修 士 論 文

題　目　　JavaScript における関数入出力定義による
非推奨 API の半自動修正手法の提案

主任指導教員　　水野 修　教授

指導教員　　崔 恩瀞　助教

京都工芸繊維大学大学院 工芸科学研究科

情報工学専攻

学生番号　　22622019

氏　　名　　玉井　陽博

令和 6 年 2 月 8 日提出

JavaScript における関数入出力定義による非推奨 API の半自動修正手法の提案

令和 6 年 2 月 8 日　　22622019　玉井　陽博

概　要

ライブラリは API という形でソフトウェアに外部機能を提供し，これを用いることで開発を効率化することが出来る．しかし，ライブラリのアップデートに伴い API の利用が非推奨になり，それに伴う利用箇所のコードメンテナンスが必要になる場合がある．また，JavaScript は Web アプリやネイティブアプリ等の開発で近年最も利用されている言語となっており，そのような開発においてはライブラリが盛んに利用されている．そのため，JavaScript においても非推奨となった API のメンテナンスを支援する手法が求められる．

本研究では，MLCatchUp という Python を対象とした関数入出力定義による非推奨 API を半自動修正するツールの仕組みを応用した，JavaScript 開発における非推奨 API を半自動修正する手法を提案する．さらに，本手法のユースケースについての検証や，精度を検証する為の実験についての考察を行う．

Proposal of a Semi-Automatic Correction Method for Deprecated APIs in JavaScript Based on Function Input/Output Definitions

February 8, 2024 **22622019 TAMAI Akihiro**

Abstract

Libraries provide external functionality to software through APIs, facilitating development efficiency. However, as libraries undergo updates, API usage may become deprecated, necessitating code maintenance in the affected areas. JavaScript has become the most widely used language in recent years for development, including web apps and native apps, where libraries are heavily utilized. Therefore, methods to assist in maintaining deprecated APIs in JavaScript are needed.

In this study, we propose a method for semi-automatic correction of deprecated APIs in JavaScript development. The propsed technique is developed based on a semi-automatic correction method of deprecated APIs for Python. Finally, we conduct an examination of the use cases of this technique and experiments to verify its accuracy.

目　次

1. 緒言

ソフトウェア開発においては，機能を再利用し，開発時間を節約するために多くのサードパーティコードを使用してソフトウェアが構築されている [5]．サードパーティコードは，外部ライブラリにより提供される API(Application Programming Interface) の形で提供される．これをプログラム内に組み込む形で利用され，開発者は詳細な実装を知らずとも機能を実装できる．些細な機能を利用する場合であっても，独自のコードを開発せずに外部ライブラリが選択される場合もあるほど利用されている [6]．

そのような外部ライブラリは通常，新機能の追加，バグの修正，パフォーマンスの向上のために定期的にバージョンアップが行われる．これに合わせてライブラリのアップデートを行うことは，セキュリティおよびパフォーマンスの観点から重要である．

しかし，アップデートに伴い，提供する API の仕様変更が発生してしまい，仕様変更前の形での API の利用が非推奨になる場合がある．その場合には API を利用していて依存関係になっているソフトウェアにおいて，API を利用しているソースコードの保守作業が必要となる．そのような仕様変更が発生してしまい，メンテナンスが必要になってしまった API は非推奨 API と呼ばれる．

ここで，動的型付け言語を対象とした非推奨 API のメンテナンスを行う手法に，MLCatchUp[1] というツールを用いた手法がある．MLCatchUp は動的型付け言語である Python における機械学習ライブラリを対象として，非推奨 API を利用したソースコードのメンテナンスを半自動的に行うというツールである．

近年では JavaScript は最も人気のあるプログラミング言語の 1 つとなった [7]．これは Web アプリ，モバイルアプリ，ネイティブアプリ，そしてバックエンド開発に広く使用されている．JavaScript はプログラム実行時にデータ型が決定される動的型付け言語であるという特徴がある．また，JavaScript における外部ライブラリを管理するシステム，npm も広く利用されており [7]，ソフトウェア開発においてライブラリが広く使われている．故に，JavaScript においてもライブラリの更新及びそれに伴う非推奨 API のメンテナンスは必要である．よって，JavaScript における外部ライブラリの更新に伴う非推奨 API のメンテナンスを支援することは重要である．

本研究では，JavaScript における非推奨 API のメンテナンスを支援することを目的としたツール，JSOutdatedApiChanger の作成を行い，ライブラリの更新を支援する手法提案および正当性の検証を行う．JSOutdatedApiChanger の作成には，JavaScript と同じく動的型付け言語の Python を対象とした MLCatchUp の仕組みを応用する．

また，手法の有効性について議論を行うために，JSOutdatedApiChanger のユースケースについての検証を行い，さらに手法を定量的に評価する為の実験についての考察を行った．

本論文の以降の構成は以下の通りである．2 章では本研究の背景として，非推奨 API の利用やセマンティックバージョニング，及び JavaScript における開発の実態や先行研究である MLCatchUp について説明する．3 章では本研究の目的を説明する．4 章では本研究で提案するライブラリ更新を支援する手法の提案を行う．5 章では本手法のケーススタディおよび手法を評価する実験の方法とその妥当性についての検証について述べる．6 章では非推奨 API のメンテナンスを支援する関連研究についての紹介を行い，7 章では本研究のまとめを行う．

2. 研究背景

本章では，本研究の背景について説明する．

2.1 非推奨 API

ソフトウェア開発の際にライブラリが提供するサードパーティコードを利用することで，機能を再利用することが出来る [5]．そのようなサードパーティーコードは，外部ライブラリにより提供される API(Application Programming Interface) の形で提供される．また，新機能の追加，バグの修正，パフォーマンスの向上を目的として，ライブラリは定期的にバージョンアップされる．よってセキュリティおよびパフォーマンスの観点からもライブラリのバージョンアップを適用するアップデートは重要である．

しかし，バージョンアップに伴いライブラリの仕様変更が発生し，提供する API の利用が非推奨になってしまう場合がある．そのような仕様変更が発生してしまった API は非推奨 API と呼ばれる．非推奨 API が暫くの間そのまま利用可能なケースもあるが，いずれ API が削除されてしまうため，そのままの API の利用は望ましくない．API が非推奨になってしまった場合には，非推奨 API と依存関係になっているソフトウェアにおいて，その API を利用しているソースコードの保守作業が必要となる．しかしながら一般的にソフトウェアのメンテナンスは開発者に嫌われることが多く，開発者がソースコードのメンテナンスを避けるために，機能上は動作に問題が無ければライブラリのアップデートを放置してしまうケースがある [8].

本論文では，ライブラリのバージョンアップに伴う非推奨 API のコード利用箇所のメンテナンスを，非推奨 API のメンテナンスと呼ぶ．このような非推奨 API のメンテナンスの手間が積極的にライブラリ更新が行われない大きな要因となる．さらに，ライブラリ更新が行われないことは脆弱性の原因となりうる．例えば，2021 年に Java にログ出力機能を提供する Apache Log4j というライブラリに任意のコードをリモート実行出来る重大な脆弱性，Log4Shell が発見された．これを対処する為に該当ライブラリのバージョンをアップデートする必要があった [9]．これは極めて危険な脆弱性であったため，至急の対応が必要となったが Log4j を利用するソフトウェ

アの多さから，メンテナンスは困難を極めた．

2.2 セマンティックバージョニング

セマンティックバージョニングは，バージョニングルールの一種である．ライブラリを用いたソフトウェアの開発時，ライブラリ更新時に非推奨 API の出現に伴いライブラリの依存関係を解決する為のメンテナンスが必要になる場合が発生しうる．それを行う為のコストが開発において問題になる．そういった問題を解決するために，このようなバージョニングルールが提案されている (注 1)．

セマンティックバージョニングは，

メジャーバージョン. マイナーバージョン. パッチーバージョン

の形式で表される．

メジャーバージョンは，API 互換性の無い破壊的なアップデートに伴い変化するバージョンである．

マイナーバージョンは，API 互換性を有する非破壊的なアップデートに伴い変化するバージョンである．

パッチバージョンは，機能変更を伴わないバグや脆弱性の修正のみを行った場合に変化するバージョンである．

以上より，セマンティックバージョニングが導入されているライブラリのアップデートにおいて，メジャーバージョン以外のみ更新がある場合は，API 互換性を保持しており，非推奨 API のメンテナンスを必要としないことが判断出来る．このようなメンテナンスを必要としないライブラリ更新の場合，バージョンを積極的に更新することが出来る．

しかし実際には，セマンティックバージョニングが導入されていてもこのようなルールの適用が厳密になされていない場合があることが複数の研究で示されている [10],[11]．そのため，本バージョニングが導入されている場合でも必ずしもソースコード修正是非の基準には出来ないと言える．

(注 1)：https://semver.org/

2.3 JavaScript における開発

JavaScript（JS）は，1995 年に Web ブラウザに動的なクライアントサイドの機能を追加するために導入されたプログラミング言語である [12]．現在では Web アプリ開発のみならずモバイルアプリ，デスクトップクライアント，そしてバックエンド開発等に広く使用されている．2023 年の Stackoverflow の調査によると，現在最も広く使用されているプログラミング言語である [7].

JavaScript は動的に型付けされており，実行時に型がチェックされる．静的型付けによる開発を行うための JavaScript のスーパーセットとして TypeScript(TS) が Microsoft によって 2012 年にリリースされた (注 2)．TypeScript は実行前にコンパイラによって JavaScript にトランスパイルされ，その際に型チェックを行い，関連する型のバグがある場合検出する [13]．ここで，JavaScript と TypeScript について，ソフトウェア品質において一概にどちらが優れているとは言えないという研究結果がある [14].

JavaScript 及び TypeScript 開発におけるライブラリの管理には，npm (注 3) というパッケージマネージャーが広く使われている．JavaScript 及び TypeScript は特に web システムにおけるシングルページアプリケーション（SPA）の開発において人気がある．だが，特にインターネット上で公開されている web システムは脆弱性の放置が深刻な問題となる．故に，JavaScript および TypeScript における開発においても，ライブラリ更新を支援する為に，非推奨 API の更新に伴うメンテナンスが重要になる．

2.4 MLCatchUp[1]

AI アプリケーションの台頭に伴い，機械学習ライブラリはより利用しやすくなり，それを記述するための最も一般的なプログラミング言語が Python である．そのような Python 機械学習ライブラリに関しても，ライブラリの更新に伴う非推奨 API の問題が発生している．ディープラーニングライブラリの古いバージョンを使用すると，セキュリティの問題やパフォーマンスの低下が起こる可能性についても懸念されている [15]．そこで，そのような Python 機械学習ライブラリの非推奨 API のメ

(注 2)：https://www.typescriptlang.org/
(注 3)：https://www.npmjs.com/

ンテナンスを半自動的に行い，ライブラリの更新を支援することを目的としたツールが MLCatchUp である．MLCatchUp は，機械学習やコード例の用意の無しにそのようなメンテナンスを行うことができる．ただし，1 種類ずつの非推奨 API を削除する場合および 1 種類の非推奨 API の変更にのみ対応している．そのため 1 種類の非推奨 API の仕様変更が新しい複数種類の API にまたがる場合のメンテナンスには対応していない．

MLCatchUp に API が関数として提供される Python ライブラリの非推奨 API 及び，更新された API について，それぞれのシグネチャ(関数入出力定義) を入力すると，変更点が MLCatchUp 独自のドメイン言語（DSL）で表現される．そして，DSL を入力することで，該当 API と依存関係にある Python コードの置き換えが自動的に完了する．

MLCatchUp のアーキテクチャを，図 2.1 に示す.

Python ファイル（O-1）と，非推奨の API と更新された API シグネチャ（O-2）を入力とする．入力された Python ファイルは，Python が提供する抽象構文木モジュールを使用して AST に変換され（S-1），入力ファイル AST（O-3）が作成される．この AST から MLCatchUp は，非推奨 API の利用箇所を非推奨 API のシグネチャに基づいて検出および特定する．

入力の API シグネチャは変換推論プロセス（S-2）で使用され，API 移行に必要な変換が自動的に推論される．

これらの必要な変換は DSL コマンドの形式であり（O-4），更新に必要な一連の操作（メソッドの名前変更，パラメータの名前変更など）が順次実行される．

次に，DSL は DSL パーサー（S-3）によって，MLCatchUp で実行できる操作のリストに解析される．

これらの操作は入力ファイル AST（S-4）に適用され，更新された AST（O-5）が生成される．

更新された AST は，コードの差分チェッカー（S-5）を使用して入力ファイル AST と比較される．そして 2 つの AST 間のすべてのコードの違いをリストアップし，更新差分（O-6）を出力する．

最後に，更新差分（O-6）のコードの違いに基づいて，必要な変更のみを行いながら入力の Python ファイルに更新が適用される（S-6）.

本研究で提案するツールは MLCatchUp のアーキテクチャを踏襲したものとなっている.

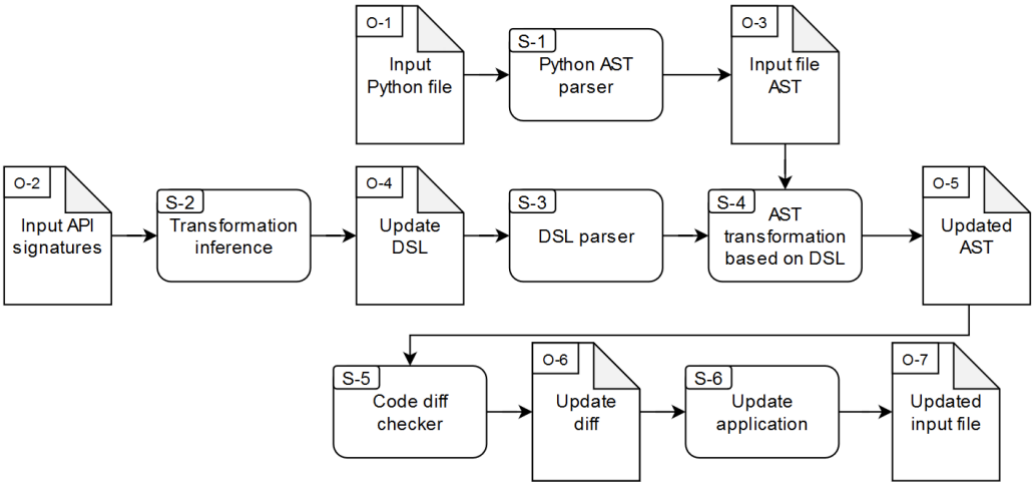

図 2.1　MLCatchUp のアーキテクチャ．[1] より引用．

3. 目的

本研究の目的は，JavaScript を用いたソフトウェアの開発において，ライブラリのアップデートを支援する手法の提案である．ライブラリのアップデートを支援するアプローチの 1 つとして，次の方針が考えられる．それは，ライブラリのバージョンアップ時に発生する可能性がある，非推奨 API を利用しているソースコードのメンテナンスを支援するというものである．

そこで，本研究では，JavaScript における非推奨 API のメンテナンスを支援することを目的としたツール，JSOutdatedApiChanger の作成を行う．具体的には，修正すべき非推奨 API と更新された API の対から，ソースコード中で修正すべき場所と修正候補を提示し，半自動的にメンテナンスを出来るようにする．また，そのようなアプローチを評価する為に，JSOutdatedApiChanger のユースケースについての検証や，精度を検証する為の実験についての考察も行う．

4. 提案手法

本章では，JavaScript を用いた開発におけるライブラリのアップデートを支援する為の手法として，JavaScript における非推奨 API の半自動修正を行うツール，JSOutdatedApiChanger(以下, 本ツールと記す) について説明する．

4.1 概要

1 章及び 2 章でも述べた通り，非推奨 API の置き換えが JavaScript においても重要な課題となっている．JSOutdatedApiChanger の目的は，半自動的に非推奨 API の置換を行い，JavaScript における非推奨 API の修正を支援することである．そのためのツールを作成するに当たって，Python における非推奨 API のメンテナンスを行う MLCatchUp[1] の仕組みを応用した．

JavaScript および Python は，どちらも動的型付け言語である．これらの点から，JavaScript における非推奨 API のメンテナンスを行う JSOutdatedApiChanger を作成するに当たって，MLCatchUp の手法を応用することが効果的であると考察した．

利用手順は MLCatchUp と同様に，非推奨 API 及び，更新された API，それぞれのシグネチャを入力とする．両 API のシグネチャはユーザーが手動で取得する．すると非推奨 API の移行に必要な操作が独自のドメイン言語（DSL）で表現される．これをプログラムに入力すると，それを基に自動的に API を利用しているコードの修正が行われる．本ツールはこのような半自動的な手順で利用できる．

非推奨 API の更新された API への移行について，類型に分けることが出来る．本手法ではそうした類型に基づいた変換操作を行う．本手法において DSL で表現される非推奨 API メンテナンスの類型を 4.2 節で説明する．また，DSL の仕様を 4.3 節で説明する．

4.2 非推奨 API 類型

本手法で対応している，非推奨 API を更新された API に移行するために必要な API の更新操作のパターンを説明する．

4.2.1　パラメータの削除

この操作は非推奨 API の使用法の関数パラメータの削除を行う．

- Foo.Bar(123,”STR”);

+ Foo.Bar(”STR”);

このような API の変更で，*Foo.Bar* のパラメータは 1 番目に number 型のパラメータを，2 番目に string 型のパラメータを持っているが，更新後，1 番目の number 型のパラメータが削除されている．JavaScript の場合，Python と異なりキーワードパラメータは導入されておらず，位置パラメータのみ導入されている．

4.2.2　関数の名前変更

この操作は非推奨の API の関数の名前を新しい更新された名前に変更する．

- ReactPerf.getMeasurementsSummaryMap();

+ ReactPerf.getWasted();

このような API の変更で，$ReactPerf.getMeasurementsSummaryMap$ の関数名が $ReactPerf.getWasted$ に置き換えられている．

4.2.3　パラメータの追加

この操作では API の呼び出しにパラメータの追加を行う．

- app.param((a)= {});

+ app.param(”NAME”, (a)= {});

このような API の変更で，*app.param* のパラメータの 1 番目に number 型の 1 番目に string 型の引数が追加され，元の 1 番目の function 型を持つパラメータが 2 番目の引数に移動している．本ツールでは，パラメータの削除と同様に，JavaScript の場合，Python と異なりキーワードパラメータは導入されておらず，位置パラメータのみ導入されている．

4.2.4　API の削除

この操作は，非推奨 API を削除し，その使用法を更新された API で置き換えない場合に行う．これは通常，非推奨 API が不要になったか，呼び出されたときに効果がなくなった場合に使用される．本手法では，非推奨 API の前後にコメントで非推奨 API の利用を強調し，API の置換を促す．例えば，*chalk.hasColor*() という非推奨 API を削除する操作を行う場合は以下のようになる．

```
- Foo = chalk.hasColor();
+ Foo = /*DEPRECATED*/chalk.hasColor()/*DEPRECATED*/;
```

4.2.5　フィールド名の変更

この操作は関数ではフィールドの形で提供される非推奨の API の関数のフィールド名を新しい更新された名前に変更する．

```
- Foo = ChalkOptions;
+ Foo = Options;
```

このような API の変更で，*ChalkOptions* というフィールドが *Options* に置き換えられている．

4.3　アーキテクト

JSOutdatedApiChanger のアーキテクトを図 4.1 に示す．

JS ファイル (1) と，非推奨の API と更新された API シグネチャ(2) を入力とする．入力された JS ファイル (1) は，JavaScript のスーパーセットである TypeScript がネイティブに提供する Compiler API の機能を使用して抽象構文木 (AST) に変換される (3)．本手法では，Compiler API のラッパである ts-morph を用いて AST の作成や操作を行っている．

また，入力された API シグネチャ(2) の情報を基に，API 移行に必要な変換が前節で述べたどの類型に当てはまるかを自動的に推論する (4)．そうしたら，変更点を本手法の DSL(ドメイン固有言語) としてデータ化する (5)．

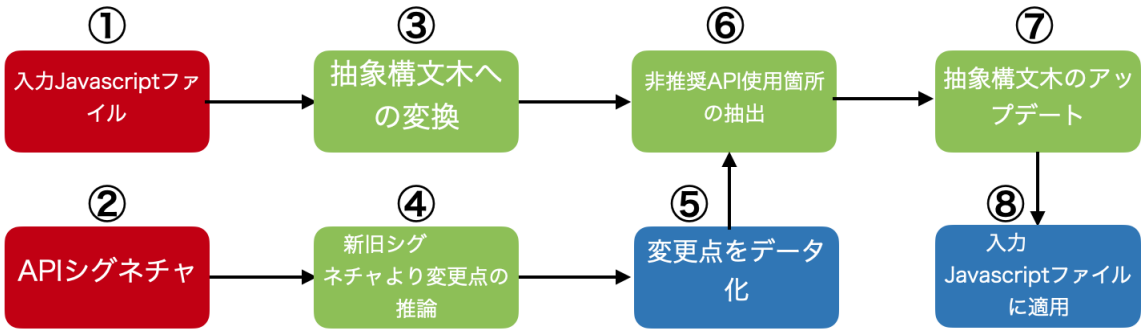

図 4.1　JSOutdatedApiChanger のアーキテクト．入力に関する箇所は赤色、出力に関する場所は青色で示している．

DSL を表 4.1 に示す．本ツールでは，パラメータの追加及び削除の際，対象となる引数の先頭からの位置を指定する．また，パラメータの追加の際には，追加するパラメータの値を直接入力する．直接入力された追加値は，(7) の際にトランスパイルされる．

次に，AST を解析し，JS ファイル内の全ての関数及びフィールドを正規化した後，DSL を基に非推奨 API の利用箇所を特定し (6)，自動的に非推奨 API の (7) アップデートを行う．

最後に，アップデートされた AST を再び JavaScript コードに復元し，出力する (8) ことで，一連の置換プロセスが完了する．

表 4.1　JSOutdatedApiChanger の API 置換に関する DSL

操作	DSL
パラメータの削除	$remove\_parameter$ < 非推奨 API の関数名 > < 除去する引数の位置 >
関数の名前変更	$rename\_method$ < 非推奨 API の関数名 > < 更新後 API の関数名 >
パラメータの追加	$add\_parameter$ < 非推奨 API の関数名 > < 引数の挿入位置 >< 追加する引数の中身 >
API の削除	$remove\_method$ < 非推奨 API の関数名 >
フィールド名の変更	$rename\_field\_name$ < フィールド名 > < フィールド名 >

5. 評価実験

本章では，3 章で説明した JSOutdatedApiChanger のユースケースについての説明及び，正確性を調査するために行った実験の手法とその正当性への脅威について議論する．

5.1 ユースケース

ユースケースについて解説を行う．以下のユースケースは全て実際のソフトウェア開発のリポジトリより抽出したものである．

5.1.1 ケース 1

ライブラリの仕様変更に伴い，API を呼び出している関数名を変更するメンテナンスが必要になる場合がある．そのような修正を行った例として，udata というソーシャルプラットフォームを提供するフレームワークのコミット (注 4)に，本手法を適用する場合を考える．

Listing5.1 は，非推奨 API のメンテナンスを行う前のソースコードの抜粋である．

該当のプログラムには，moment ライブラリの $langData()$ という API が利用されている．ここで，ライブラリの Changelog (注 5)を確認すると，バージョン 2.8.0 へのライブラリアップデートに伴い $langData()$API の利用が非推奨となり，$localeData()$API に置き換えることが推奨されていることが分かる (図 5.1).

これより，非推奨 API のシグネチャを $langData()$，更新した API のシグネチャを $localeData()$ とし，JSOutdatedApiChanger にこれを入力し，DSL を作成する．生成された DSL は以下のようになる．

$$rename\_method\ langData\ localeData$$

続いて，該当コードが記述されている JS ファイルを対象に DSL を実行すると，Listing5.2 に示すコードに置き換えられ，半自動的な非推奨 API のメンテナンスが完了した．

(注 4): https://github.com/opendatateam/udata/commit/c6d32be1e9f5d3e37658a2a11f8aa4d41e54d6bf
(注 5): https://github.com/moment/moment/blob/develop/CHANGELOG.md

Listing 5.1　ケース 1 の置換前ソースコードの抜粋

```
locale: {
    applyLabel: i18n._('OK'),
    cancelLabel: i18n._('Cancel'),
    fromLabel: i18n._('From'),
    toLabel: i18n._('To'),
    weekLabel: i18n._('week-label'),
    customRangeLabel: i18n._('Custom Range'),
    firstDay: moment.langData()._week.dow
}
```

2.8.0 See changelog

- Release July 31, 2014
- incompatible changes
  - #1761: moments created without a language are no longer following the global language, in case it changes. Only newly created moments take the global language by default. In case you're affected by this, wait, comment on #1797 and wait for a proper reimplementation
  - #1642: 45 days is no longer "a month" according to humanize, cutoffs for month, and year have changed. Hopefully your code does not depend on a particular answer from humanize (which it shouldn't anyway)
  - #1784: if you use the human readable English datetime format in a weird way (like storing them in a database) that would break when the format changes you're at risk.
- deprecations (old behavior will be dropped in 3.0)
  - #1761 `lang` is renamed to `locale`, `langData` -> `localeData`. Also there is now `defineLocale` that should be used when creating new locales
  - #1763 `add(unit, value)` and `subtract(unit, value)` are now deprecated. Use `add(value, unit)` and `subtract(value, unit)` instead.
  - #1759 rename `duration.toIsoString` to `duration.toISOString`. The js standard library and moment's `toISOString` follow that convention.

図 5.1　moment の Changelog.md の抜粋．下線部は langData の更新情報．

Listing 5.2　ケース 1 の置換後ソースコードの抜粋

```
locale: {
    applyLabel: i18n._('OK'),
    cancelLabel: i18n._('Cancel'),
    fromLabel: i18n._('From'),
    toLabel: i18n._('To'),
    weekLabel: i18n._('week-label'),
    customRangeLabel: i18n._('Custom Range'),
    firstDay: moment.localeData()._week.dow
}
```

5.1.2 ケース 2

API を呼び出している関数名の変更および，新たな引数が追加された場合を考える．そのような修正を行った例として，meteor-routecore という Meteor.js のプラグインのコミット (注 6)を基に，本手法を適用する場合を考える．

Listings5.3 は，非推奨 API のメンテナンスを行う前のソースコードである．

該当のプログラムには，React ライブラリの $this.getDOMNode()$ という API が利用されている．ライブラリの changelog (注 7)を確認すると，バージョン 1.4.0 へのライブラリアップデートに伴い $this.getDOMNode()$ の利用が非推奨となり，$ReactDOM.findDOMNode(this)$API に置き換えることが推奨されていることが分かる (図 5.2)．

非推奨 API のシグネチャを $this.getDOMNode()$，更新した API のシグネチャを $ReactDOM.findDOMNode(this)$ とし，JSOutdatedApiChanger にこれを入力し，DSL を作成する．生成された DSL は以下のようになる．

$$rename\_method \quad this.getDOMNode \quad React.findDOMNode$$
$$add\_parameter \quad React.findDOMNode \quad 0 \quad this$$

続いて，該当コードが記述されている JS ファイルを対象に DSL を実行すると，Listing5.4 に示すコードに置き換えられ，半自動的な非推奨 API のメンテナンスが完了した．

5.2 実験方法

JSOutdatedApiChanger を定量的に評価するために，以下の手法による実験を試みた．

5.2.1 データセットの選定

以下の手順により，データセットの収集を行った．まずは，最も npm で利用されている上位 10 ライブラリを取得した．npmjs.com にログインすると，そのトップペー

(注 6)：https://github.com/mystor/meteor-routecore/pull/25/commits/b654671c93da69e7c3c65fa8512b120ebe33ae80
(注 7)：https://github.com/facebook/react/blob/main/CHANGELOG.md

Listing 5.3　ケース2の置換前ソースコードの抜粋

```
componentDidMount: function() {
  this._dynamicTmpl = new Iron.DynamicTemplate();
  this._dynamicTmpl.template(component);
  this._dynamicTmpl.data(this.props);
  this._dynamicTmpl.insert({ el: this.getDOMNode() });
},

componentWillReceiveProps: function(newProps) {
  this._dynamicTmpl.data(newProps);
},
```

0.14.0 (October 7, 2015)

Major changes

- Split the main `react` package into two: `react` and `react-dom`. This paves the way to writing components that can be shared between the web version of React and React Native. This means you will need to include both files and some functions have been moved from `React` to `ReactDOM`.
- Addons have been moved to separate packages (`react-addons-clone-with-props`, `react-addons-create-fragment`, `react-addons-css-transition-group`, `react-addons-linked-state-mixin`, `react-addons-perf`, `react-addons-pure-render-mixin`, `react-addons-shallow-compare`, `react-addons-test-utils`, `react-addons-transition-group`, `react-addons-update`, `ReactDOM.unstable_batchedUpdates`).
- Stateless functional components - React components were previously created using React.createClass or using ES6 classes. This release adds a new syntax where a user defines a single stateless render function (with one parameter: `props`) which returns a JSX element, and this function may be used as a component.
- Refs to DOM components as the DOM node itself. Previously the only useful thing you can do with a DOM component is call `getDOMNode()` to get the underlying DOM node. Starting with this release, a ref to a DOM component *is* the actual DOM node. **Note that refs to custom (user-defined) components work exactly as before; only the built-in DOM components are affected by this change.**

Breaking changes

- `React.initializeTouchEvents` is no longer necessary and has been removed completely. Touch events now work automatically.
- Add-Ons: Due to the DOM node refs change mentioned above, `TestUtils.findAllInRenderedTree` and related helpers are no longer able to take a DOM component, only a custom component.
- The `props` object is now frozen, so mutating props after creating a component element is no longer supported. In most cases, `React.cloneElement` should be used instead. This change makes your components easier to reason about and enables the compiler optimizations mentioned above.
- Plain objects are no longer supported as React children; arrays should be used instead. You can use the `createFragment` helper to migrate, which now returns an array.
- Add-Ons: `classSet` has been removed. Use classnames instead.
- Web components (custom elements) now use native property names. Eg: `class` instead of `className`.

Deprecations

- `this.getDOMNode()` is now deprecated and `ReactDOM.findDOMNode(this)` can be used instead. Note that in the common case, `findDOMNode` is now unnecessary since a ref to the DOM component is now the actual DOM node.
- `setProps` and `replaceProps` are now deprecated. Instead, call ReactDOM.render again at the top level with the new props.
- ES6 component classes must now extend `React.Component` in order to enable stateless function components. The ES3 module pattern will continue to work.
- Reusing and mutating a `style` object between renders has been deprecated. This mirrors our change to freeze the `props` object.

図 5.2　React の Changelog.md の抜粋．下線部は $this.getDOMNode()$ の更新情報．

Listing 5.4　ケース2の置換後ソースコードの抜粋

```
componentDidMount: function() {
  this._dynamicTmpl = new Iron.DynamicTemplate();
  this._dynamicTmpl.template(component);
  this._dynamicTmpl.data(this.props);
  this._dynamicTmpl.insert({ el: ReactDOM.findDOMNode(this) });
},

componentWillReceiveProps: function(newProps) {
  this._dynamicTmpl.data(newProps);
},
```

ジ [16] から上位 10 ライブラリを取得することが出来る．それらのライブラリの中から，プロダクトのソースコードに API を組み込む形で機能を提供するライブラリを絞り込み，対象ライブラリとした．選択したライブラリを表 5.1 に示す．ここで，tslib というライブラリは，TypeScript のコンパイル時にビルドサイズを圧縮するランタイムであったため除外した．

対象ライブラリより，本実験に用いるデータセットを収集するフローを図 5.3 に示す．対象ライブラリの更新履歴が記されている CHANGELOG.md ファイルを調査した．ここで，JSOutdatedApiChanger で対応している形式の非推奨 API 及び更新後 API の入出力情報を抽出した (1)．続けて，GitHub より，スター数の多い順に javascript を利用しているリポジトリを取得し (2)，それぞれにパッケージ依存についての情報が記されている package.json ファイルを調べた．JavaScript には，Jest という自動テストシステムがあり，これを導入することでテストケースに沿った自動テストを行うことが出来る．ここで Jest が導入されており，かつ対象ライブラリが 1 つ以上インストールされているものに絞り込んだ (3)．そして，絞り込んだライブラリの全コミットを取得し，各コミットの diff ファイルから，変更前に非推奨 API が含まれており，かつ変更後に推奨 API が含まれているものをテキストベースで検索し，コミットを抽出した (4)．本研究では，スター数 123 以上のリポジトリの中から 598 リポジトリに絞り込み，248 コミットを抽出した．

5.2.2 実験

本実験手法におけるコミットの位置関係を図 5.4 に示す．

1 つの抽出したコミットにつき該当リポジトリをダウンロードし，そのコミット (A) の 1 つ前のコミット (B) で，JSOutdatedApiChanger を利用して非推奨 API の置換を行う．この状態を (C) とする．次に，該当コミット (B) の Jest テストケースで (C) に対するテストを行い，全ケーステストが通過することを確認出来た場合に JSOutdatedApiChanger が正常に動作していると見做す．これを全ての抽出したコミットに対して行い，各コミットに対して全テストケースが通過した割合で本手法の性能を評価する．

表 5.1　上位 10 ライブラリ及び選定したライブラリ

ライブラリ名	選定
react	o
react-dom	o
lodash	o
axios	o
chalk	o
commander	o
express	o
moment	o
vue	o
commander	o
tslib	x

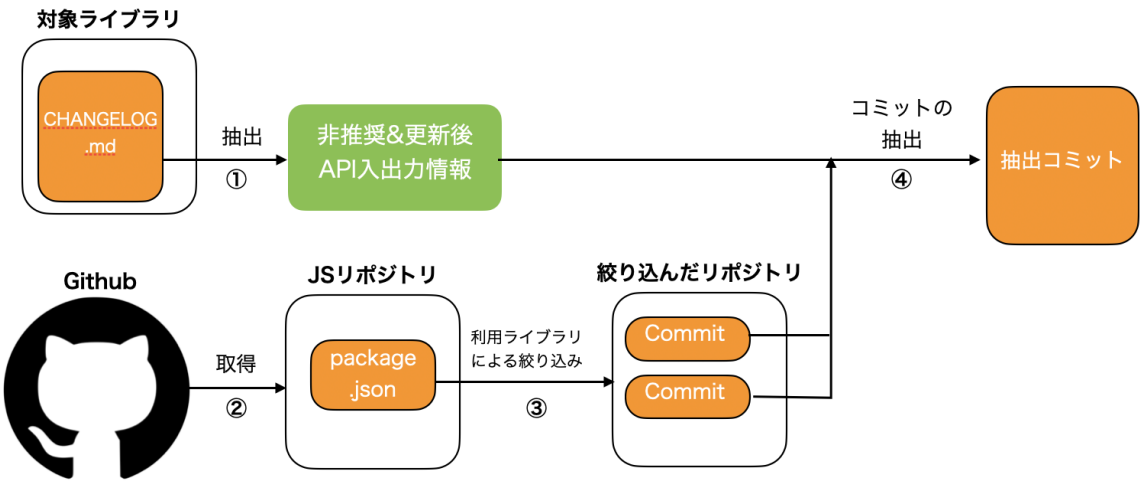

図 5.3　データセット収集方法のフロー

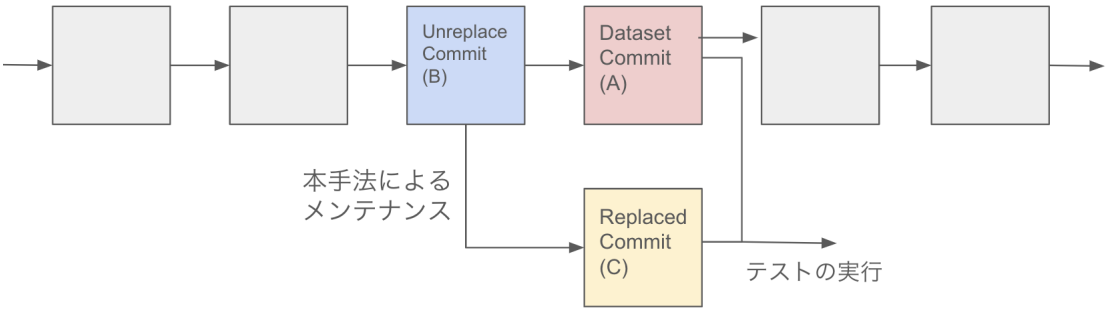

図 5.4　本実験手法におけるコミットの位置関係

5.3 実験の結果

以上の手法による評価実験を，まず最初に 25 個のコミットに対して行った．しかし，これらのコミットが評価実験に用いることが不適なことが分かった．不適なコミットの類型を以下に挙げる．

1. 自動テストに通過していないままデプロイされている．
2. テストケースがデフォルトテンプレートのまま変更されておらず，単に Jest をインストールした状態のままとなっている．
3. 該当の非推奨 API の置換が行われているコミットが，テストケースの置換を含有していない．
4. 対象となる API の変更がデータセットのコミットに含まれていない．

以上より，本実験におけるデータセットの取得方法が不適であると判断し，取得した全コミットに対して評価実験を行うことが出来なかった．

5.4 妥当性への脅威

本節では，JSOutdatedApiChanger およびの評価実験の妥当性についての議論を行う．

5.4.1 評価実験の妥当性

本研究では，JSOutdatedApiChanger の評価に対して，定量的なアプローチを試みた．しかし，本研究のデータセット収集手法では，実験に適したデータセットを取得することが出来ず，評価への妥当性に疑問が残る結果となってしまった．

本研究のデータセット収集では，自動テストが導入されているリポジトリにおいて非推奨 API のメンテナンスが行われたコミットの収集を行うことを目的とした．しかし，実際には自動テストが導入されていても，正しく運用されていない場合があった．また，非推奨 API のメンテナンスと同時にテストケースの修正が 1 つのコミットに収められるコミットが存在しなかった．

そして，対象となる API の変更がデータセットのコミットに含まれていない場合があり，この原因についてはテキストベースの検索では API 変更の検索が不十分で

あることが考えられる．これについてはテキストベースではなく構文解析による AST を用いた検索を行うことで改善できた可能性がある．

本手法の性能を評価する為には，別のアプローチによる定量的な評価手法を考案するか，あるいは JavaScript 開発経験者による定性的な評価を実行する必要があると考えられ，今後の課題である．

5.4.2 JSOutdatedApiChanger の妥当性

MLCatchUp には，1 種類の非推奨 API の仕様変更が新しい複数種類の API にまたがる場合のメンテナンスには対応していないという問題があった．JSOutdatedApiChanger においてもこの問題は解決しておらず，そのような複雑な非推奨 API のメンテナンスを行うことが出来ない．

実際の非推奨 API のメンテナンスにおいては，複数種類の API に跨った修正が必要になる場合がある．これに対応したソースコードの自動修正手法を開発することが，より実用的なライブラリのアップデートを支援する手法を開発するために必要であると考えられる．

6. 関連研究

本章では，非推奨 API のメンテナンスを支援する手法に関する関連研究を紹介する．

6.1 COCCINELLE[2]

COCCINELLE は，C 言語における非推奨 API の更新を自動化するツールである．Linux のデバイスドライバが提供する API を用いた C 言語プログラムの開発を対象としている．

COCCINELLE は，次の流れで非推奨 API の更新を自動化する．最初に，非推奨 API と更新後 API のコード片を取得する．そして，利用者は用意されたコード片から，移行パターンの類型を判断し，COCCINELLE の独自言語である SmPL (Semantic Patch Language) を記述する．その後，SmPL に基づいて C 言語プログラムをパースするパーサーによって，C 言語プログラム内の非推奨 API が自動的に置換される．

SmPL は通常の C コードの断片と伝統的な patch ファイルの形式を使用した変換の記述を行っている．そのため開発者は SmPL を読み取りやすく理解しやすい．SmPL では，パターンの選択肢を記述する機能，存在してはならないコードを指定する機能，およびパターンの一部をオプションとして宣言する機能が実装可能である．さらに，識別子，定数，さまざまな型の式，ステートメント，およびパラメータリストなど，すべての種類の用語を抽象化するメタ変数も使用できる．

これらの SmPL の特徴により，開発者がコードの構造に関するルールをコード構造の観点から書くことが可能になる．

COCCINELLE は利用者が非推奨 API の移行パターンを理解し独自言語に落とし込む必要がある，という点において JSOutdatedApiChanger と大きく異なる．

6.2 AppEvolve[3]

Java を用いた Android アプリの開発においても，ライブラリの更新に伴う非推奨 API の発生及びそのメンテナンスが問題となっている．そこで，AppEvolve という，Android の API の廃止に応じて Android アプリの更新を自動化するツールが提唱さ

れた．多くのソフトウェアプロジェクトが同様の方法でAPIを使用する可能性があり，したがって既存のコードベースにはすでに廃止されたAPIの使用方法の例が含まれているかもしれないという仮説が建てられる．これを基にした仕組みにより，非推奨APIの置き換えがなされている．

廃止されたAPIを使用するターゲットアプリと，それを置き換える新しいAPIメソッドに関する情報が与えられた場合，AppEvolveは既存のコードベースから更新例を収集する．続いて変数の参照を抽象化し，それらの期待される適用可能性に基づいてそれらをランク付けする．そして抽象化された例をターゲットアプリに1つずつ適用する．抽象化された例がコードと一致する場合，更新は適用可能となる．適用可能な更新はさらにテストによって検証される．AppEvolveの特徴の1つは，特定の使用コンテキストに固有の情報を失わないように，抽象化された例を単一の一般的なパターンにマージしないことである．

AppEvolveは，JSOutdatedApiChangerと比べて，既存の非推奨APIの置き換えを行ったコード例を用意する必要があるという点で大きく異なっている．

6.3 NEAT[4]

NEAT（No Example API Transformation）も，Androidアプリの開発を対象にしているツールである．AppEvolveと異なり，コード例が利用できない場合（例えば，最近廃止された場合など）に於いても，廃止されたAPIの置き換えを開発者が支援できる変換ルールを生成する．これによりコード例を利用する非推奨APIのメンテナンスを補完することが出来る．

NEATは1対1のマッピング（1つの廃止されたAPIから1つの置換APIへのマッピング）に焦点を当てる．つまり，廃止されたAPIのシグネチャ（つまり，メソッド名，クラス名，およびパラメータ），その置換APIのシグネチャ，およびこれら2つのAPIを含むライブラリのソースコードを入力として受け取る．

さらに，廃止されたAPIについて，クライアントアプリケーションから呼び出されるか，またはクライアントアプリケーションによってイベントハンドラーとして実装されるか，の2つのケースに区分する．これらのケースを区別する方法は，API名がonで始まるかどうか（ex:onMetadataChanged）で，始まる場合はイベントハン

ドラーと見なす.

それより，非推奨および更新後の両方の API がイベントハンドラーであるかどうかを確認する．どちらもイベントハンドラーでない場合，どちらもイベントハンドラーである場合，どちらかのみイベントハンドラーである場合に分けて変換ルールを生成する．

NEAT は静的型付け言語である Java を対象としており，一方 JSOutdatedAPI は動的型付け言語である JavaScript を対象としているという点で異なっている．

6.4 MLCatchUp[1]

MLCatchUp は，Python の機械学習ライブラリを対象とした非推奨 API の更新を自動化するツールである．これまでに紹介いた手法は C 言語や Java といった，静的型推論を用いる言語のみを対象としていたが，本手法は動的型推論の言語の一種である Python を対象としている．

MLCatchUp は，NEAT と同様に機械学習やコード例の用意の無しにそのようなメンテナンスを行うことができる．

MLCatchUp に Python ライブラリの非推奨 API 及び更新された API についてそれぞれのシグネチャを入力する．そうすると変更点が MLCatchUp 独自のドメイン言語（DSL）で表現される．続いて，DSL を入力することで，該当 API と依存関係にある Python コードの置き換えが自動的に完了する．

JSOutdatedApiChanger と MLCatchUp はどちらも同じ動的型付け言語を対象としている．だが，JavaScript と Python 間には言語仕様に相違点があり，それに伴い対象とする API 移行類型において特に両手法間で大きな違いがある．

7. 結言

本研究では，JSOutdatedApiChanger の開発を通じて，JavaScript 開発における非推奨 API を半自動修正する手法についての提案を行った．

JSOutdatedApiChanger の作成に当たって MLCatchUp という Python を対象とした関数入出力定義による非推奨 API を半自動修正するツールの仕組みを応用した．これにより，非推奨 API と更新後 API の利用コード例を必要とせず，シグネチャを基に非推奨 API を半自動的にメンテナンスする仕組みを取り入れることが出来た．

また，本手法の定量的な評価を行う実験方法についての考察を行い，実行した結果，手法の妥当性に問題のある結果となった．いくつかのユースケースを示すことは出来たものの，本手法についての妥当な評価を行うことは今後の課題としたい．

謝辞

本研究を行うにあたり，研究課題の設定や研究に対する姿勢，本報告書の作成に至るまで，全ての面で丁寧なご指導を頂きました，本学情報工学・人間科学系水野修教授，崔 恩瀞助教に厚く御礼申し上げます．本報告書執筆にあたり貴重な助言を多数頂き，校閲にもご協力を頂いた，ソフトウェア工学研究室の皆さん，学生生活を通じて著者の支えとなった友人，著者の学生生活を全面的に支援して頂いた家族に深く感謝致します．

参考文献

[1] S. Haryono, F. Thung, D. Lo, J. Lawall, and L. Jiang, “Characterization and automatic updates of deprecated machine-learning API usages,” In Proceedings of 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.137–147, 2021.

[2] Y. Padioleau, J. Lawall, R.R. Hansen, and G. Muller, “Documenting and automating collateral evolutions in linux device drivers,” ACM Sigops Operating Systems Review 42(4), pp.247–260, 2008.

[3] F. Thung, S.A. Haryono, L. Serrano, G. Muller, and J. Lawall, “Automated deprecated-api usage update for Android apps: How far are we?,” In Proceedings of IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp.602–611, 2020.

[4] F. Thung, H.J. Kang, L. Jiang, and D. Lo, “Towards generating transformation rules without examples for Android API replacement,” In Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.213–217, 2019.

[5] J. Li, R. Conradi, C. Bunse, M. Torchiano, O.P.N. Slyngstad, and M. Morisio, “Development with off-the-shelf components: 10 facts,” IEEE Software 26(2), p.80–87, 2009.

[6] R. Abdalkareem, O. Nourry, S. Wehaibi, S. Mujahid, and E. Shihab, “Why do developers use trivial packages? An empirical case study on npm.,” In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp.385–395, 2017.

[7] StackOverflow, Stack Overflow Developer Survey, StackOverflow（オンライン）, 入手先〈https://survey.stackoverflow.co/2023/〉（参照 2024-1-31）.

[8] G. Artha, A. Prana, A. Sharma, L.K. Shar, D. Foo, A.E. Santosa, A. Sharma, and D. Lo, “Out of sight, out of mind? How vulnerable dependencies affect open-source projects,” Empirical Software Engineering,Volume 26, pp.1–34, 2021.

[9] National Institute of Standards and Technology, NVD - CVE-2021-44228, National Institute of Standards and Technology（オンライン）, 入手先〈https://nvd.nist.gov/vuln/detail/CVE-2021-44228〉（参照 2024-1-31）.

[10] S. Raemaekers, A.V. Deursen, and J. Visser, “Semantic versioning versus breaking changes: A study of the Maven repository,” In Proceedings of IEEE 14th International Working Conference on Source Code Analysis and Manipulation, pp.215–224, 2014.

[11] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, “Keep me updated: An empirical study of third-party library updatability on Android,” In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp.2187–2200, 2017.

[12] J. Flanagan, Javascript: The Definitive Guide - Master the World’s Most-Used Programming Language, O’Reilly Media, Sebastopol, 2020.

[13] Y. Fain and A. Moiseev, TypeScript Quickly, Manning, Birmingham, 2020.

[14] B. Justus and M. Manuel, “To type or not to type? A systematic comparison of the software quality of JavaScript and TypeScript applications on GitHub,” In Proceedings of the 19th International Conference on Mining Software Repositories, pp.658–669, 2022.

[15] J. Wang, K. Liu, and H. Cai, “Exploring how deprecated Python library APIs are (not) handled,” In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp.233–244, 2020.

[16] npm, npm — Home, npm（オンライン）, 入手先〈https://www.npmjs.com/〉（参照 2023-9-27）.

[17] S.A. Haryono, F. Thung, H.J. Kang, L. Serrano, G. Muller, J. Lawall, D. Lo, and L. Jiang, “Automatic Android deprecated-API usage update by learning from single updated example,” In Proceedings of the 28th International Conference on Program Comprehension, pp.401–405, 2020.