Vol.2022-SE-212 No.7
2022/12/10

コールグラフ差分を利用したライブラリアップデート影響解析

小泉 雄太[1,a)]　山口 大輔[1,b)]

概要：ソフトウェア開発においてライブラリは広く利用されており，アップデートによってライブラリを最新に追従させることは重要とされている．しかし，アップデートによるソフトウェアの動作変化の把握の難しさからライブラリアップデートは実施されないことが多い．本発表では，ライブラリアップデート前後のコールグラフの差分を取得することでアップデートによって動作変化が潜在的に生じうるプログラムの範囲を示す影響解析手法を提案する．小規模サンプルソフトウェアを対象に実験評価を行い本手法について議論する．

キーワード：インパクト解析，コールグラフ，ライブラリアップデート

An Impact Analysis for Library Update with Call-graph Differencing

Yuta Koizumi[1,a)]　Daisuke Yamaguchi[1,b)]

Abstract: Libraries are widely used in software development, and it is important to keep the libraries up-to-date through updates. However, library updates are often not implemented due to the difficulty of understanding changes in software behavior due to updates. In this paper, we propose an impact analysis method that shows the range of programs that can potentially cause behavioral changes due to library updates by obtaining differences in call graphs before and after library updates. Experimental evaluation for small sample software is performed and this method is discussed.

Keywords: Impact analysis, Call-graph, Library update

1. はじめに

ソフトウェア開発においてライブラリは生産性や品質向上のために広く利用されている [6,7,11,15]．ライブラリは脆弱性やバグの修正，機能向上がなされた新しいバージョンの公開がなされてゆくことから，ライブラリ利用者には自身が利用しているライブラリをアップデートする重要性が強調されている [14]．しかし，アップデートのよるソフトウェアの動作変化の把握の難しさからライブラリアップデートは実施されないことが多い [12,19]．

一般にプログラムに変更が加わったとき，そのプログラムの変更が変更箇所とは異なる箇所に影響することによって開発者が予期しない動作が引き起こされることがある．そのため，プログラムへ変更を加える場合は変更による影響や変更後のプログラムの動作を把握することは重要である．これはライブラリアップデートの場合も同様である．ライブラリアップデートはライブラリのプログラムに変更を加えることに相当するためである．

変更影響解析 (*change impact analysis*, IA) はソフトウェアへの変更がもたらす潜在的な結果を特定するプロセスである [1,17]．近年の IA に関する研究の多くはプログラムに変更が加わることで動作変化が潜在的に生じるプログラムの自動特定に焦点が当てられており，盛んに研究されている領域の一つとなっている．中でも，静的依存分析に基づく IA や動的実行情報分析に基づく IA [5,10] は中心的

[1] 日本電信電話株式会社
NTT Corporation, Minato-ku, Tokyo 108–0075, Japan
[a)] yuuta.koizumi.hr@hco.ntt.co.jp
[b)] daisuke.yamaguchi.be@hco.ntt.co.jp

なアプローチである．静的依存分析に基づく IA はコールグラフ，制御フローグラフ，依存関係グラフなどのソースコードから派生したプログラムのグラフ表現に対して到達可能性判定を行うことで動作変化が起こりうるプログラムを特定する [2]．このアプローチはプログラムのすべての可能な実行を考慮することができる反面，プログラムのグラフ表現は保守的にプログラムを近似することから偽陽性が多い傾向にあることが知られている．対照的に，動的実行情報分析に基づく IA はプログラムの実行情報 (e.g., 実行トレース [9]) に基づいて判定をするため偽陽性は少ない反面，プログラムの実行に用いられる入力値に依存するため偽陰性が多い，すなわち，網羅的に動作変化が生じうるプログラムを特定することが難しい．そのため，これらの分析を組み合わせたアプローチ [2] も提案されており，より高い精度が達成されている．

IA をライブラリのプログラム変更による影響を判定するために応用した手法もまた提案されている．例えば，Eclipse Steady [13] はライブラリの脆弱性修正パッチに着目し，修正パッチで行われたライブラリのプログラムの変更箇所を脆弱性の原因プログラムとみなし，ライブラリのプログラムの変更箇所までのコールグラフを解析することで潜在的にソフトウェアが脆弱性の原因プログラムを実行する可能性があるか判定する．実行する可能性がなければソフトウェア全体として脆弱性の影響はないことが分かる．

しかし，ライブラリのアップデートによるライブラリプログラムの変更は一般にコールグラフ上到達不能であるプログラムの範囲へ動作変化を与えるものが存在するため，コールグラフの到達可能性判定は直裁ではない．ライブラリアップデートでなされる典型的なライブラリのプログラムの変更の中には，ライブラリのプログラムの変更箇所へのコールグラフの解析でのソフトウェアの動作影響の判定を困難にするものがある．Dig らの調査 [3] によれば，pulled-up method (pushed-down method) と呼ばれるプログラムの変更はライブラリアップデートではしばしば行われる．これらは，メソッドの継承機構を利用したリファクタリングに用いられるが，結果的にメソッドのコールサイト（呼び出し側のコード）を書き換えずに，実行時に呼び出されるプログラムに変更が生じる．すなわちこれらは，ソフトウェアのエントリポイントからコールグラフの解析によって実行され得ないと判ったプログラムの変更が実際にはソフトウェア全体の動作に変化が及ぶ可能性があることを示唆している．

本研究では，ライブラリアップデートの文脈において，コールグラフ上到達するライブラリ変更箇所だけでなく，アップデート前後のコールグラフの差分を取得することでアップデートによって動作変化が潜在的に生じうるプログラムの範囲も示す影響解析手法を提案する．

提案手法は，ライブラリアップデートによって生じたライブラリのプログラムの変更について影響解析を行うことで，ライブラリアップデート後のソフトウェアの正常動作確認を支援する技術である．影響解析はソフトウェアのエントリポイントを開始メソッドとしたコールグラフ解析によるアップデート変更への到達判定と，ライブラリアップデート前のソフトウェアとライブラリアップデート後のソフトウェアのそれぞれのコールグラフから取得したコールグラフの差分取得の結果を併用する．影響解析の結果はソフトウェアへの影響のありうるライブラリのコード差分とコールグラフ差分の一覧によって示される．出力された両一覧が空であるならば，すなわち，ライブラリのアップデートに伴うコード変更に対し，ソフトウェアから実行される可能性のあるものが存在しないことを意味するため，ライブラリアップデート後もソフトウェアは正常に動作することが保証される．逆に，出力された一覧が空でないならば，一覧に含まれるコード差分とコールグラフ差分によってライブラリアップデート後のソフトウェアの動作が正常でなくなる可能性がある．

提案技術を用いることで，ライブラリのアップデートによって発生するソフトウェア全体への動作影響の可能性を，pulled-up method (pushed-down method) と呼ばれるプログラムの変更による動作影響の可能性も含めて，判定することが可能になる．

我々は提案手法を Java 向けに影響解析ツールを実装し，pulled-up method (pushed-down method) を伴うプログラム変更を含むサンプルソフトウェア，サンプルライブラリを対象に評価を行った．結果，サンプルソフトウェアから到達するコード差分に加えて，既存技術では見落としてしまう pulled-up method (pushed-down method) と対応するコールグラフ差分を判定，提示できることを確認した．

2. 既存技術が見落とすアップデート影響

本節では, 既存技術が見落としてしまう pulled-up method (pushed-down method) を伴うプログラム変更を例示し，ソフトウェア開発者へ提示すべき情報について説明する．

図 1 は pulled-up method (pushed-down method) を伴うプログラム変更例である．図 1 上のコードはアップデートに伴い変更されたライブラリ，図 1 下はそのライブラリを利用しているソフトウェアのソースコードを模した例である．色掛けされた行はプログラム変更を表現しており，黄色の色掛けは変更によって編集された箇所がある行，緑色の色掛けは変更によって追加された行を示している．

図 2 は `App.main(String[])` をエントリとしたコールグラフで，コールグラフ上の緑色の色掛けがされたメソッドノードはアップデートに伴いソースコード上で追加されたノード，黄色の色掛けがされたノードはアップデートに伴う変更行が存在するメソッドノードを示す．既存技術は図 2(a) のように，ソフトウェアのメソッドをエント

リとしたコールグラフ上の到達判定により，変更された箇所を持つ Human.<init>(...)[*1]と，新しく追加された Human.getSex() のコード差分へ到達することを判定，提示する．しかし，既存技術ではプログラム変更に伴い発生している push-down method の情報を一部提示できない．以降，あるクラス C に属するメソッド A(args) が他のメソッド B(args) が呼び出す関係を C.A(args)-->B(args) と表現する．図 1 の例では，ソフトウェア 8 行目の呼び出し関係 App.main(String[])-->getSex() において，getSex() の実体がアップデート前は Human クラスの継承元である Hominina クラスのメソッド Hominina.getSex() が呼び出されているのに対し，アップデート後はオーバーライドメソッドの追加により，Human クラスのメソッド Human.getSex() が呼び出されている．既存技術は App.main(String[])-->getSex() の呼び出し実体が変化したこと見落としてしまい，アップデートに伴うアプリケーションの退行発生の原因の追跡を困難にする要因となりえてしまう．

我々の提案手法が目指すのは，図 2(b) のように，到達するコード差分（すなわち，Human.<init>(...)，Human.getSex()）だけでなく，pulled-up method (pushed-down method) に対応するコールグラフ差分（すなわち，図 2(b) 内の水色の破線と実線で示される，deleted: App.main(String[])-->Hominina.getSex()，added: App.main(String[])-->Human.getSex()）も含めてライブラリアップデートに伴う動作影響の可能性を提示することである．

3. 提案手法

本節では，2 節で示した pulled-up method (pushed-down method) の判定，提示機能を兼ね備えた手法について解説していく．

まずは pulled-up method (pushed-down method) が持つ性質を列挙し (3.1 項)，その性質を判定する手法概要を述べ (3.2 項)，最後に手法を形式的に定義する (3.3, 3.4 項)．

3.1 pulled-up method (pushed-down method) の性質

pulled-up method (pushed-down method) の持つ性質としては，以下が存在する．

(1) 呼び出し関係 C.A(args)-->B(args) について，アップデート前後で同一のメソッドシグネチャ B(args) を呼び出しているが，呼び出される実体が変化している．

(2) 呼び出し関係 C.A(args)-->B(args) について，アップデート前後で実体が異なる B(args) が属するクラス同士にはクラス階層構造が存在する．すなわち，片方のクラスはもう片方のクラスの先祖または子孫クラスである．

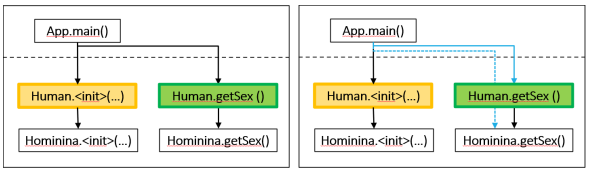
(3) 呼び出し関係 C.A(args)-->B(args) について，コールサイトにはコード差分がない．

[*1] C.<init>(args) はあるクラス C のコンストラクタを指す．

```java
5  public class Human extends Hominina{
6      public Human(Double height, Double weight, int age,
7          super(height, weight, age+1, sex);
8          System.out.println("this is Human.");
9      }
10
11     @Override
12     public boolean isOverWeight(){
13         return this.getBMI() > 25;
14     }
15
16     @Override
17     public String getSex(){
18         return super.getSex();
19     }
20 }
```

```java
3  import library.l1.Human;
4
5  public class App {
       Run | Debug
6      public static void main(String[] args) {
7          Human man = new Human(170.2, 60.0, 20, "Male");
8          System.out.println(man.getSex());
9          return;
10     }
11 }
```

図 1: pushed-down method の例



(a) 既存ツールの提示情報　(b) 本稿が提示したい情報

図 2: 判定結果の比較

3.2 手法概要

3.1 項で示した pulled-up method (pushed-down method) の性質を判定するために，提案手法は，変更影響解析を行う際に，コールサイトを用いたコード差分到達判定だけでなく，アップデート前後それぞれのライブラリソースコードを用いた新旧コールグラフを作成及び比較を実施する．

性質 (1) について，新旧コールグラフの差分を計算することで判定を行う．図 4 は図 1 の App.main(String[]) をエントリとした新旧のコールグラフである．図 1 の例では，新旧コールグラフの比較によって，呼び出し関係 App.main(String[])-->Hominina.getSex() が削除され，呼び出し関係 App.main(String[])-->Human.getSex()，

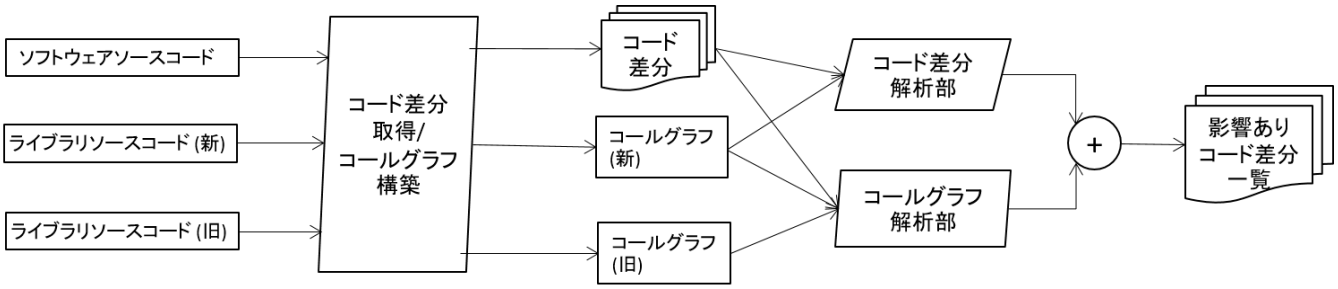


図 3: 提案手法の処理フロー

`Human.getSex()-->Hominina.getSex()` が追加されていることが分かる．このとき，`App.main(String[])-->Hominina.getSex()` と `App.main(String[])-->Human.getSex()` について，コールサイトの一致及び同一のメソッドシグネチャ `getSex()` を呼び出しているが，呼び出し先の実体が異なる性質（1）を満たしていることが検出される．

性質（2）について，ソースコード差分解析時，性質（1）を満たす追加・削除された呼び出し関係のペアから，`Human` クラスと `Hominina` クラスがクラス階層構造になっているかを判定する．図 1 ライブラリコードの 5 行目から，`Human` クラスが `Hominina` の子クラスであり，クラス階層構造関係になっていることがわかる．このとき，`App.main(String[])-->Hominina.getSex()` と `App.main(String[])-->Human.getSex()` は性質（2）を満たしていることが検出される．

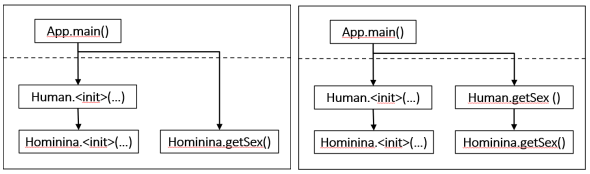
性質（3）についてソースコード差分情報から，`App.main(String[])` 上のコールサイト `getSex();` に変更がない[*2]ことが分かる．このとき，`App.main(String[])-->Hominina.getSex()` と `App.main(String[])-->Human.getSex()` は性質（3）を満たしていることが検出される．

以上より，新旧コールグラフ上に存在する呼び出し関係のペア `App.main(String[])-->Hominina.getSex()` と `App.main(String[])-->Human.getSex()` は pulled-up method (pushed-down method) の性質（1–3）を満たすため，pulled-up method (pushed-down method) として検出される．なお, 図 1 のライブラリソースコード 11 – 14 行目で追加されている `isOverweight()` も pulled-up method (pushed-down method) であるが，アプリケーションからは到達しないため提案手法では検出されない．

3.3 形式的定義

以下では，前述の手法概念を形式的に定義する．図 3 は提案手法の処理フローを示している．

[*2] なお，`App.main()` を含むアプリケーションのソースコードはライブラリアップデートの変更影響解析時点で変更されていない前提である．



(a) 旧コールグラフ　(b) 新コールグラフ

図 4: 新旧コールグラフ

対象言語はオブジェクト指向プログラミング言語とし，図 5 に示す抽象構文を有すものとする．

クラス C	::=	`class` C `extend` D $\{cns\ \overline{m}\}$
コンストラクタ cns	::=	$C(\overline{C\ x})\{\overline{\sigma};\}$
メソッド m	::=	$\tau\ m(\overline{C\ x})\{\overline{\sigma};\}$

図 5: 対象言語 ($\bar{t}$ は 式 t の 0 回以上の繰り返しを表す．)

クラス C にはクラス名 C の宣言部と，継承先クラス（親クラス）D の宣言部がある．また，クラス本体の定義はコンストラクタ cns とメソッド m の定義から成る．ただし，τ は型を表すメタ記号，x は変数を表すメタ記号，σ は文を表すメタ記号である．

ソースコード差分はクラス名と API 名の組 (API の完全修飾子に相当) とする．ライブラリのアップデートで生ずるソースコード差分全体を以下のコード差分のリスト Δ として扱う．

コード差分のリスト Δ	::=	$\delta\ \Delta$
コード差分 δ	::=	$(C, name)$
API 名 $name$	::=	$m \mid cons$

プログラムのコールグラフは，3 つ組 (S, S_0, R) で定義される．ただし，S はメソッド，コンストラクタの集合，S_0 エントリポイントの集合である．R は遷移関係であり，$R \subseteq S \times S$ である．

ここで $s \in S$ プログラムのメソッド，(またはコンストラクタ) であるため，s についてプログラム上のクラス名及びメソッド名 (コンストラクタ名) が一意に定義される．s に対応するメソッド名 (コンストラクタ名) を取得する関数を getQName(s) として予め定義しておく．

定義 1 (getQName)

S はコールグラフのメソッド，コンストラクタの集合とし，$s \in S$ とする．このとき，

$\mathrm{getQName}(s) = (C_s, n_s) \iff s$ のクラス名，メソッド名 (コンストラクタ名) はそれぞれ C_s, n_s である．

また，以下を定義しておく．

定義 2 (1 ステップ遷移)

$s_1, s_2 \in S, r \in R$ とするとき，$r = (s_1, s_2)$ を $s_2 = r(s_1)$，または $s_1 = r^{-1}(s_2)$ と書くことにする．

影響ありコード差分一覧はソースコード差分のリストに内包される．

3.4 アルゴリズム

疑似コードのコード幅を短くする目的で，以下では「ソフトウェア」を「ソフト」と略記する．

トップレベル

提案手法の全体のフローを疑似コード表 1 に示す．S000 から S003 は入力データの前処理であり，コード差分のリスト，コールグラフの構築をする．S004 から S005 は図 3 のコード差分解析部とコールグラフ解析部に対応し，後述するように，コールグラフ解析部では pulled-up method (pushed-down method) の判定結果を出力する．最終的に，コード差分解析の結果（通常の差分到達解析）とコールグラフ解析の結果の両方を出力する．

コード差分解析処理

コード差分解析はコード差分 Δ について，コールグラフ上存在するメソッド，コンストラクタを残すようにフィルタリングを行う．コールグラフを $\mathcal{G} = (S_\mathcal{G}, S_{\mathcal{G}0}, R_\mathcal{G})$ とするとき，コード差分解析の結果は以下の式 $\Delta_\mathcal{G}$ で与えられる．

$$\Delta_\mathcal{G} = \{(C, name) \in \Delta \mid s \in S_\mathcal{G} \wedge (C, name) = \mathrm{getQName}(s)\}$$

ただし，getQName の定義は定義 1 に従う．

コールグラフ解析処理

処理部 S005 のコールグラフ解析処理について，疑似コード表 2 に示す．

ことのき，$\mathcal{G}_N$ はライブラリアップデート後のソフトのコールグラフであり，$\mathcal{G}_N = (S_{\mathcal{G}_N}, S_{N0}, R_{\mathcal{G}_N})$ であるものする．また，$\mathcal{G}_O$ はライブラリアップデート前のソフトのコールグラフであり，$\mathcal{G}_O = (S_{\mathcal{G}_O}, S_{O0}, R_{\mathcal{G}_O})$ であるものする．

Fold の処理定義

発明技術では，コールグラフ間で新しいメソッドまたは，コンストラクタの追加があるとき，この変更はコールサイト（呼び出し側のプログラム）の変更に起因するものだとして扱う．pulled-up method 及び，pushed-down method ではコールサイトには変更が見られないが，実行時に呼び出されるメソッドが変更されることから，コールサイトに変更があるものとみなす．

Fold の計算結果は，コールグラフ間で新しいメソッドまたは，コンストラクタの追加があるとき，その要因のコールサイトのメソッドまたは，コンストラクタを発見し返す．一般にコールサイトは複数存在するので，Fold は計算結果を集合として返す．Fold はコールグラフ上のメソッドまたはコンストラクタ，メモ用の集合を受け取り，メソッドまたはコンストラクタからなる集合を返す関数である．Fold は以下のように再帰的に定義されており，コールグラフを辿るようにして処理が進む．コールグラフの遷移関係 R は循環を許すため，Fold は停止せずコールグラフを辿り続ける可能性がある．そこで，メモに一度辿ったメソッドまたはコンストラクタを記録し，一度辿ったことのあるメソッドまたは，コンストラクタを再訪したことを検知することで，Fold がコールグラフを辿り続けて停止しないことを回避させる．

$$\mathsf{Fold}(s, M) = \begin{cases} \emptyset & (\textbf{if } s \in M) \\ \{getQName(s)\} & (\textbf{if } s \in S_{\mathcal{G}_O}) \\ \bigcup_{s' \in \{r^{-1}(s) \mid r \in R_{\mathcal{G}_N}\}} \mathsf{Fold}(s', M \cup \{s\}) & (\textbf{otherwise}) \end{cases}$$

ただし，$\mathcal{G}_O$ はライブラリアップデート前のソフトのコールグラフであり，$\mathcal{G}_O = (S_{\mathcal{G}_O}, S_{O0}, R_{\mathcal{G}_O})$ であるものする．

4. 実験

4.1 実装

3 節で示した判定手法について，pulled-up method (pushed-down method) が発生し得るプログラミング言語である Java を対象とした解析ツールを実装し，図 1 のサンプルを対象として検出が可能かの簡易評価を実施した．実装に際し，コールグラフ作成には Java の静的解析フレームワークである WALA [18] を，ソースコード差分解析には構文差分の解析フレームワークである gumtree diff [4] を利用した．

4.2 結果

図 1 のサンプルに対してツールを適用したときの出力を図 6 に示す．サンプル検体では，新旧コールグラフから呼び出し実体の異なる呼び出し関係 `App.main(String[])-->Hominina.getSex()` と `App.main(String[])-->Human.getSex()` を抽出し，このコールグラフ差分が pulled-up method (pushed-down method) の性質を満たすこと（すなわち，アップデート前後で同じコールサイトであるが，呼び出される `getSex()` 実体が異なること，`getSex()` が属する二つのクラス `Human` と `Hominina` がクラス階層構造の関係にあること，コールサイトである `App.main(String[])` 上の (getSex();) にコード差分がないこと）を機械的に判定することに成功した．

処理:	トップレベル	
入力:	新版のライブラリソースコード $\mathcal{L}_N$	
入力:	旧版のライブラリソースコード $\mathcal{L}_O$	
入力:	ソフトウェアソースコード $\mathcal{A}$	
出力:	影響ありコード差分一覧 $\Delta_{\mathcal{F}}$	
S000	$(P_N, P_O, P_{\mathcal{A}}) \leftarrow \text{Parse}(\mathcal{L}_N, \mathcal{L}_O, \mathcal{A})$	入力されたソースコードを構文解析する
S001	$\Delta \leftarrow \text{ASTDiff}(P_N, P_O)$	AST の差分を取得しコード差分のリストの取得 (注 1)
S002	$\mathcal{G}_O \leftarrow \text{BuildCallGraph}(\mathcal{A}, P_O)$	ライブラリアップデート前のソフトのコールグラフの構築 (注 2)
S003	$\mathcal{G}_N \leftarrow \text{BuildCallGraph}(\mathcal{A}, P_N)$	ライブラリアップデート後のソフトのコールグラフの構築 (注 2)
S004	$\Delta_S \leftarrow$ コード差分解析 $(\Delta, \mathcal{G}_N)$	
S005	$\Delta_{\mathcal{G}} \leftarrow$ コールグラフ差分解析 $(\Delta, \mathcal{G}_O, \mathcal{G}_N)$	
S006	$\Delta_{\mathcal{F}} \leftarrow \Delta_S \cup \Delta_{\mathcal{G}}$	コード差分一覧の集合和を求める
S007	**return** $\Delta_{\mathcal{F}}$	

表 1: 提案手法の全体処理
注 1 AST (Abstract Syntax Tree) の差分取得には gumtree diff [4] などのツールを利用することができる.
注 2 コールグラフの構築は WALA [18] や Soot [8] などのツールを利用することができる.

処理:	コールグラフ解析	
入力:	コード差分 Δ	
入力:	ライブラリアップデート前のソフトのコールグラフ $\mathcal{G}_O$	
入力:	ライブラリアップデート後のソフトのコールグラフ $\mathcal{G}_N$	
出力:	コード差分 $\Delta_{\mathcal{G}}$	
S100	$\Sigma^+_{\mathcal{G}} \leftarrow \{r \in R_{\mathcal{G}_N} \mid r \notin R_{\mathcal{G}_O}\}$	$\mathcal{G}_N$ に追加された遷移関係を抽出
S101	$\Theta^+_{\mathcal{G}} \leftarrow \{s \in S_{\mathcal{G}_N} \mid s \notin S_{\mathcal{G}_O}\}$	$\mathcal{G}_N$ に追加されたメソッドまたはコンストラクタを抽出
S102	$\Delta_\omega \leftarrow \{(C,n) \in \Delta \mid \exists s \in \Theta^+_{\mathcal{G}}, (C,n) = \text{getQName}(s)\}$	$\Theta^+_{\mathcal{G}}$ に含まれるコード差分を抽出
S103	$\Delta_{fold} \leftarrow \bigcup_{(s_0,s_1)\in\Sigma^+_{\mathcal{G}}} \mathsf{Fold}(s_1, \emptyset)$	$\Sigma^+_{\mathcal{G}}$ の要素 $r = (s_0, s_1)$ について Fold を適用した結果を足し合わせる
S104	$\Delta_{\mathcal{G}} \leftarrow \Delta_\omega \cup \Delta_{fold}$	
S105	return $\Delta_{\mathcal{G}}$	

表 2: コールグラフ解析処理

4.3 考察，制約

実装ツールの WALA [18] は高度な context-sensitive コールグラフを構築できるが，解析対象のソフトウェア（ライブラリ含む）の規模が巨大になると context-sensitive コールグラフ計算は現実的な時間では終了しない場合がある．context-insensitive コールグラフ構築へ切り替えることでこの問題は回避できるが，その場合コールグラフの精度が下がり，実際には呼ばれない呼び出し関係が計測される可能性が発生する．また，リフレクションなどの動的に決定する要素には対応しきれておらず，こちらについてはライブラリテスト実行などで実際の呼び出し関係を補強する心積もりである．

5. 関連研究

5.1 変更解析

ソフトウェアへの破壊的変更が静的 [3,16]，あるいは動的な手法 [12] で分類されてきた．こういった分類調査はライブラリアップデートの文脈でも非常に有益な情報源となってきた．一方で，ソフトウェア（ライブラリのクライアント）への影響という点では，ライブラリ内部のコード変更に対する調査に留まっている．つまり，ソフトウェアとライブラリの関係まで深堀し，ライブラリのアップデートに伴う変更が実際にソフトウェアに影響するのか否かという調査はなされてこなかった．

我々の技術はソフトウェアとライブラリの両方を対象に解析するため，実ソフトウェアと実ライブラリを対象に多くの検体で調査ができれば，これらの研究からは得られなかった知見が得られる可能性が高い．

5.2 変更影響解析

変更影響解析 (*change impact analysis*, IA) は 1 節でも紹介した通り，ソフトウェアへの変更がもたらす潜在的な結果を特定するプロセス [1,17] や技術群 [2,5,10,13] である．1 節でも述べた通り，既存の変更影響解析技術は

```
deleted edge: sample.s6.uselib.App.main([Ljava/lang/String;) --> library.l1.Hominina.getSex()
  added edge: sample.s6.uselib.App.main([Ljava/lang/String;) --> library.l1.Human.getSex()
```

図 6: ツールの出力

pulled-up method (pushed-down method) のようなライブラリアップデートに伴う変更の影響を見落としてしまう可能性があり，網羅性の担保という点ではまだ検討の余地が残っている．

我々の技術は pulled-up method (pushed-down method) まで含めた既存技術では検出できない影響も検出することを目指しており，他の影響を与える要素についても判定・検出できるよう拡張していく見込である．

6. むすびに

本稿では，ライブラリアップデートの文脈において，コールグラフ上到達するライブラリ変更箇所だけでなく，アップデート前後のコールグラフの差分を取得することでアップデートによって動作変化が潜在的に生じうるプログラムの範囲も示す影響解析手法を提案した．

我々は提案手法を Java 向けに影響解析ツールを実装し，pulled-up method (pushed-down method) を伴うプログラム変更を含むサンプルソフトウェア，サンプルライブラリを対象に評価を行った．結果，サンプルソフトウェアから到達するコード差分に加えて，既存技術では見落としてしまう pulled-up method (pushed-down method) と対応するコールグラフ差分を判定，提示できることを確認した．

付　　録

A.1 実世界ライブラリアップデートへの適用の試み

3 節の判定手法について，実世界のライブラリアップデートへの適用を試みた内容を付録として記載する．検体として，HTML パーサライブラリの Jsoup[*3] を用意した．

適用した結果を表 A·1 に記載する．表中の検出 pulled-up / pushed-down はアプリケーションから到達する範囲で判定された pulled-up method (pushed-down method) の数である．メソッド到達率はアップデートに伴う差分が存在するメソッドの内，アプリケーションから到達するメソッドの割合を示す．

Jsoup の例ではアプリケーションから到達する影響範囲に pulled-up method (pushed-down method) の性質（1，3）を満たす呼び出し関係のペアが 3 件検出された．検出された一例が図 A·1 である．呼び出し関係 `Document.findFirstElementByTagName(String, Node)-->nodeName()` について，新たな `Node` クラスを継承

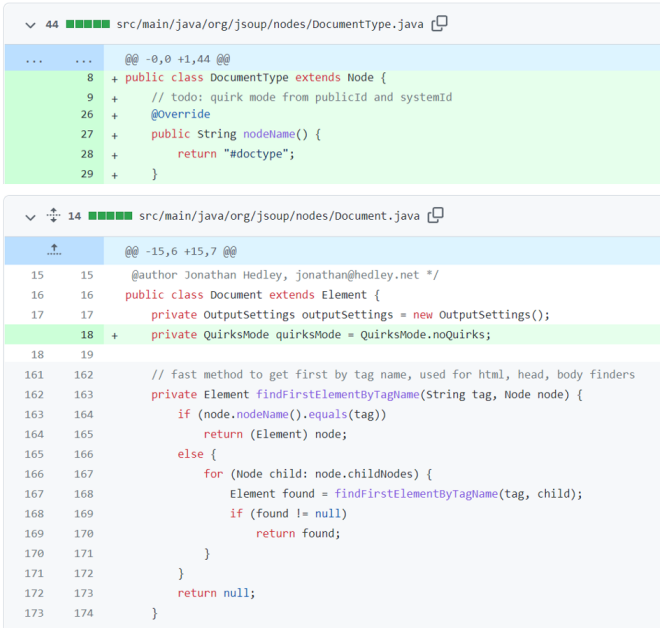


図 A·1: Jsoup 1.5.2 -> 1.6.0 内で検出された性質（1，3）を満たすアップデート影響

する `DocumentType` クラスが追加されることで呼び出し関係が変化し，またコールサイトの 164 行目 `node.nodeName()` も変更されていない．ただし，`DocumentType` クラスと対応するクラスは `XmlDeclaration` であり，両クラスとも `Node` クラスの子クラスにあたる（すなわちクラス階層構造の関係にない）ため性質（2）を満たさず pulled-up method (pushed-down method) とは判定されなかった．pulled-up method (pushed-down method) の性質（1，3）を満たし，性質（2）を満たさないケースは，ライブラリアップデートに伴う実装クラスの置換などの例が考えられ，pulled-up method (pushed-down method) と同様に既存技術では見逃す影響が存在し得ると考えられる．

この先，アプリケーションやライブラリのバリエーションを増やし，pulled-up method (pushed-down method) を含めた既存技術が見逃す影響について検出例を増やしていくことを検討している．また，今回適用した検体も含め，アプリケーションから到達しない範囲に存在する pulled-up method (pushed-down method) についてもどの程度存在するのか調査する見込である．

[*3] `https://github.com/jhy/jsoup`

参考文献

[1] Bohner, S. and Arnold, R.: *Software Change Impact Analysis*, Practitioners Series, Wiley.
[2] Breech, B., Tegtmeyer, M. and Pollock, L.: Integrating Influence Mechanisms into Impact Analysis for In-

ライブラリ	Jsoup
アップデートバージョン	1.5.2 -> 1.6.0
アプリケーション	Jsoup の単体テストを模したサンプルコード
検出 pulled-up / pushed-down	0
メソッド到達率	52.47%
(到達数/全体数)	117/223
ライブラリ SLoC	14,421

表 A·1

creased Precision, *2006 22nd IEEE International Conference on Software Maintenance*, pp. 55–65 (online), DOI: 10.1109/ICSM.2006.33 (2006).

[3] Dig, D. and Johnson, R.: How Do APIs Evolve? A Story of Refactoring: Research Articles, *J. Softw. Maint. Evol.*, Vol. 18, No. 2, p. 83–107 (2006).

[4] Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M. and Monperrus, M.: Fine-Grained and Accurate Source Code Differencing, *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, New York, NY, USA, Association for Computing Machinery, p. 313–324 (online), DOI: 10.1145/2642937.2642982 (2014).

[5] Hattori, L., Guerrero, D., Figueiredo, J., Brunet, J. and Damásio, J.: On the Precision and Accuracy of Impact Analysis Techniques, *Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008)*, pp. 513–518 (online), DOI: 10.1109/ICIS.2008.104 (2008).

[6] Konstantopoulos, D., Marien, J., Pinkerton, M. and Braude, E.: Best Principles in the Design of Shared Software, *2009 33rd Annual IEEE International Computer Software and Applications Conference*, Vol. 2, pp. 287–292 (online), DOI: 10.1109/COMPSAC.2009.151 (2009).

[7] Kotonya, G. and Hutchinson, J.: Analysing the Impact of Change in COTS-Based Systems, *COTS-Based Software Systems* (Franch, X. and Port, D., eds.), Berlin, Heidelberg, Springer Berlin Heidelberg, pp. 212–222 (2005).

[8] Lam, P., Bodden, E., Lhoták, O. and Hendren, L.: The Soot framework for Java program analysis: a retrospective (2011).

[9] Law, J. and Rothermel, G.: Whole Program Path-Based Dynamic Impact Analysis, *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, USA, IEEE Computer Society, p. 308–318 (2003).

[10] Li, B., Sun, X., Leung, H. and Zhang, S.: A survey of code-based change impact analysis techniques, Vol. 23, No. 8, pp. 613–646 (online), DOI: 10.1002/stvr.1475.

[11] Moser, S. and Nierstrasz, O.: The effect of object-oriented frameworks on developer productivity, *Computer*, Vol. 29, No. 9, pp. 45–51 (online), DOI: 10.1109/2.536783 (1996).

[12] 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, ACM, pp. 215–225 (online), DOI: 10.1145/3092703.3092721 (2017).

[13] Ponta, S. E., Plate, H. and Sabetta, A.: Detection, assessment and mitigation of vulnerabilities in open source dependencies, *Empirical Software Engineering*, Vol. 25, No. 5, pp. 3175–3215 (online), DOI: 10.1007/s10664-020-09830-x (2020).

[14] Prana, G. A. A., Sharma, A., Shar, L. K., Foo, D., Santosa, A. E., Sharma, A. and Lo, D.: Out of sight, out of mind? How vulnerable dependencies affect open-source projects (2021).

[15] Raemaekers, S., van Deursen, A. and Visser, J.: Measuring software library stability through historical version analysis, *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pp. 378–387 (online), DOI: 10.1109/ICSM.2012.6405296 (2012).

[16] Raemaekers, S., van Deursen, A. and Visser, J.: Semantic Versioning and Impact of Breaking Changes in the Maven Repository, *J. Syst. Softw.*, Vol. 129, No. C, pp. 140–158 (online), DOI: 10.1016/j.jss.2016.04.008 (2017).

[17] Ren, X., Shah, F., Tip, F., Ryder, B. G. and Chesley, O.: Chianti: A Tool for Change Impact Analysis of Java Programs, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, New York, NY, USA, Association for Computing Machinery, p. 432–448 (online), DOI: 10.1145/1028976.1029012 (2004).

[18] WALA: Watson Libraries for Analysis, `https://wala.sourceforge.net/wiki/index.php/Main_Page` (2018).

[19] Xavier, L., Brito, A., Hora, A. and Valente, M. T.: Historical and impact analysis of API breaking changes: A large-scale study, *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 138–147 (online), DOI: 10.1109/SANER.2017.7884616 (2017).