，Ekstazi を適用した場合，の 3 つのシナリオを実行し，テスト選択を利用しない場合に対する，提案手法と Ekstazi それぞれのテストケース削減率とエンドツーエンド時間削減率の観点から評価を行う．

テストケース削減率とは，テスト選択を利用しない場合，つまり全てのテストを実行した場合に対して，テスト選択技術を適用した場合に，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．あるリビジョン r のテストケース数を T_r，テスト選択技術によって選択されたテストケース数を $T'r$ とし，n 回のリビジョンにわたる平均テストケース削減率 $Rtest$ は以下のように表せる．

$$Rtest = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i - T'_i}{T_i} \times 100.$$

エンドツーエンド時間の削減率は，全てのテストを実行する場合に対して，テスト選択技術を適用した場合に，実行時間をどの程度削減できたかを示す指標である．エンドツーエンド時間とは，テスト実行及びテスト選択技術が費やす時間の合計である．Ekstazi の場合，ビルドコマンドを実行することで，ファイルの変更の有無の確認からテスト選択，テスト実

行，依存関係の更新が行われるため，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，ソース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，それらの合計をエンドツーエンド時間として計測する．あるリビジョン r の全テスト実行時のエンドツーエンド時間を t_r，テスト選択技術を使用したリビジョン r のエンドツーエンド時間を $t'r$ とした場合，n 回のリビジョンにわたる平均エンドツーエンド時間削減率 $Re2e$ は以下のように表現できる．

$$R_{e2e} = \frac{1}{n}\sum_{i=1}^{n}\frac{t_i - t'_i}{t_i} \times 100.$$

4.3.2 RQ2: 提案手法の適合率はどの程度か？

RQ2 の実験の設定について説明する．RQ2 では，提案手法が必要なテストを削減していないかを確認するため，提案手法の適合率を評価する．既存のテスト選択手法を比較した論文では，変更のあるファイルにミュータントを埋め込み，テスト選択技術によって選択したテストでどの程度のミュータントを検出できるかを調べ，欠陥検出力を求めている [23]．この手法は，ファイルレベルの依存関係を用いるテスト選択手法に適用できるが，行単位の依存関係を用いている本手法には適用できない．そこで本研究では，変更後のテストとソースコードの依存関係から変更箇所を通過するテストを抽出し，抽出したテストを正解データとして扱い，提案手法で選択したテストが正解データとどの程度一致しているかを調べる．この調査手法では，変更内容が追加もしくは編集であれば，変更後の依存関係から正解データを抽出できるが，削除があった場合は，削除箇所が変更後に存在しないため，正解データを抽出できない．しかし，削除があった場合について，提案手法では削除された行を通過するテストを選択するため，削除があった場合の適合率はある程度担保されていると考えられる．従って，追加または編集があった場合の，提案手法のテスト選択の適合率を求める．正解データに含まれるテストメソッドの集合を $\{T_c\}$，提案手法で選択したテストメソッドの集合を $\{T_p\}$ とすると，適合率 $precision[\%]$ は以下のように表現できる．

$$precision = \frac{|\{T_c\} \cap \{T_p\}|}{|\{T_c\}|} \times 100$$

4.3.3 RQ3: 提案手法が有用に働くプロジェクトの特徴は何か？

RQ3 の実験の設定ついて説明する．提案手法が有用に働くようなプロジェクトの特徴を明らかにするため，提案手法の有用性及び各手法のエンドツーエンド時間削減率とプロジェクトの特徴の相関を調査する．提案手法の有用性を，Ekstazi のエンドツーエンド時間削減率に対する，提案手法のエンドツーエンド時間削減率の差分 (U_{diff}) として定義する．Ekstazi

のエンドツーエンド時間削減率を R_{e2eE}，提案手法のエンドツーエンド時間削減率を R_{e2eP} とした場合，U_{diff} はそれぞれ下記のように表せる．

$$U_{diff} = R_{e2eP} - R_{e2eE}$$

プロジェクトの特徴として，いくつかのメトリクスを利用する．
静的メトリクスとして，以下の 5 個のメトリクスを利用する．

- プロダクトコードのクラス数
- テストコードのクラス数
- テスト数
- 論理 LOC
- クラスベースの条件分岐の割合

クラスベースの条件分岐の割合とは，全クラスの LOC のうち，条件分岐を含む行数の割合である．クラス数やテスト数，論理 LOC はコードメトリクスとして広く利用されており，これらのメトリクスから提案手法の有用性を予測することができれば望ましいため，これらのコードメトリクスを用いる．また，条件分岐の割合はテストの削減率に影響を与える可能性があり，併せてエンドツーエンド時間にも影響を与える可能性があるため，クラスベースの条件分岐の割合も用いる．
動的メトリクスとして，以下の 10 個のメトリクスを利用する

- テスト実行時間
- テストベースの条件分岐の割合
- メソッド呼び出し回数
- バイトコード命令の実行回数
- メソッド呼び出し命令の種類数
- バイトコード命令の種類数
- 1 テストあたりのメソッド呼び出し回数 （メソッド呼び出し回数 / テスト数）
- 1 テストあたりのバイトコード命令の種類数（バイトコード命令の種類数 / テスト数）

- 1 テストあたりのメソッド呼び出し命令の種類数
  （メソッド呼び出し命令の種類数 / テスト数）
- 1 テストあたりのバイトコード命令の種類数（バイトコード命令の種類数 / テスト数）

テストベースの条件分岐の割合とは，全テストが実行する行数のうち，条件分岐を含む行数の割合である．命令の種類数は，同じ箇所で繰り返し実行される命令を 1 種類としてカウントする．一方，命令の実行回数については，同じ箇所で繰り返し実行された場合，実行された回数分カウントする．これらの動的メトリクスは，実行トレースの収集時間やテストの実行経路の複雑さの特徴を捉えている可能性があり，提案手法の有用性に関連する可能性があるため，本調査で利用する．今回の実験では，最も古いリビジョンに対して取得したメトリクスを利用する．

4.4 評価結果

4.4.1 RQ1：提案手法は既存手法 Ekstazi を上回るか？

全テスト実行時間と，提案手法および Ekstazi それぞれのテスト削減率，エンドツーエンド時間削減率を表 2 に示す．テスト数削減率に関しては，全てのプロジェクトで提案手法が Ekstazi を上回った．提案手法がテスト数削減率を上回った理由として，提案手法側が各テストケースの実行経路を考慮しており，無駄なテストが省けている点や，Ekstazi がテストをテストクラス単位で選択しているのに対し，提案手法がテストメソッド単位で選択している点が考えられる．一方で，各リビジョンで選択したテストの数を調査したところ，提案手法が Ekstazi よりも多くのテストを選択しているケースがあった．例えば，変数のリネームや，バイナリには変更のないリファクタリング等の編集が行われた場合である．Ekstazi ではバイナリファイルのチェックサムを用いてファイルの変更の有無を判断しているため，バイナリに変更のない編集は無視できるが，提案手法はファイルのトークン列の差分で変更の有無を判断しているため，無視できない．このような変更を考慮するためには，Ekstazi と同様に，バイナリファイルのチェックサムを利用して，変更の有無を判断する必要がある．

エンドツーエンド時間削減率に関しては，4 つのプロジェクトでは Ekstazi を上回ったが，3 つのプロジェクトでは Ekstazi を下回った．また，4 つのプロジェクトでは，エンドツーエンド時間削減率が負の値になっており，全テストを実行するよりも実行時間が長いという結果になった．選択したテストの実行時間とテスト選択技術によって生じるオーバーヘッドがどの程度かを表すエンドツーエンド時間の内訳を図 6 に示す．各プロジェクトにおいて，全テストを実行した時間，Ekstazi を適用した場合に選択したテストの実行時間とオーバーヘッド，提案手法を適用した場合に選択したテストの実行時間とオーバーヘッドを示してい

る．commons-cli や hilbertCurve，commons-jxpath などのテスト実行時間が短いプロジェクトでは，全てのテスト実行時間に対して提案手法のオーバーヘッドの比率が大きく，削減したテスト実行時間よりも大きい．従って，これらのテスト実行時間が短いプロジェクトにおいて，提案手法のエンドツーエンド時間は，全てのテストを実行した場合よりも大きくなっている．一方，実行時間の長い jitwatch や commons-net では，全てのテスト実行時間に対してオーバーヘッドの比率が低く，かつテスト実行時間が大幅に削減できていることから，エンドツーエンド時間も大幅に削減できている．NuProcess においても，削減したテスト実行時間がオーバーヘッドより大きく，エンドツーエンド時間を削減できている．このようにテスト実行時間が長いプロジェクトでは，提案手法によって削減されるテスト実行時間が，提案手法によって生じるオーバーヘッドを上回り，高いエンドツーエンド時間削減率を達成できる傾向がある．

各プロジェクトにおけるエンドツーエンド時間の削減率の箱ひげ図を図 7 に示す．全プロジェクトにおいて，提案手法のエンドツーエンド時間削減率の最小値が，Ekstazi のエンドツーエンド時間削減率の最小値より小さくなっている．図 6 を見ても分かるように，基本的には提案手法のオーバーヘッドが Ekstazi のオーバーヘッドよりも大きいため，初回のリビジョンなど全てのテストを実行する際に，提案手法のエンドツーエンド時間削減率が Ekstazi のエンドツーエンド時間削減率よりも小さくなる．

各プロジェクトの箱ひげ図に着目すると，commons-jxpath と hilbertCurve については，中央値は両手法で大差はないが，平均値は提案手法が大きく下回っている．提案手法側の第一四分位数，最小値が Ekstazi に比べて大きく下回っていることから，いくつかのリビジョンにおいて，エンドツーエンド時間削減率が非常に悪くなっていることがわかる．この原因は，特定のテストを実行した場合に，実行トレースの収集時間が非常に長くなることが考えられる．そこで，実行トレースを取得する SELogger を組み込まずに全テストを実行する場合と SELogger を組み込んで全テストを実行する場合を比較したところ，commons-jxpath は約 563%，hilbertCurve は約 1354%もテスト実行時間が増加した．残りの jacksonXML，jitwatch，NuProcess，commons-net に関しては，平均値，中央値ともに，提案手法が上回っている．jacksonXML に関しては，他のプロジェクトに比べ，クラス数が多いため，クラスのハッシュを取っている Ekstazi のオーバーヘッドが大きく，Ekstazi のエンドツーエンド時間削減率が低い結果になったと考えられる．jitwatch，NuProcess，commons-net に関しては，提案手法の第一四分位数が大きいことから，実行時間の長いテストが省くことができたため，エンドツーエンド時間削減率が高い値になっている可能性が考えられる．実際に，jitwatch には実行時間が約 30 秒のテストクラス，NuProcess には約 27 秒，約 17 秒のテストクラス，commons-net には，約 47 秒, 約 45 秒のテストクラスが含まれている．これらの実行時間の長いテストが細かい粒度の解析を用いている提案手法で省くことができる場合，

表 2: テスト削減率，エンドツーエンド時間削減率の結果

Projects	全テスト実行時間 [s]	テスト削減率 [%]		エンドツーエンド時間削減率 [%]	
		提案手法	Ekstazi	提案手法 (R_{e2eP})	Ekstazi (R_{e2eE})
commons-cli	13.25	**89.85**	67.42	-4.07	**17.11**
hilbertCurve	14.65	**81.66**	71.40	-77.07	**39.93**
commons-jxpath	18.77	**84.61**	74.81	-38.56	**13.58**
jacksonXML	19.73	**70.54**	52.70	**-14.45**	-30.47
jitWatch(core)	38.91	**89.82**	75.05	**48.17**	33.18
NuProcess	65	**32.30**	29.29	**25.21**	21.88
commons-net	124	**95.44**	90.03	**82.26**	77.56

より有用に働く可能性がある．

RQ1 の回答

テスト数削減率では，全プロジェクトにおいて提案手法が既存手法 Ekstazi を上回った．エンドツーエンド時間削減率では，半数のプロジェクトで提案手法が既存手法 Ekstazi を上回った．また，テスト実行時間が長いプロジェクトにおいては，提案手法によって削減されるテスト実行時間が，生じるオーバーヘッドを上回り，有用に働く傾向にある．

4.4.2 RQ2: 提案手法の適合率はどの程度か？

RQ2 の結果を表 3 に示す．precision は，各プロジェクトのリビジョンに対して，連続的に提案手法を適用し，選択したテストの適合率である．precision において，jitwatch では適合率が 100%であったが，他のプロジェクトでは適合率が 100%にはならなかった．また，jacksonXML，commons-net では，非常に低い値になっている．連続的に提案手法を適用した場合，あるリビジョンで必要なテストが選択されなかった場合に，テストとソースコードの依存関係が一部欠落し，その後のリビジョンで正しいテストを選択できていない可能性がある．そこで，各リビジョンにおいて，変更前の全テストを実行して構築した正確な依存関係を用いて提案手法を適用し，選択したテストの適合率を precision*として調査した．その結果，precision と precision*を比較すると，precision*の方が全体として値が大きく，特に commons-net の場合は，60%から 97%と大幅に向上している．この結果から，必要なテストが選択されなかった場合に起きる一部の依存関係の欠落が，後のリビジョンでのテスト選択での適合率に大きな影響を与えていることが分かる．しかし precision*においても，全体として適合率は上がっている一方，100%でないプロジェクトが存在していることから，正

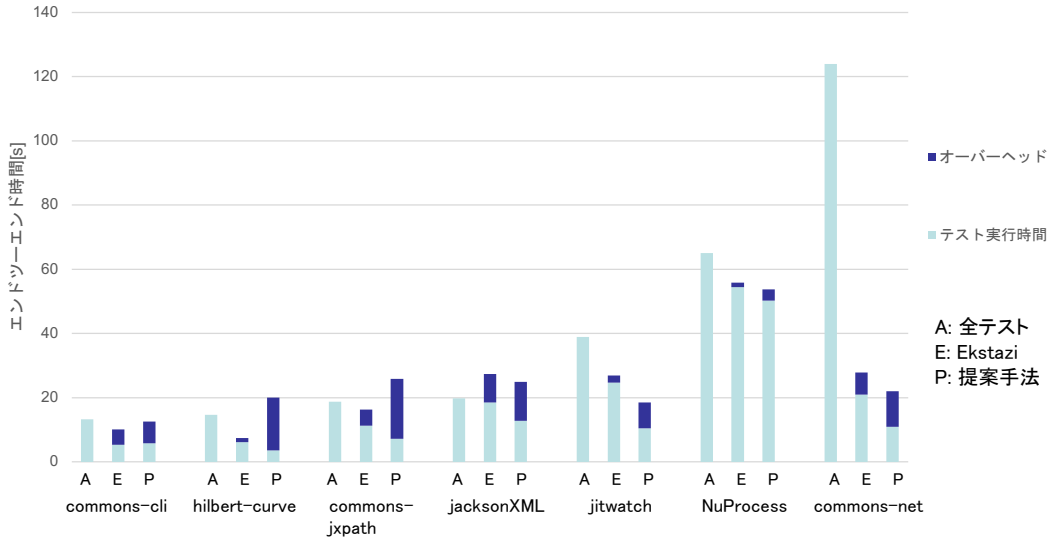

図 6: エンドツーエンド時間の内訳

確な依存関係を用いた場合でも，必要なテストが選択できていないことが分かる．そこで，precision*が 100%にならない原因を各リビジョンの変更内容をもとに調査した．その結果，提案手法で正しいテストが選択できない場合を 6 個特定した．それぞれの場合について説明する．

1 つ目は，オーバーライドするメソッドの追加があった場合である．オーバーライドするメソッドが追加された場合，該当のメソッドを呼び出している箇所を実行するテストを選択する必要があるが，本手法では追加があった直前行を選択するため，不十分である．この場合に対応するためには，各メソッドを呼び出す箇所を記録するとともに，変更内容にオーバーライドが含まれるかどうかを解析する必要がある．

2 つ目は，実行毎に実行経路が変わるテストの存在する場合である．実行毎に実行経路が変わることで，テストとコード間の正確な依存関係が構築できないため，正しくテストを選択できない場合がある．今回対象としたプロジェクトにも該当のテストが存在するかを調査した．具体的には，同一のリビジョンにおいて，全テストの実行及びテストとソースコードの依存関係の構築を 2 回行い，生成される 2 つの依存関係ファイルの中身が一致しているかをハッシュを用いて検査した．その結果，commons-cli，commons-jxpath，NuProcess では，生成される 2 つの依存関係ファイルのハッシュ値が一致せず，実行毎に実行経路が変わるテストが存在した．この場合への対応は，行単位の依存関係を用いる本手法では難しいと考えられる．

3 つ目は，フィールド変数を新たに使った変更である．例えば，以前はフィールド変数を

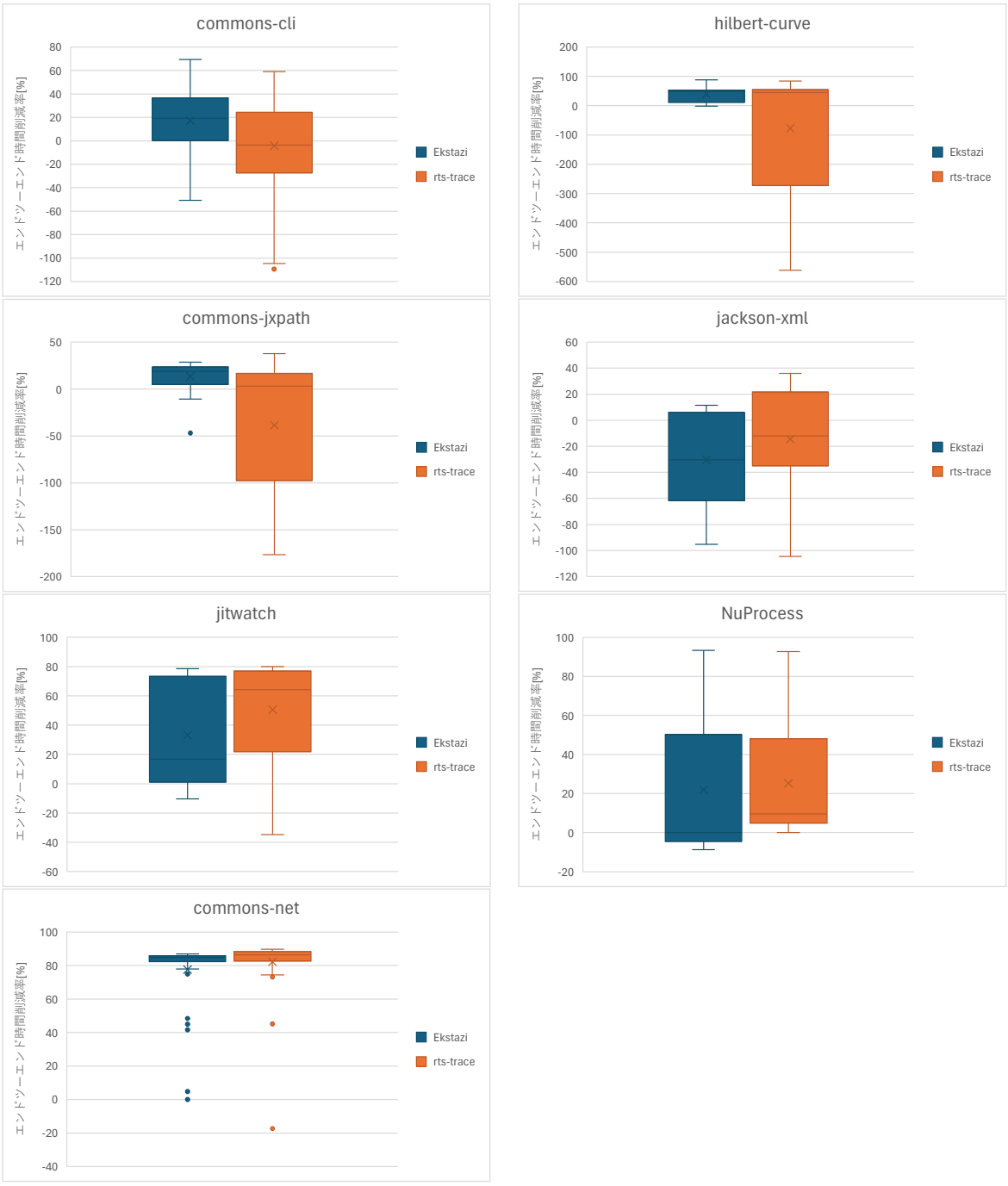

図 7: 各プロジェクトのエンドツーエンド時間削減率

使っていなかった行に，フィールド変数を使った文を追加した場合である．この変更の場合，該当のフィールド変数を参照するテストを選択する必要があるが，本手法では選択できない．この場合に対応するためには，変更内容を解析し，フィールド変数が含まれるかを調べる必

要がある．

4 つ目は，複数行に跨るコードの編集である．編集例を図 8 に示す．この編集例において，提案手法はトークン列で差分を取っているため，2 行目にあるトークンに編集があったと判別する．従って 2 行目を実行するテストを選択する．しかし，SELogger では，2 行に跨るコードの行番号を 1 行目の行番号として扱い，実行経路として 1 行目の行番号を記録する．従って，テストとソースコードの依存関係において，編集箇所の 2 行目は存在せず，1 行目のみ存在するため，提案手法において必要なテストが選択できない．この場合に対応するためには，SELogger を改造し行番号の記録方法を変更したり，ソースコードを解析し，2 行に跨るコードを特定する必要がある．

5 つ目は，メソッド内の先頭に追加があった場合である．SELogger で解析する場合，メソッドの宣言箇所が先頭行として記録されず，メソッド内の実行可能な先頭の行が先頭行として記録される．従って，メソッド内の先頭に追加があった場合，提案手法では直前行を実行するテストを選択しようとするが，そのようなテストはテストとソースコードの依存関係には存在せず，選択できない．この場合に対応するためには，SELogger を改造し記録方法を変更したり，ソースコードを解析しメソッドの宣言箇所を記録しておく必要がある．

6 つ目は，文の先頭に追加があった場合である．例えば，final 修飾子の追加である．この場合，本来は該当の行に変更があるため，該当する行を実行するテストを選択する必要がある．しかし，提案手法ではトークン列で差分を取っているため，トークンの追加があったと判別し，直前のトークン，つまり直前の行を実行するテストを選択してしまい，該当の行を実行するテストを選択できない．一般的に修飾子の追加などは，本来テストは必要がない変更になるため，テストが選択できなかった場合でも大きな影響はないと考えられるが，この場合に対応するためには，トークン列で差分を取った際，行の先頭であるかを判定する必要がある．

これらに対応する場合，変更内容やテスト実行時情報をより深く解析する必要があるため，現在の手法に比べてよりオーバーヘッドが増加すると考えられる．

RQ2 の回答

適合率は，平均値が 88.35%，中央値が 93.96%，適合率*は平均値が 96.67%，中央値が 98.83%であり，いくつかの変更パターンに対しては対応できていなかったが，基本的な変更パターンには対応できていた．適合率に比べて，適合率*の値が大きく向上したことから，テストとコード間の依存関係が一部欠落すると，それ以降のテスト選択における適合率に大きな影響を及ぼす．対応できていないパターンへ対応するためには，変更内容やテスト実行時情報をより深く解析する必要がある．

```
1 -                 System.out.println(
2 -                     "Usage: aa");
3 +                 System.out.println(
4 +                     "Usage: bb");
```

図 8: 複数行に跨るコードの編集例

表 3: 提案手法の適合率

プロジェクト名	precision[%]	precision*[%]
commons-cli	99.21	99.88
hilbertCurve	96.20	98.83
commons-jxpath	93.96	100
jacksonXML	75.80	81.89
jitwatch	100	100
NuProcess	93.33	98.73
commons-net	60.00	97.39

4.4.3 RQ3: 提案手法が有用に働くプロジェクトの特徴は何か？

提案手法の有用性，各手法の削減率と各メトリクス間の相関係数を表 4 に示す．提案手法の有用性と各メトリクス間で t 検定を行ったところ，p 値が 0.05 未満となるような有意差のあるデータは「U_{diff}」と「バイトコード命令の実行回数」,「U_{diff}」と「1 テストあたりのバイトコード命令の実行回数」(バイトコード命令の実行回数 / テスト数),「R_{e2eP}」と「テスト実行時間」,「R_{e2eE}」と「テストクラス数」,「R_{e2eE}」と「テスト実行時間」の各相関結果のみであり，その他のデータに有意差は見られなかった．有意差のあるデータのうち,「U_{diff}」と「バイトコード命令の実行回数」,「U_{diff}」と「1 テストあたりのバイトコード命令の実行回数」はそれぞれ強い負の相関関係を示している．有意差はなかったが,「R_{e2eE}」と「バイトコード命令の実行回数」は正の相関を表している一方で,「R_{e2eP}」と「バイトコード命令の実行回数」は負の相関を示している.「R_{e2eP}」と「1 テストあたりのバイトコード命令の実行回数」についても,「R_{e2eE}」と「1 テストあたりのバイトコード命令の実行回数」の相関結果とは傾向が異なり，負の相関を示している．以上のことから,「バイトコード命令の実行回数」,「1 テストあたりのバイトコード命令の実行回数」が多い場合，提案手法は Ekstazi に対して有用でなくなる傾向にある．これは,「バイトコード命令の実行回数」が多い場合に，SELogger を用いた実行トレースの収集に時間を要することが原因として考えられる．また,「クラスベースの条件分岐の割合」や「テストベースの条件分岐の割合」に関し

表 4: プロジェクトのメトリクスと U_{diff}，各 R_{e2e} の相関係数

メトリクス	U_{diff}	R_{e2eP}	R_{e2eE}
クラス数	0.06	0.18	0.20
テストクラス数	0.29	-0.26	-0.77*
テスト数	0.18	0.01	-0.23
論理 LOC	0.37	0.48	0.24
クラスベースの条件分岐の割合 [%]	-0.04	-0.23	-0.30
テスト実行時間 [s]	0.37	0.80*	0.72*
メソッド呼び出し回数	0.19	0.63	0.69
バイトコード命令の実行回数	-0.83*	-0.50	0.33
メソッド呼び出し回数 / テスト数	0.29	0.52	0.41
バイトコード命令の実行回数 / テスト数	-0.83*	-0.56	0.23
メソッド呼び出し命令の種類数	0.37	0.27	-0.06
バイトコード命令の種類数	0.31	0.26	0.00
メソッド呼び出し命令の種類数 / テスト数	0.52	0.50	0.08
バイトコード命令の種類数 / テスト数	0.46	0.49	0.14
テストベースの条件分岐の割合 [%]	-0.08	-0.54	-0.71

*：統計的に有意（p 値 <0.05）

て，条件分岐の割合が大きくなれば提案手法が有用に働くと考えていたが,「U_{diff}」とはほとんど相関がなく,「R_{e2eP}」とは負の相関を示している．

「R_{e2eP}」と「テスト実行時間」,「R_{e2eE}」と「テスト実行時間」はそれぞれ正の相関関係にある．この結果から，テスト選択技術はテスト実行時間が長いプロジェクトのほうが有用に働く傾向がある．また,「R_{e2eP}」と「テスト実行時間」の相関が「R_{e2eE}」と「テスト実行時間」の相関よりも強いため，提案手法の方がよりテスト実行時間に依存して有用になる傾向にある．

RQ3 の回答

提案手法が有用に働くプロジェクトの特徴は，本調査で用いたメトリクスでは見つからなかった．一方,「バイトコード命令の実行回数」が多いプロジェクトでは，提案手法の解析時間が増大し，有用に働かない傾向にある．

5　妥当性の脅威

5.1　構成概念妥当性

今回の評価実験において，提案手法の適合率の調査を行ったが，実際に十分な故障検出能力を有しているかを評価していない．変更後のプログラムにおいて，変更箇所を通過するようなテストを正解データとして扱っているため，正解データに大きな問題はないと考えられるが，正解データが不十分である可能性はある．提案手法が十分な故障検出能力を有しているかを評価するには，欠陥データセットに手法を適用したり，ミューテーションテストを行うなどして評価を行う必要がある．

5.2　外的妥当性

今回は 7 個の OSS に対して評価実験を行ったが，評価実験の結果については，評価対象のプロジェクトの特徴によって変わりうる．今回はこの脅威を軽減するため，規模やテスト実行時間，リビジョン数が異なる複数のプロジェクトで実験を行った．また，対象としたリビジョンの変更内容にも評価の結果は依存する．この脅威を軽減するため，最新のリビジョンから最大 50 リビジョンを評価対象のリビジョンとし，多様な変更内容を含むようにした．

5.3　内的妥当性

提案手法では，行単位の依存関係を用いているが，本来複数行で書かれるべきプログラムが 1 行で書かれていた場合については考慮していない．既存手法は，クラスファイル以外に設定ファイル等の外部ファイルと依存関係を取得しているが，提案手法では外部ファイルとの依存関係を取得していない．既存手法はプラグインや javaAgent としてテスト選択に関するフェーズがすべて自動で実行され，その実行時間を計測しているが，提案手法では diff を取る部分や各フェーズの実行など，本来手動の各フェーズをスクリプトで自動化し，各フェーズの実行時間を合計して，全体の実行時間を計測している．そのため，プラグイン内での動作等，既存手法側にのみかかる時間や，実行トレースファイルの読み書き等の提案手法側のみにかかる時間があり，依存関係の粒度の差に関連のない点で実行時間に差が生じている可能性がある．提案手法において，選択したテストがない場合に本来必要のないコンパイルを行うことで，既存手法の Ekstazi と差をなくし，依存関係の粒度に着目した比較評価を行っている．厳密な比較評価を行うためには，Ekstazi と同様に，提案手法をプラグインや JavaAgent として実装する必要がある．

6　おわりに

本研究では，テスト-コード間の行単位の動的な依存関係を用いたテスト選択手法を提案し，既存手法の Ekstazi と比較評価を行った．評価の結果，半数のプロジェクトにおいて，提案手法が最先端の既存のテスト選択手法よりも高いエンドツーエンド時間削減率を示した．提案手法の適合率を調査したところ，いくつかの変更パターンには対応できていなかったが，よくある変更パターンについては基本的に対応できていた．また，正しいテストが選択できていないリビジョンの変更内容を調査し，対応できていないいくつかの変更パターンを特定した．対応できていない変更パターンへの対応については，SELogger の改造や本手法の実装面でサポートの必要がある．プロジェクトの特徴と提案手法の有用性の関連性を調査したところ，提案手法が有用に働くプロジェクトの特徴は特定できなかったが，バイトコード命令の実行回数が増加するにつれ，提案手法が有用に働かなくなる傾向があることが分かった．今後の課題は以下の通りである．

対応できていない変更パターンへの対応と対応後の性能調査

本手法において，対応できていない変更パターンをいくつか発見した．今後の課題として，より複雑な解析手法を提案手法に組み合わせることによる，解析の精度向上があげられる．本手法で用いたソースコードの差分と行単位の実行経路の情報に加えて，継承関係や実行ごとのランダム性，さらには実行に影響を与えない修飾子の考慮のような複雑な解析を組み合わせることで，より多くの変更パターンを網羅できる．一方で，解析の複雑さは増し，実行時のオーバーヘッドも増加すると考えられるため，精度とパフォーマンスのトレードオフについて考慮する必要がある．

プラグインや JavaAgent としての実装

本研究では，テストを選択するフェーズ，テストを実行しテスト実行時情報を収集するフェーズ，依存関係を構築/更新するフェーズを独立して実装した．ソフトウェア開発者が実務でテスト選択技術を利用する際には，各フェーズを手動で行うことはコストがかかり，利用しづらいと考えられる．従って，行単位の依存関係を用いた手法を，Ekstazi のようにプラグインや JavaAgent として実装することが好ましいと考えられる．

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 肥後芳樹 教授には，研究活動に対して多くの貴重な御助言や御指導を賜りました．心より深く感謝申し上げます，

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，研究室の発表機会において，御意見，御助言を賜りました．松下誠 准教授に心より深く感謝申し上げます．

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 神田哲也 助教には，研究活動の直接のご指導，論文の執筆に至るまで，あらゆる場面で多くのご指導を賜りました．心より深く感謝申し上げます．

奈良先端科学技術大学院大学先端科学技術研究科情報科学領域 嶋利一真 助教には，研究活動における様々な御支援や御助言，論文執筆における数多のご指導を賜りました．心より深く感謝申し上げます．

最後に，その他様々な御指導，御助言等を賜りました，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肥後研究室の皆様に，心より深く感謝申し上げます．

参考文献

[1] Openclover - java, groovy and aspectj code coverage tool. `https://openclover.org/`. (Accessed on 02/01/2024).

[2] Songyu Chen, Zhenyu Chen, Zhihong Zhao, Baowen Xu, and Yang Feng. Using semi-supervised clustering to improve regression test selection techniques. In *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation*, pp. 1–10. IEEE, 2011.

[3] Tsong Yueh Chen and Man Fai Lau. Dividing strategies for the optimization of a test suite. *Information Processing Letters*, Vol. 60, No. 3, pp. 135–141, 1996.

[4] Emelie Engström and Per Runeson. A qualitative survey of regression testing practices. In *Product-Focused Software Process Improvement: 11th International Conference, PROFES 2010, Limerick, Ireland, June 21-23, 2010. Proceedings 11*, pp. 3–16. Springer, 2010.

[5] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 211–222, 2015.

[6] Pooja Gupta, Mark Ivey, and John Penix. Testing at the speed and scale of google, 2011.

[7] Dan Hao, Lingming Zhang, Lu Zhang, Gregg Rothermel, and Hong Mei. A unified test case prioritization approach. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 24, No. 2, pp. 1–31, 2014.

[8] M Jean Harrold, Rajiv Gupta, and Mary Lou Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 2, No. 3, pp. 270–285, 1993.

[9] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. In *Proceedings. Conference on Software Maintenance 1990*, pp. 302–310, 1990.

[10] Bill Hetzel. *The complete guide to software testing*. QED Information Sciences, Inc., 1988.

[11] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *Proceedings of the International Symposium on Foundations of Software Engineering*, pp. 583–594, 2016.

[12] Owolabi Legunsen, August Shi, and Darko Marinov. Starts: Static regression test selection. In *Proceedings of the International Conference on Automated Software Engineering*, pp. 949–954, 2017.

[13] Bil Lewis. Debugging backwards in time. *CoRR*, Vol. cs.SE/0310016, , 2003.

[14] Zheng Li, Mark Harman, and Robert M Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering*, Vol. 33, No. 4, pp. 225–237, 2007.

[15] Yiling Lou, Dan Hao, and Lu Zhang. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 46–57. IEEE, 2015.

[16] Mika V Mäntylä, Bram Adams, Foutse Khomh, Emelie Engström, and Kai Petersen. On rapid releases and software testing: a case study and a semi-systematic literature review. *Empirical Software Engineering*, Vol. 20, pp. 1384–1425, 2015.

[17] Atif Memon, Zebao Gao, Bao Nguyen, Sanjeev Dhanda, Eric Nickell, Rob Siemborski, and John Micco. Taming google-scale continuous testing. In *Proceedigns of the International Conference on Software Engineering: Software Engineering in Practice Track*, pp. 233–242, 2017.

[18] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. Exploring test suite diversification and code coverage in multi-objective test case selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pp. 1–10, 2015.

[19] Pilar Rodríguez, Alireza Haghighatkhah, Lucy Ellen Lwakatare, Susanna Teppola, Tanja Suomalainen, Juho Eskeli, Teemu Karvonen, Pasi Kuvaja, June M Verner, and Markku Oivo. Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of systems and software*, Vol. 123, pp. 263–291, 2017.

[20] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pp. 179–188, 1999.

[21] Kazumasa Shimari, Takashi Ishio, Tetsuya Kanda, and Katsuro Inoue. Near-omniscient debugging for java using size-limited execution trace. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pp. 398–401, 2019.

[22] Kazumasa Shimari, Masahiro Tanaka, Takashi Ishio, Makoto Matsushita, Katsuro Inoue, and Satoru Takanezawa. Selecting test cases based on similarity of runtime information: A case study of an industrial simulator. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pp. 564–567, 2022.

[23] Min Kyung Shin, Sudipto Ghosh, and Leo R. Vijayasarathy. An empirical comparison of four java-based regression test selection techniques. *Journal of Systems and Software*, Vol. 186, p. 111174, 2022.

[24] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability*, Vol. 22, No. 2, pp. 67–120, 2012.

[25] Jianyi Zhou, Junjie Chen, and Dan Hao. Parallel test prioritization. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 31, No. 1, pp. 1–50, 2021.