Vol.2022-SE-211 No.30
2022/7/30

# ライブラリアップデート時のテスト失敗解析におけるライブラリテスト差分の有効性に関する検討

小泉 雄太[1,a)]　野元 励[1,b)]

**概要**：ライブラリの利用はアプリケーション開発において一般的となっている．ライブラリは脆弱性やバグを修正した新しいバージョンが随時リリースされるため，ライブラリを利用したアプリケーション運用では，利用しているライブラリのアップデートを日々実施する必要がある．ライブラリアップデートによりアプリケーションのテストが失敗するようになった場合，デバッグ作業の一環として失敗原因の特定が必要となる．失敗原因の特定には原因となる API 呼び出し箇所と API 仕様の変更内容の二点の把握が必要となる．第三者が作成した OSS ライブラリのアップデートでは，API 仕様の変更内容はアプリケーション開発者にとって非自明であり把握が困難な場合がある．本稿はライブラリアップデートでのテスト失敗について，失敗原因となる API 仕様の変更内容の把握にライブラリテストのアップデートによる差分が有効である可能性があることを示し，失敗原因の把握に有効なライブラリテストの特定を支援する技術を検討した．

**キーワード**：ソフトウェアライブラリ，影響解析，テストエラー解析

# Examination of the effectiveness of library test differences for test failure analysis in updating library

Yuta Koizumi[1,a)]　Nomoto Tsutomu[1,b)]

***Abstract:*** Using libraries is common in application development. Since new versions of libraries that fix vulnerabilities or bugs are released from time to time, it is necessary to update libraries which application depends on too. If the application test fails due to a library update, application developer should identify the cause of the failure as part of the debugging work. To identify the cause of the failure, application developer should understand the two points i.e. 1) the API call location that causes the failure and 2) the API specifications changed by updating. In the library created by a third party such as OSS, the API specifications changed in updating are non-obvious to the application developer and difficult to grasp. This article shows that for test failures in updating library, there is a possibility that library tests with differences are effective in understanding the changed API specifications that cause the failure, and need technique to help identify library tests that are effective in understanding the cause of the error. Moreover, I examined the one of the technique.

***Keywords:*** Software libraries, Impact analysis, Test failure analysis

## 1. 背景

### 1.1 ライブラリアップデートの現状

ライブラリ（Library）とはアプリケーション開発に便利な機能を Application Programming Interface（API）とし

[1] NTT ソフトウェアイノベーションセンタ
〒 108-0023 東京都港区芝浦 3-4-1 グランパークタワー 33F
NTT Software Innovation Center
Granpark Tower 33F, 3-4-1, Shibaura, Minato-ku, Tokyo, 108-0023, Japan
[a)] yuuta.koizumi.hr@hco.ntt.co.jp
[b)] tsutomu.nomoto.cb@hco.ntt.co.jp

て提供するコンポーネントである．以下，本稿で扱うライブラリは，アプリケーション開発者がテストコードを含めたソースコードを参照できるライブラリ（例えば OSS ライブラリ）とする[*1]．

アプリケーションの開発において，ライブラリの利用は一般的となっている．ライブラリを利用したアプリケーション開発には開発工数の削減，品質の向上といったメリットが存在する．1,200 件以上の商用アプリケーションへの調査 [1] では，調査したアプリケーションリポジトリの 99%がライブラリを利用し，ライブラリがアプリケーションのソースコードの実に 70%を占めている．

ライブラリは脆弱性やバグの修正のため，日々新しいバージョンがリリースされる．アプリケーションの運用において，これら脆弱性などのリスクを回避するために日々のライブラリのアップデートが必要となる．一つのアプリケーションは複数のライブラリを利用している場合が多く [2][*2]，アプリケーションが依存するライブラリの更新リリースは頻繁に発生する．

### 1.2 ライブラリアップデートを含めた保守作業

アプリケーション開発者は保守作業時，リリース前に退行（regression）が発生していないかを検証するためにテストを実施する．退行はアプリケーションが起動しなくなる，期待通りの動作をしなくなる現象として定義される [3]．本稿ではテストの失敗を，意図しないデータ出力（Assert 違反），もしくは例外発生によるクラッシュで検出可能なものと想定する．テストが失敗した場合，アプリケーション開発者は失敗原因を特定・修正するデバッグ作業を実施する．このうち，失敗原因の特定は以下の 2 点を把握する作業である．

- テスト失敗の原因箇所（以下，ソースコード位置）
- 原因箇所へ実施した変更処理，変更の意図（以下，仕様の変更内容）

1.1 節で述べたライブラリアップデートもまた，アプリケーション保守作業の一つである．アップデートに伴うライブラリの仕様変更は上記の退行，テストの失敗を引き起こす原因となり [4], [5]，アプリケーションとライブラリの依存関係次第でデバッグ作業が発生する．

### 1.3 変更実施者の違いによるテスト失敗の原因特定の差異

以下では，テスト失敗の原因となる変更をアプリケーション開発者が実施した場合と，第三者が実施した場合について，テスト失敗の原因特定の差異を説明する．

#### 1.3.1 テスト失敗の原因となる変更をアプリケーション開発者が実施した場合

アプリケーション開発者（あるいは開発チーム）がテスト失敗の原因となる変更を実施した場合，テストの失敗原因の特定の関心事は主にソースコード位置である．保守作業がアプリケーション内に閉じるため，変更する仕様内容もアプリケーション開発者が決定する．このとき，テスト失敗の原因となるソースコード位置を特定できれば，当該位置へ行った変更処理，及びその変更内容の把握は容易であるためである．

#### 1.3.2 テスト失敗の原因となる変更を第三者が実施した場合

本項で述べる変更を第三者が実施した場合とは，具体的にはライブラリアップデートを想定している．すなわち，第三者とはライブラリ開発者である．前述の通り，ライブラリアップデートもアプリケーションに退行（テスト失敗）が発生し得る保守作業の一つである．また，そのため，1.3.1 項と同様アプリケーション開発者はテスト失敗時に失敗原因を特定し修正を実施する必要がある．

ライブラリアップデート時，失敗原因の特定ではソースコード位置だけでなく，仕様の変更内容も関心事となる．1.3.1 項において仕様の変更内容の把握が容易なのは，テスト失敗原因を引き起こす変更処理を実施したのがアプリケーション開発者自身（もしくは開発チーム）であるためである．ライブラリアップデートの場合，テスト失敗原因を引き起こす変更処理は第三者（ライブラリ開発者）によって実施される．そのためアプリケーション開発者はソースコード位置，例えば原因となる API 呼び出し箇所などを把握したとしても API 仕様の変更内容までは把握できない[*3]．つまりライブラリアップデート時においては，ソースコード位置だけではなく仕様の変更内容の把握も別途必要となる．

### 1.4 既存研究

テスト失敗原因を特定することを目的とした既存研究には，Fault Localization[6], [7], [8]，などの研究分野が存在する．

（Spectrum-based）Fault Localization[6], [7] は失敗原因となるソースコード位置（コード行）を特定することを目的とした手法である．特に Spectrum-Based[6], [7] な手法は単一の失敗原因について，複数のテストを実行し，失敗するテスト固有のコード行を識別する手法である．これは失敗するテスト固有のコード行はテスト失敗と関連性が高いという直感に根ざしている．

[*1] ライブラリの提供形式やレイヤは様々であり，OSS ライブラリをはじめとする外部ライブラリ，OS カーネルが提供するシステムコール群，言語処理系が提供する標準ライブラリ，サーバ等ミドルウェアが提供する操作インターフェース，HTTP プロトコルを利用した Web API サービス等が存在する．

[*2] [2] の調査によると，Java プロジェクトでは中央値 20 個のライブラリが利用されている．

[*3] 仮にライブラリ内部の原因箇所を把握できたとしても，アプリケーション開発者にとって第三者が実装したライブラリの内部実装の理解は困難であることが多い．

```
1  import org.jsoup.Jsoup;
2  import org.jsoup.Document;
3
4  @Test public void testAddTable(){ // APP test
5    String html =
6      "...␣hoge.com?North=true&Lang&Zone=jp␣...";
7    Document d = Jsoup.parse(html);
8      ...
9    APP.addTable(d);
10   AssertEqual(
11     "...␣hoge.com?North=true&Lang&Zone=jp␣...",
12     d.body().html());
13 }
```

**図 1** Web アプリケーションのテストの例
**Fig. 1** A example of test code in web application

Fault Localization はテスト失敗の原因となるソースコード位置の特定に焦点を当てている．Fault Localization はテスト失敗の原因となる仕様の変更内容の把握を支援する技術としては十分ではない．これは Fault Localization が変更処理がアプリケーション内に閉じていることを前提にしているためである．

### 1.5 本稿の構成

本稿はライブラリアップデートでのテスト失敗について，失敗原因となる仕様の変更内容の把握に活用可能な情報を整理した上で，ライブラリテストのアップデートによる差分が有効である可能性が高いことを示す．また，テスト失敗原因と関連するライブラリテストの特定を支援する技術を検討した．

本稿の構成は次の通りである．まず，2 章ではライブラリアップデートに伴うアプリケーションのテスト失敗例を元に，テスト失敗原因の仕様の変更内容の把握にライブラリテストのアップデートによる差分が有効である可能性が高いこと，支援技術が必要であることを説明する．3 章では支援技術の一例として検討した手法について説明する．4 章では 3 章で提示した支援技術の評価とその結果について説明する．5 章で関連研究について説明する．6 章でまとめを述べ本稿の結びとする．

## 2. 検討及び提案

本章ではライブラリアップデートに伴うアプリケーションのテスト失敗例を元に，既存研究では支援されてこなかった仕様の変更内容の把握に活用できる情報を検討する．検討を通して，仕様の変更内容の把握にライブラリテストのアップデートによる差分が有効である可能性があること，また，テスト失敗と関連性の高いライブラリテストを特定する支援技術が必要であることを説明する．

### 2.1 ライブラリアップデートに伴うテスト失敗例

図 1 は HTML パーサライブラリである jsoup 1.5.1 を利用している Web アプリケーションのテスト例である．本例ではアプリケーションの仕様を変えず，ライブラリアップデートのみを実施するとする．この例において jsoup のバージョンを 1.6.0 へアップデートすると，期待する出力にならずテストが失敗する．具体的には，文字列内部の URL パラメータの箇所が `North=true&Lang&Zone=jp` と期待されているが，アップデート後は `North=true《&Zone=jp` となるため assertion 違反となる．

ライブラリアップデートのみを実施することから，アプリケーションテストの失敗はライブラリアップデートによる API の動作変化に起因する．アプリケーション開発者はまず，アプリケーション上においてどこで動作が期待通りではなくなったのかを特定する．この場合，1.4 節で示した既存の技術や，デバッガ等でアップデート前後の各行の実行結果を比較するなどして，7 行目の API 呼び出し `Jsoup.parse(html)` の出力の時点で期待通りではないことを突き止め，`parse()` がテスト失敗原因の API であることを把握する．

しかし，アプリケーション開発者はアプリ上で観測された情報（ログ・トレース等）のみでは API の出力や動作がなぜ `North=true&Lang&Zone=jp` から `North=true《&Zone=jp` へ変化したのかを類推することはできない．アプリ上ではライブラリのソースコード差分をはじめとしたアップデートに関する情報を観測できないため，仮に API 内部の情報（例えば Java 言語で，jar ファイルの中身やバイトコードのトレース情報など）まで原因箇所を掘り下げても同様である．API 動作がなぜ変化したのかをアプリケーション開発者が類推するためには，アップデートに関するライブラリの情報を収集し，その中から API 仕様の変更内容を把握する必要がある．

### 2.2 API 仕様の変更内容の把握に活用できる情報の検討

1.1 節で述べた通り，本稿では OSS ライブラリ等，ライブラリのテストコードを含めたソースコードをアプリケーション開発者が参照できることを想定している．このとき，ライブラリアップデートに関連する情報として以下のような情報が利用できる．

- リリースノート
- コミット（メッセージ，コード差分を含む）
- バージョン間のソースコード差分
- その他

バージョン間のソースコード差分には，API のコードボディ及び API のコードボディから呼び出される内部処理コードの差分（以下，処理コード差分）と，API を呼び出すライブラリテストコードの差分（以下，ライブラリテスト差分）が含まれる．その他には Javadoc などの API Reference ドキュメントや，バグチケット[*4]などの情報，リリースノート・コミットメッセージ内で引用されているリ

[*4] サポートの有無や名称はライブラリを公開しているプラットフォーム（GitHub 等）に依存する．

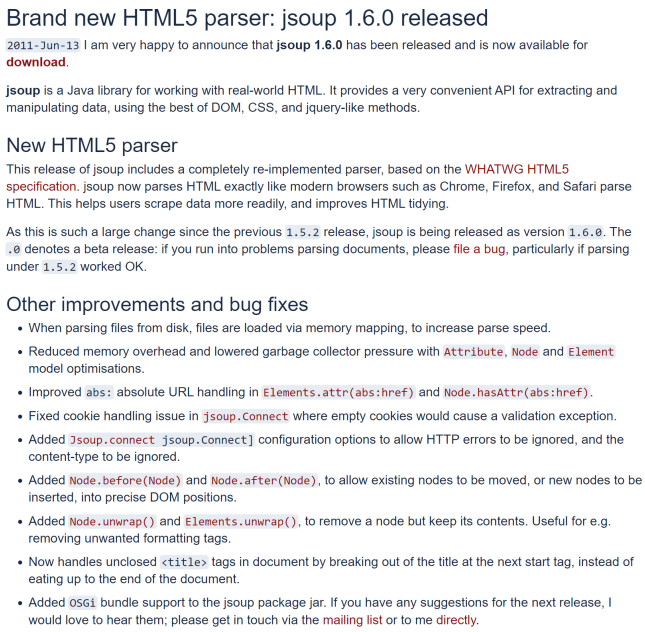


**図 2** リリースノートの記述例
`https://jsoup.org/news/release-1.6.0`
2022/6/16 閲覧

**Fig. 2** A sample of release note

ンク先の情報などが含まれる．その他に含まれる情報はライブラリによっては存在しない場合もあるため，今回の検討では対象外とする．

以下では，2.1 節で示した jsoup の例を元に，各ライブラリアップデートに関連する情報について検討していく．

### 2.2.1 リリースノート

まず，リリースノートについて検討する．図 2 は実際に公開されているリリースノートの記述の一例である．この例では機能の追加，バグ修正に関連した一部の API への変更について言及している．しかし，`parse()` をはじめとした失敗したアプリケーションテストと関連した API への直接の言及はない．`parse()` などの API の仕様変更は HTML5[*5]の仕様へのリンクをもって替えられており，リリースノートのみの情報からではテスト失敗と関連した API 仕様の変更内容の把握はできない．

リリースノートの記述フォーマットはライブラリによって異なるが，本例のように網羅的に個々の API について記述されないケースが存在する．これは本例のように仕様の変更内容の（とりわけ自然言語による）列挙自体がライブラリ開発者にとって現実的ではないと場合があるためと思われる．

なお，API の列挙がある例としては，Guava[*6]のように JDiff[9] から追加・削除・変更された API を自動列挙するケースがある．ただし，この場合でも API を列挙するだけであり，どのような変更がなされたかを把握することはできない．別の例では，Spring[*7] のようにアップデートに伴う退行の修正例を記載した Migration Guide を提供するケースもある．ただし，Migration Guide は Spring の例でもメジャーバージョンアップのような一部のアップデートで作成される[*8]のみで，提供されるケースは限定的となる．

以上より，リリースノートから API 仕様の変更内容の把握ができないケースが多々存在すると考える．

*5 HTML5.0 は 2011 年 5 月に仕様が確定，公開された．https://www.w3.org/TR/2011/WD-html5-20110525/ 2022/6/16 閲覧
*6 `https://github.com/google/guava` 2022/6/16 閲覧

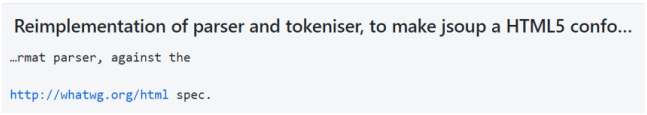


**図 3** テスト失敗の原因となる変更を含むコミットのメッセージの例
`https://github.com/jhy/jsoup/commit/8749726a79c22451b1f01b14fb2137f734e926b4`
2022/6/16 閲覧

**Fig. 3** A sample of commit introducing application test failure

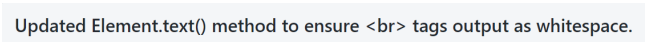


**図 4** API の仕様変更を説明したコミットメッセージの例
`https://github.com/jhy/jsoup/commit/9c5d3cdf94a48dd745fdabddc0996dcd0f4b7ca1`
2022/6/16 閲覧

**Fig. 4** A sample of message describing the change of API specification

### 2.2.2 コミットメッセージ

次に，コミットメッセージについて検討する．図 3 は 2.1 節で示したテスト失敗の原因となる変更を含むコミットである．この例でコミットメッセージから読み取れることは，HTML5 という新仕様に jsoup が追従したという旨で，`parse()` に対する仕様変更の具体的な内容について言及していない．図 4 のようにコミットメッセージから API の動作変更が読み取れることもあるが，コミットの粒度やメッセージ内容はライブラリや開発者によって様々であり，任意のライブラリ，任意のアップデートで活用できるわけではない．

以上より，コミットメッセージから API 仕様の変更内容の把握ができないケースが多々存在すると考える．

### 2.2.3 処理コード差分

次に，API ボディ，及び API 内部で呼ばれている処理のソースコード差分を検討する．図 5 内上図は API ボディの差分，図 5 内下図はソースコード差分を精査し，ライブラリのソースコードを手動解析し得られた当該のアプリケーションテスト失敗を引き起こす API 処理の実装箇所である．ライブラリ内部の修正に携わっていないアプリ

*7 `https://spring.io/` 2022/6/16 閲覧
*8 `https://github.com/spring-projects/spring-boot/wiki/Spring-Boot-2.0-Migration-Guide` 2022/6/16 閲覧

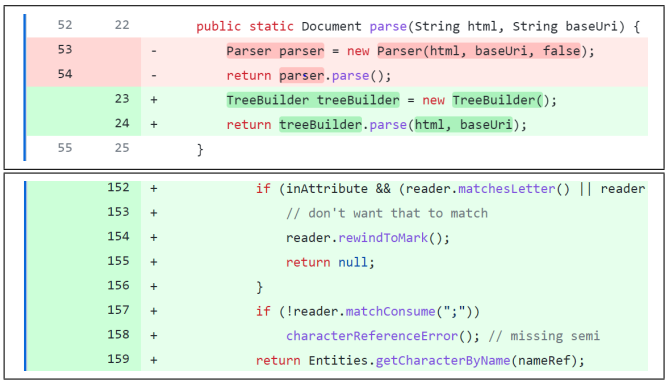


**図 5** API 内部のソースコード差分の例
org.jsoup.parser.Parser.java 及び
org.jsoup.parser.Tokenizer.java

**Fig. 5** A sample of source code differences inside the API

ケーション開発者にとって，仕様の変化を読み取ることは難しいと考えられる．加えて 2.2.1 項でも言及した通り，`parse()` 内部は HTML5 への追従が実施され，その実現のために数千行の追加実装が行われている．当該箇所を絞り込むことを含め，処理コード差分から API 仕様の変更内容を解釈することは難しいと思われる．

以上から，少なくとも今回の例のようなケースでは，処理コード差分から仕様の変更内容の把握することは困難である．

#### 2.2.4 ライブラリテスト差分

ここまで，リリースノート，コミットメッセージ，処理コード差分のそれぞれについて API 仕様の変更内容の把握への活用を検討したが，これらは機会や難易度の観点から API 仕様の変更内容を把握するのは困難であると考えられる．

そこで本稿では，ライブラリ API の仕様変更に関する情報として，アップデートに伴う差分があるライブラリテストに焦点を当てる．図 6 はアップデートによって生じた jsoup の unescape 処理に関するテストの差分である．ライブラリテストには API の仕様を検査する用途が存在し，かつその差分は API の動作の変化を反映している．この差分及びテストから，アプリケーション開発者は API 仕様の変更内容（すなわち，`parse()` 内部で呼び出される unescape 処理の変更）を読み取りうる．

本稿で扱うライブラリの仮定から，アプリケーション開発者が参照できるライブラリテストは存在するため，ライブラリテスト差分は任意のライブラリで利用が可能である．また，具体的な動作の変更が反映されているため，アプリケーション開発者でも仕様の変更内容の把握しやすいと考える．

以上からライブラリテスト差分は機会，難易度の観点で API 仕様の変更内容の把握に活用できる可能性があると思われる．

### 2.3 テスト特定支援技術検討の提案

1.3.2 項で述べた通り，ライブラリアップデートではアプリケーションテストの失敗原因となる仕様の変更内容の把握もアプリケーション開発者にとって関心事となる．また 2.2 節にて，その把握にアップデートに伴う差分があるライブラリテストが有効である可能性があることを提示した．一方で一つの API について，仕様・テストが複数存在している．そのため，アプリケーションテストの失敗と関連する仕様を検証している適切なライブラリテストを絞り込む必要がある．このとき，ライブラリテストの特定が困難である場合が存在する．例えば 2.1 節で示した例では，`parse()` を呼び出し，かつアップデートに伴う差分があるテストは 30 件も存在する．これらの各テストを 1 件ずつ見比べ，アプリテストの失敗との関連を調査することはアプリケーション開発者への大きな負担となる．

そのため仕様の変更内容の把握に有効なライブラリテストの特定を支援する技術を提案する．この支援技術を実現することにより，ライブラリアップデートで発生するデバッグ作業の稼働削減が可能となる．

## 3. 支援技術

本章では，ライブラリテスト特定を支援する技術を説明する．

これまでの前提から，テスト失敗の原因となる API 呼び出しは与えられているとする．各テストによって実行されるコード（以下，実行経路）は実行トレース等から取得できるとし，API 仕様の変更内容と対応する API 内部の部分的な実行経路（例えば図 5 下図と対応する範囲のコード）が存在すると仮定する．

特定したいライブラリテスト $Test_{related}$ は（失敗した）アプリケーションテスト $Test_{app}$ と共通の API 仕様変更の影響を受けている．検討手法のアイデアは，$Test_{app}$ と $Test_{related}$ は API 仕様の変更内容と対応する共通の部分的な実行経路を実行していることから，実行経路の類似性によって検出が可能であるという直感である．

Algorithm 1 はアプリケーションのテスト失敗と関連するライブラリテストを特定するアルゴリズムの疑似コードである．テスト失敗原因の API $api$，各テストの実行経路の情報を含むライブラリ情報 $lib$，（失敗した）アプリケーションテストの実行経路の情報を含むアプリ情報 $app$ が与えられるとする．このとき，$lib.Tests_{w/\ diff}$ はライブラリアップデートに伴うコード差分があるライブラリテスト群，$lib.Tests_{wo/\ diff}$ はライブラリアップデートに伴う差分がないライブラリテスト群とする．また，実行経路 $call_{api}$ はコード領域と対応する要素を持つ集合として表現可能であるとする．$union()$，$sub()$，$inter()$ は引数の集合に対する和・差・積集合を計算する関数とする．

API 仕様の変更内容と対応する API 内部の部分的な実

```
@Test public void strictAttributeUnescapes() {
    String html = "<a id=1 href='?foo=bar&mid&lt=true'>One</a> <a id=2 href='?foo=bar&lt;qux&lg=1'>Two</a>";
    Elements els = Jsoup.parse(html).select("a");
    assertEquals("?foo=bar&mid&lt=true", els.first().attr("href"));
    assertEquals("?foo=bar|&lt=true", els.first().attr("href")); // &mid gets to | because not tailed by =; lt is so not unescaped
    assertEquals("?foo=bar<qux&lg=1", els.last().attr("href"));
```

**図 6** アップデートに伴う差分があるライブラリテスト
（org.jsoup.parser.AttributeParseTest#strictAttributeUnescapes）
**Fig. 6** A example of library test difference

**Algorithm 1** Detect Related Tests

**Require:** $api, lib, Test_{app}$
1: $Tests_{w/} \leftarrow lib.Tests_{w/\ diff}$
2: $Tests_{wo/} \leftarrow lib.Tests_{wo/\ diff}$
3: $WhiteList \leftarrow union(call_{api}\ in\ Tests_{w/})$
4: **for** $call_{api,i}\ in\ Tests_{w/}$ **do**
5: $unique_i \leftarrow sub(call_{api,i}, WhiteList)$
6: **if** $inter(Test_{app}.call_{api}, unique_i) \neq null$ **then**
7: $Tests_{related} \leftarrow Tests_{related} + call_{api,i}.test$
8: **end if**
9: **end for**
10: **return** $Tests_{related}$

行経路は非自明である．かつ，二つの API 呼び出しの実行経路 $Test_{app}.call_{api}$ と $call_{api,i}$ の重複箇所には，アプリケーションのテスト失敗とは無関係の実行経路が多く含まれている．そこで，API 仕様の変更内容と対応する部分的な実行経路が存在する仮定から，アップデートに伴う差分がないライブラリテストで呼び出される API の実行経路は，アップデートによる API の動作変化と関係しないと見做せる．そこで，アップデートに伴う差分がないライブラリテスト群 $lib.Tests_{wo/\ diff}$ の API $api$ の実行経路の和を計算し，アプリケーションのテスト失敗と無関係の除外対象（以下，ホワイトリスト）とする（l.3, 図 7 White List）．アップデートに伴う差分があるライブラリテスト群 $lib.Tests_{w/\ diff}$ の API 呼び出しごとに，実行経路 $call_{api,i}$ からホワイトリストを除外した実行経路 $unique_i$ を計算する（l.5）．$unique_i$ には API 仕様の変更内容と対応する部分的な実行経路が包含される．このとき，アプリケーションテスト上での API 呼び出しの実行経路 $Test_{app}.call_{api}$ が，ある差分ありライブラリテストに属する $unique_i$ と重複する場合，$Test_{app}.call_{api}$ と $call_{api,i}$ は共通の API 仕様変更の影響を受けている可能性が高い．（l.6, 図 7 t1）．$Test_{app}.call_{api}$ と $unique_i$ が重複する場合，アプリケーションのテスト失敗と関連するテストとして出力する（l.7, l.10）．

検討手法はアプリケーションのテストとライブラリテストのでの API 呼び出しについて実行経路の計測のみで実現できる．また，$unique_i$ などの計算量は $O(|call_{api,i}|)$ であり，テストでは API 呼び出しが有限回であると見込まれるため，現実的な時間で計算可能である．

## 4. 検討技術の評価

3 章での検討手法について Java 言語を対象にツール実装を行った．Java 用の既存カバレッジ計測ツールである JaCoCo[10] を改変し，制御フローにおける分岐カバレッジ粒度の実行経路を取得した．

3 章での手法が有効なケースが存在するかを検証するために評価を実施した．検体として jsoup 1.5.1 から 1.6.0 へのアップデートを実施したサンプルアプリケーションを評価した．3 章でも述べたように前提として，エラー原因となるアプリテスト上の API 呼び出し `parse()` は特定済みであるとする．検体と実験結果の概略を表 1 に示す．# of tests は当該の API を呼び出すライブラリテストの数，w/ differnce はそのうち，アップデートに伴う差分があるライブラリテストの数である．

表 1 の Sample は 2.1 節で提示した， jsoup を用いた Web アプリケーションである．アップデートにより URL パラメータの文字列の置換ルールが変更され，出力される HTML 文字列が期待通りではなくなる．事前に手動で全テストを確認し，特定を期待するライブラリテストは，図 6 に示す strictAttributeUnescapes 一つであることを確認した，結果として，検討手法は前述のアルゴリズムから当該のテストの絞込みに成功した．

図 8 はホワイトリストの実行経路に strictAttributeUnescapes の実行経路を加えた場合の実行経路の差分（すなわち，strictAttributeUnescapes の $unique_i$）の一部を示している[*9]．ここは図 5 下図で説明した通り，アプリケーションのテスト失敗と関連するコード領域である．当該箇所を含めた $unique_i$ と対応する実行経路をアプリケーションのテストと strictAttributeUnescapes の両方ともが実行していることも実行トレースから確認した．

以上の評価を通して，アップデートに伴う差分がないライブラリテストの実行経路を除外することで，$unique$ に API 仕様の変更内容を引き起こす API 内部の部分的な実行経路が包含され，検討したアルゴリズムによってアプリケーションのテスト失敗と関連するライブラリテストの特

[*9] 図 8 は JaCoCo のカバレッジレポート機能を利用して可視化している．緑色の箇所は実行され，赤色の箇所は実行されなかった（到達しなかった）範囲を示し，黄色い箇所は全分岐の内実行されていない分岐が存在することを示している．

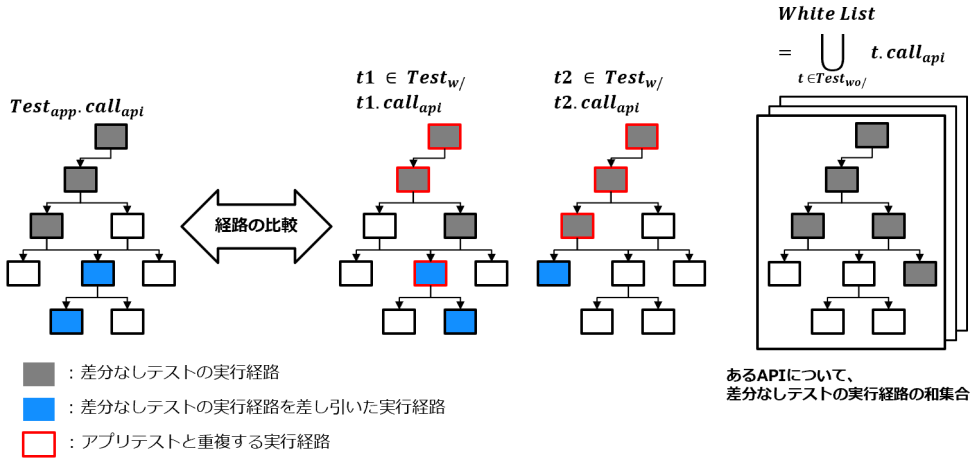


**図 7** アルゴリズムイメージ
**Fig. 7** Image of algorithm 1

| Application | Library<br>api name | versions<br># of tests (w/ differnce) | # of detected tests | specification |
|---|---|---|---|---|
| Sample | Jsoup<br>Jsoup.parse() | 1.5.1 → 1.6.0<br>198 (30) | 1 | unescaping |

**表 1** 評価結果概略
**Table 1** Evaluation result

```
152.          if (inAttribute && (reader.matchesLetter() || reader.matchesDigit() || reader.matches('=') )) {
153.              // don't want that to match
154.              reader.rewindToMark();
155.              return null;
156.          }
157.          if (!reader.matchConsume(";"))
158.              characterReferenceError(); // missing semi
159.          return Entities.getCharacterByName(nameRef);
```

```
152.          if (inAttribute && (reader.matchesLetter() || reader.matchesDigit() || reader.matches('=') )) {
153.              // don't want that to match
154.              reader.rewindToMark();
155.              return null;
156.          }
157.          if (!reader.matchConsume(";"))
158.              characterReferenceError(); // missing semi
159.          return Entities.getCharacterByName(nameRef);
```

**図 8** 検出された $unique_i$ の一部
org.jsoup.parser.Tokenizer.java
**Fig. 8** A part of detected $unique_i$

定が可能であることを確認した．

## 5. 関連研究

### 5.1 デバッグ支援技術

デバッグ支援の一環としてテスト失敗原因を特定することを目的とした既存手法が存在する．

（Spectrum-based）Fault Localization[6], [7] は失敗原因となるソースコード位置（コード行）を特定することを目的とした手法である．詳細は 1.4 節で述べた通りである．

Version Control System の普及に伴い，テスト失敗を導入するコード差分を持つコミットを特定するというアプローチ [11], [12], [13] も検討されている．例えば Git Bisect[11] は git が提供する，テスト失敗を導入するコミット[*10]の特定技術である．SZZ[12], [13] はバグを修正するコミットを識別し，バグを導入するコミットのパターンを検出する技術である．これらの技術はアプリケーション開発者の誤った修正に起因するテスト失敗の原因特定に役立つ手法である．

既存研究はテスト失敗の原因となるコード修正を実施するのがアプリケーション開発者であることを前提としている．そのためソースコード位置の特定が関心事である一方で，テスト失敗の原因となる仕様の変更内容の特定は焦点となっていない．

### 5.2 テスト特定技術

以下に，特定のテストコードを検出する，という方法論に関する関連研究を提示する．

Test Suite Reduction[14], [15] は，アプリケーションのテストスイート最適化を目的とした研究分野である．実行経路情報などを計測することで冗長なテストを検出する．正常に終了するテスト群を対象とすることから，テストが失敗することを想定しておらず，デバッグ作業の文脈での活用には適さない．

Test Suite Selection[16], [17] は，特定の項目に関連するテストスイートを選別することを目的とした研究分野である．例えば修正したソースコード箇所を実行するテストを Call Graph などから絞り込むことで，回帰テストのコストを軽減する．単一のプロジェクトでの（すなわちアプリケーションに閉じた）利用が前提であり，ライブラリまでを跨いでの適用は難しい．

Code Search[18], [19] は，コードの集合から特定のコー

[*10] Bug/Error Introducing Commit．テスト失敗が発生するようになった（時系列的に最も早い）コミット．

ドを検索する研究分野である．キーワード検索や全文検索といったパターンマッチ [18] から，コードの意味論を検索する手法 [19] が提示されている．意味論を用いた手法では，コード全体が類似したテストコード同士を紐づけることが期待できる．ただし，テスト失敗の原因把握で対象となるのはアプリケーションのテストコードとライブラリのテストコードであるため，それぞれが検証するテスト項目は異なる．そのため，意味論検索によるテストそのものの類似性比較は，アプリケーションのテスト失敗と関連する他のテストの検出に用いるのは困難である．

## 6. Conclusion

アプリケーション保守におけるテスト失敗について，失敗原因の把握にはソースコード位置と仕様の変更内容の二点が把握が必要である．本稿では，ライブラリアップデートでは後者の把握が困難な一方，既存研究では支援できていないことを示した．ライブラリアップデートで仕様の変更内容の把握に活用可能な情報を整理した結果，アップデートに伴う差分を持つライブラリテストが有効である可能性があることを示した．また，テスト失敗原因の把握に有効なライブラリテストの特定を支援する技術の検討を提案し，支援技術の一例を実装した．例示した技術は簡易評価を通して実際にライブラリテストを特定できる事例があることを示した．

## 7. 参考文献

### 参考文献

[1] Synopsys: *2020 Open Source Security and Risk Analysis report*, Synopsys (2020).
[2] Wang, Y., Chen, B., Huang, K., Shi, B., Xu, C., Peng, X., Wu, Y. and Liu, Y.: An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects, *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 35–45 (online), DOI: 10.1109/ICSME46990.2020.00014 (2020).
[3] Nir, D., Tyszberowicz, S. and Yehudai, A.: Locating Regression Bugs, *Proceedings of the 3rd International Haifa Verification Conference on Hardware and Software: Verification and Testing*, HVC'07, Berlin, Heidelberg, Springer-Verlag, p. 218–234 (2007).
[4] Mostafa, S., Rodriguez, R. and Wang, X.: Experience Paper: A Study on Behavioral Backward Incompatibilities of Java Software Libraries, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, New York, NY, USA, Association for Computing Machinery, p. 215–225 (online), DOI: 10.1145/3092703.3092721 (2017).
[5] Chen, L., Hassan, F., Wang, X. and Zhang, L.: Taming Behavioral Backward Incompatibilities via Cross-Project Testing and Analysis, *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, New York, NY, USA, Association for Computing Machinery, p. 112–124 (online), DOI: 10.1145/3377811.3380436 (2020).
[6] Wong, W., Gao, R., Li, Y., Abreu, R. and Wotawa, F.: A Survey on Software Fault Localization, *IEEE Transactions on Software Engineering*, Vol. 42, pp. 1–1 (online), DOI: 10.1109/TSE.2016.2521368 (2016).
[7] Goues, C. L., Nguyen, T., Forrest, S. and Weimer, W.: GenProg: A Generic Method for Automatic Software Repair, *IEEE Transactions on Software Engineering*, Vol. 38, pp. 54–72 (2012).
[8] 一真嶋利，隆　石尾，克郎井上：ライブラリのバージョン更新支援のための実行トレースからのテストケース生成，ウィンターワークショップ 2018・イン・宮島論文集， No. 2018, pp. 22–23（オンライン），入手先 ⟨https://ci.nii.ac.jp/naid/170000176273/⟩ (2018).
[9] Apiwattanapong, T., Orso, A. and Harrold, M. J.: JDiff: A differencing technique and tool for object-oriented programs, *Automated Software Engineering Journal*, Vol. 14, pp. 3–36 (2007).
[10] Contributors, E. et al.: JaCoCo, `https://www.eclemma.org/jacoco/` (2006).
[11] Conservancy, S. F.: Git, `https://git-scm.com/docs/git-bisect` (2009).
[12] undefinedliwerski, J., Zimmermann, T. and Zeller, A.: When Do Changes Induce Fixes?, *Proceedings of the 2005 International Workshop on Mining Software Repositories*, MSR '05, New York, NY, USA, Association for Computing Machinery, p. 1–5 (online), DOI: 10.1145/1083142.1083147 (2005).
[13] Rodríguez-Pérez, G., Robles, G., Serebrenik, A., Zaidman, A., Germán, D. and Gonzalez-Barahona, J.: How bugs are born: a model to identify how bugs are introduced in software components, *Empirical Software Engineering*, Vol. 25, No. 2, pp. 1294–1340 (online), DOI: 10.1007/s10664-019-09781-y (2020).
[14] Chen, Y., Probert, R. L. and Ural, H.: Regression Test Suite Reduction Using Extended Dependence Analysis, *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SOQUA '07, New York, NY, USA, Association for Computing Machinery, p. 62–69 (online), DOI: 10.1145/1295074.1295086 (2007).
[15] Cruciani, E., Miranda, B., Verdecchia, R. and Bertolino, A.: Scalable Approaches for Test Suite Reduction, *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pp. 419–429 (online), DOI: 10.1109/ICSE.2019.00055 (2019).
[16] Pasala, A., Lew Yaw Fung, Y., Akladios, F., Appala Raju, G. and Gorthi, R.: Selection of Regression Test Suite to Validate Software Applications upon Deployment of Upgrades, *19th Australian Conference on Software Engineering (aswec 2008)*, pp. 130–138 (online), DOI: 10.1109/ASWEC.2008.4483201 (2008).
[17] Legunsen, O., Shi, A. and Marinov, D.: STARTS: STAtic regression test selection, *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 949–954 (online), DOI: 10.1109/ASE.2017.8115710 (2017).
[18] GitHub, I.: GitHub Advanced Search, `https://github.com/search/advanced` (2021).
[19] Husain, H., Wu, H., Gazit, T., Allamanis, M. and Brockschmidt, M.: CodeSearchNet Challenge: Evaluating the State of Semantic Code Search, *CoRR*, Vol. abs/1909.09436 (online), available from ⟨http://arxiv.org/abs/1909.09436⟩ (2019).