特別研究報告

題目

React アプリケーションを対象としたコードクローン発生原因の調査

指導教員
井上 克郎 教授

報告者
小池　耀

令和 3 年 2 月 9 日

大阪大学 基礎工学部 情報科学科

令和 2 年度 特別研究報告

React アプリケーションを対象としたコードクローン発生原因の調査

小池　耀

内容梗概

近年の Web ページ開発では Web UI フレームワークとして React が広く用いられている．React は JavaScript を拡張した．JSX や TSX と呼ばれる言語を用いることが特徴の 1 つである．

また，近年の Web ページ開発では様々な理由から保守コストの抑制が求められている．その理由の 1 つとして，使用される言語の仕様やライブラリの頻繁な更新が挙げられる．例えば JavaScript の言語仕様は 2010 年から 2020 年までに 7 回改定されている．一方，C の言語仕様は同期間において 2 回しか改定されていない．言語仕様が更新されるとそれに伴って既存ソースコードの修正が必要になる場合があり，保守コストの増大に繋がる．また，もう 1 つの理由として，Web ページの要件の複雑化が挙げられる．例えば，Web ページの閲覧に用いられる端末の画面サイズは多様化しており，画面サイズに合わせたデザインを提供する必要がある．

一方，ソフトウェア保守を困難にする要因としてコードクローンの存在が指摘されている．コードクローンとはソースコード中に含まれている互いに一致，または類似しているコード片のことであり，主に，既存のソースコードのコピー・アンド・ペーストによって発生する．既存研究においてコードクローンの存在はソフトウェアの保守コストを増大させることが明らかになっている．例えば，修正が必要なコード片に，コードクローンが存在した場合，それら全てのコードクローンに対して修正が必要か検討する必要がある．Web ページ開発においてもコードクローンによって保守コストが増大している可能性はあるが，それに関する研究はほとんど行われていない．

そこで本研究では，React アプリケーションの 1 つである Grafana を対象にコードクローンの発生原因について調査した．調査にあたっては各コードクローンをソースコード中のコメントやコミットメッセージ，編集履歴など様々な観点から詳細に分析した．また，既存の検出ツールの多くは JSX や TSX に対応しないため既存ツールを拡張することでコードクローンを検出した．

調査の結果既存研究で対象となった一般的なアプリケーションとは異なり，Grafana では異なるプラットフォームへの対応のため発生するコードクローンやソースコードの改変時に安定性維持のため発生するコードクローンが存在しないことを確認した．また，Grafana のディレクトリ構成の影響により，異なるディレクトリ間で類似した処理を行うコードクローンが多く存在することを確認した．さらに，いくつかのコードクローンについて保守コストの増大につながる例を確認した．

主な用語

React
コードクローン
ソフトウェア保守

目 次

1　まえがき **4**

2　背景 **6**

2.1　TypeScript 6
2.2　React 6
2.3　近年の Web ページ開発 8
2.4　コードクローン 8
2.4.1　コードクローンの分類 9
2.4.2　コードクローンの発生原因 10

3　調査 **12**

3.1　調査目的 12
3.2　調査対象 12
3.3　調査手順 12
3.3.1　手順 1：対象プロジェクトのトークン化 13
3.3.2　手順 2：SourcererCC によるコードクローン検出 14
3.3.3　手順 3：発生原因の特定 14
3.3.4　手順 4: コードクローンの長さおよび分布の測定 15

4　結果 **16**

4.1　発生原因 16
4.2　発生原因とコードクローンの長さ 18
4.3　発生原因とコードクローンの分布 19

5　妥当性への脅威 **23**

5.1　発生原因の分類 23
5.2　異なるコードクローン検出ツール 23

6　まとめと今後の課題 **24**

謝辞 **25**

参考文献 **26**

1　まえがき

近年の Web ページ開発では Web UI フレームワークとして React が広く用いられている．以前の Web ページはサーバ側で動的に HTML を生成するか，事前に HTML で文書構造を定義し一部を Web ブラウザ上で JavaScript コードから変更するものが一般的であった．React は JavaScript を拡張した JSX や TSX と呼ばれる言語を利用し，Web ブラウザ上でほとんどの文書構造を動的に構築することが特徴の 1 つである．

また，近年の Web ページ開発では保守コストの抑制が求められている．その理由として次の 2 つが挙げられる．

1 つ目は使用する言語の仕様やライブラリの頻繁な更新である．例えば JavaScript の言語仕様は 2010 年から 2020 年までに 7 回改定されている．一方，C の言語仕様は同期間において 2 回しか改定されていない．また，コンポーネントライブラリの 1 つである material-ui のリポジトリには 2019 年から 2021 年にかけて約 5,700 回のコミットがあった．一方で C のライブラリである librdkafka のリポジトリには同期間で約 900 回のコミットしかなかった，このように JavaScript の言語仕様やそのライブラリは他の言語と比較して頻繁に更新される．更新に伴って互換性を損なう変更があった場合，既存ソースコードについて正常に動作するか検証し適切な修正を施す必要があり保守コストの増大に繋がる．

2 つ目は Web ページの要件の複雑化である．例えばかつて Web ページはデスクトップコンピュータで閲覧することが主流であり，比較的大画面に対応した単一のデザインを描画していた．一方，現在ではスマートフォンやタブレットのような小さな画面を持つ端末からも閲覧されることが多くなり，それぞれの画面サイズに合わせて文字サイズや要素の配置などを動的に最適化することが求められている．また，Web ページであってもネイティブなアプリケーションと同等の動作を実現することが求められている．このように Web ページに求められる機能は複雑化しており，それに伴って保守コストも増大している．

一方，既存研究においてソフトウェア保守を困難にする要因の 1 つとしてコードクローンの存在が指摘されている．コードクローンとはソースコード中に存在する互いに一致，または類似しているコード片のことであり，主に既存のソースコードのコピー・アンド・ペーストによって発生する．コードクローンが保守コストを増大させる原因の 1 つとして，バグの複製が挙げられる．ある 1 つのコード片にバグが存在した場合，そのコードクローン全てにも同様のバグが含まれている可能性が高い．開発者は全てのコードクローンについて修正を検討する必要があり，保守コストの増大に繋がる．しかし, これらコードクローンに関する研究はほとんどが一般的なアプリケーションを対象としており，Web ページにおけるコードクローンについて調査した研究はほとんど行われていない.

そこで本研究では, Kasper らが提案した発生原因の分類に基づき React アプリケーションの 1 つであ

る Grafana についてコードクローンの発生原因を調査した．加えて, コードクローンの長さや分布についても調査し発生原因との関連性を考察した．また，Grafana は JavaScript や TSX といった複数の言語を併用して記述されていたが，既存のコードクローン検出ツールの多くはこれらの言語に対応していない．本研究では調査にあたって SourcererCC を拡張することで React アプリケーションからコードクローンを検出するツールを作成した．

調査の結果 Grafana では既存研究で対象となった一般的なアプリケーションとは異なり，異なるプラットフォームへの対応のため発生するコードクローンやソースコードの改変時に安定性維持のため発生するコードクローンが存在しないことを確認した．また，Grafana のディレクトリ構成により，異なるディレクトリ間で類似した処理を行うコードクローンが多く存在することを確認した．さらにいくつかのコードクローンについて保守コストの増大につながる例を確認した．

以下，2 章では，本研究の背景として React や近年の Web 開発，コードクローンについて説明する．3 章では，調査対象や調査手法について説明する．4 章では，調査結果とその結果に対する考察を説明する．5 章では妥当性への脅威について述べる．最後に 6 章では，まとめと今後の課題について述べる．

2 背景

本章では本研究の背景として TypeScript や React，近年の Web ページ開発，コードクローンについて述べる．

2.1 TypeScript

Web ブラウザは HTML を用いて記述された HTML ドキュメントを解析し Web ページとして GUI に表示する．HTML ドキュメントは自身で動的に文書構造を変更する仕組みを持たないため，HTML ドキュメントから描画された Web ページもまた動的に変更することはできない．そこで，HTML ドキュメントを Document Object Model (以下 DOM) と呼ばれる木構造モデルに変換し，DOM に対して動的に要素の追加や削除などの操作を行うことで Web ページを変更する仕組みを Web ブラウザは備えている．また，これら DOM の操作を行うためのスクリプト言語として JavaScript が用いられている．

一方，JavaScript は静的な型付けやインターフェイスといった機能を持たないため，ツールによる開発支援が困難でプログラマの生産性低下に繋がることが指摘されている [12]．そこで，JavaScript を拡張した静的型付け言語として TypeScript[1] が開発されている．TypeScript は JavaScript に似た文法を持ち，全ての JavaScript コードを TypeScript コードとして扱うことが可能である．Web ブラウザは TypeScript コードを実行できないため，事前に TypeScript コードを JavaScript コードに変換する必要がある．

2.2 React

React は近年広く用いられている Web UI フレームワークの 1 つ [1] であり，DOM を操作することで動的に Web ページを生成・更新する．また，JavaScript や Typescript を拡張した言語である JSX や TSX を用いて UI コンポーネントを記述する．

TSX で記述された React アプリケーションの一部をソースコード 1 に示す．この例では 1 行目から 4 行目でコンポーネント `Hello` を定義している．`Hello` コンポーネントは描画時に親コンポーネントから `number` (数値) 型の値を `props.num` を通じて受け取り，それを 6 行目で `Hello React` の後に付加して表示する．7 行目から 13 行目では `Hello` コンポーネントを利用する親コンポーネント `App` を定義している．`App` コンポーネントは 10 行目で，6 行目で定義された `number` 型配列 `numbers` の各値を取り出し，`Hello` コンポーネントに渡して描画する．ソースコード 1 を JavaScript コードに変換し，Web ブラウザで実行することで図 1 の結果を得る．このように，開発者は React を用いることで DOM の更新操作を意識することなく宣言的に Web ページを記述することができる．

[1] https://www.typescriptlang.org/

ソースコード 1: React アプリケーションのコード例

```
type Props = {num: number};
const Hello = (props: Props) => {
    return <div>Hello React{props.num}!!</div>;
};

const numbers: number[] = [1, 2, 3];
const App = () => {
    return (
        <div>
            {numbers.map((num) => (<Hello num={num} />))}
        </div>
    );
};
```

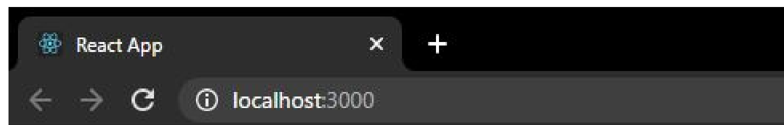

図 1: 実行結果

2.3　近年の Web ページ開発

近年の Web ページ開発では保守コストの抑制が求められている．その理由の 1 つとして Web ページ開発に用いられる言語の仕様やライブラリの頻繁な更新が挙げられる．例えば，JSX や TSX のもとになっている JavaScript の標準規格は 2010 年から 2020 年の間に 7 回メジャーバージョンアップされている [2]．一方で C 言語の標準規格は同期間で 2 回しかメジャーバージョンアップされていない [3]．また，React から利用可能なコンポーネントライブラリである material-ui[5] は 2019 年から 2021 年におけるコミット回数は約 5,700 回である．一方で C 言語のライブラリである librdkafka[4] の同期間におけるコミット回数は約 900 回であった．言語仕様やライブラリの更新に伴って以前のバージョンと互換性を損なう変更があった場合，ソースコード中から使用箇所を特定し適切な修正を施す必要があり，保守コストの増大に繋がる．

また，もう 1 つの理由として Web ページの要件の複雑化が挙げられる．例えば，かつて Web ページは主にデスクトップコンピュータのような比較的大画面で閲覧されていたため，そのサイズに合わせた単一のデザインを描画していた．一方，現在ではスマートフォンやタブレットのような小さい画面を持つ端末からも閲覧されることが多くなり，レスポンシブデザインと呼ばれる画面サイズに合わせたデザインに自動的に切り替える仕組みへの対応が求められている [16]．また，YouTube や Twitter のように Web ページであってもネイティブなアプリケーションと同等の機能やユーザビリティを実現することが求められている．さらにモバイル端末において Web ページをオフラインでも利用可能にする Progressive Web Apps と呼ばれる技術も普及している．これらを含む様々な理由から近年の Web ページに求められる機能は複雑化しており，それに伴って保守コストも増大している．

2.4　コードクローン

図 2 のように，ソースコード中に存在する互いに一致，または類似するコード片をコードクローンと呼び，互いにコードクローンであるコード片の組をクローンペアと呼ぶ．コードクローンは主に既存ソースコードのコピー・アンド・ペーストによって発生する．既存研究においてコードクローンは保守コストを増大させる要因の 1 つとして指摘されている [11]．例えば，あるコード片に欠陥が存在した場合，そのコード片のコードクローン全てに修正を検討する必要があり保守コストを増大させる．そのため, ソフトウェアの保守性を維持するためには開発者がコードクローンの存在を認識し, 適切に修正を加えることが重要である. しかし, ソースコード中から目視でコードクローンを探し出すことは困難であるためコードクローンを自動検出する手法が既存研究において提案されている [17, 18, 15, 21]. また, 検出されたコードクローンを可視化することで分析を支援するツールも開発されている [19, 22].

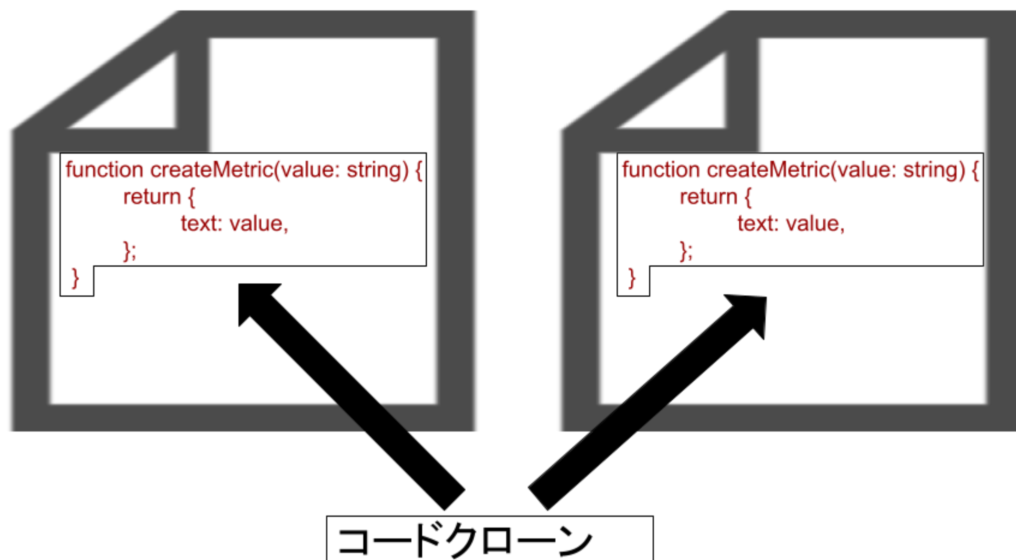

図 2: コードクローン

2.4.1 コードクローンの分類

手法によって検出されるコードクローンが異なるため [20]，コードクローンの普遍的な定義は存在しない．本研究では Roy らが提案したコードクローンの構文的な差異に基づく以下の 4 つの分類を用いる [14].

タイプ 1

空白，改行，コメントなどの違いを除いて完全に一致するコードクローン．

タイプ 2

タイプ 1 に加えて，識別子，リテラル，型の違いを除いて一致するコードクローン．

タイプ 3

タイプ 2 に加えて，文の挿入や削除，変更が行われたコードクローン．

タイプ 4

構文上は全く異なるが，同様の処理を行うコードクローン．

2.4.2 コードクローンの発生原因

コードクローンは主に既存ソースコードのコピー・アンド・ペーストによって発生するが，それには様々な理由が存在する．Kapser らはその動機に着目し，コードクローンの発生原因に基づく以下の 11 種類の分類を提案した [13].

1. ハードウェアバリエーション

新たなハードウェアに対応するドライバを作成する際，似たハードウェアに対応するドライバが既に存在している場合がある．しかし，ハードウェア間には機能に差異があることが多く，対応済みのハードウェアとの互換性を失うことなく，新たなハードウェアに対応することは難しい．そこで，既存ソースコードをコピーし，改変することで新たなハードウェアに対応するソースコードを作成することができる．このような場合に，ハードウェアバリエーションのコードクローンが発生する．

2. プラットフォームバリエーション

ソフトウェアを新たなプラットフォームに移植する際，プラットフォームに依存する機能を変更する必要がある．このとき，既存ソースコードをコピーし，プラットフォーム依存の処理を改変することでプラットフォームバリエーションのコードクローンが発生する．

3. 実験バリエーション

既存のソースコードに最適化や拡張などの変更を行いたいが，ソフトウェアの安定性を損ねたくない場合がある．このとき，変更前と変更後のソースコードの両方をソフトウェアに残し，ユーザにどちらを実行するか選択を委ねることができる．このような場合に，実験バリエーションのコードクローンが発生する．

4. 言語の表現力不足によるボイラープレート

ボイラープレートとは，言語の仕様上再利用が難しく様々な場所に組み込む必要があるソースコードのことである．このようなコードクローンが言語の表現力不足によるボイラープレートに分類される．

5.API/Library プロトコル

API やライブラリを利用する際に，事前にプログラム内で何らかの処理や設定が必要な場合がある．このような定形的な処理のコードクローンが API/Library プロトコルに分類される．

6. 言語やアルゴリズムの一般的なイディオム

イディオムとは特定の処理を実装する簡潔なソースコードの事である．このイディオムをソフト

ウェアの複数の箇所で利用した場合，言語やアルゴリズムの一般的なイディオムのコードクローンが発生する．

7. パラメータ化されたコード

パラメータ化とは類似した複数の関数を変数名やリテラルといった差異を引数として抽出し，1 つの関数に集約することである．パラメータ化可能であるがされていないコードクローンがパラメータ化されたコードに分類される．

8. バグ回避

ソフトウェアの開発中にバグを発見した際，アクセス権の問題などによって直接バグを修正することが困難な場合がある．このとき，バグが存在するソースコードをコピーして修正し，バグが存在する関数をオーバーロードすることで問題を解決することがある．このような場合，バグ回避のコードクローンが発生する．

9. 複製と専門化

何らかの処理を記述する際，類似した問題を解決する既存ソースコードをコピーし対象の処理に合わせて改変することがある．このような場合，複製と専門化のコードクローンが発生する．

10. 横断的関心事

横断的関心事とは，アクセス制御やロギングのような 1 つのモジュールにまとめることが困難な処理のことである．横断的関心事はソースコードの様々な箇所に記述する必要がある．このような処理がコードクローンとして検出された場合，横断的関心事に分類される．

11. 逐語的コード片

コードクローンを検出する際，for 文の 1 行目のようなそれ自体では意味を持たない微小なコードが検出されることがある．このようなコードクローンは逐語的コード片と分類する．

表 1: Grafana の詳細

言語	ファイル数	LoC
JavaScript	129	25,112
JSX	0	0
TypeScript	1,422	149,718
TSX	1,180	95,203

3 調査

3.1 調査目的

近年，Web ページは肥大化・複雑化しており，その開発や保守にかかるコストも増大している．一方，既存研究においてコードクローンはソフトウェアの保守コストを増大させる要因の 1 つとして指摘されており，Web ページの保守性に悪影響を及ぼしていると考えられる．しかし，既存研究では Web ページにおけるコードクローンの影響について調査されていない．そこで本研究では近年広く用いられている Web UI フレームワークの 1 つである React のアプリケーションにおけるコードクローンの実態について調査する．そのために，コミットログやソースコードのコメントなどから検出されたコードクローンの発生原因を特定する．また，コードクローンの長さおよび分布を計測し発生原因との関連性を考察する．

3.2 調査対象

本研究では，GitHub 上で OSS として公開されている React アプリケーションの 1 つである Grafana[9] を対象に調査を行う．Grafana は DB にクエリを発行して保存されたログやデータを取得し，リアルタイムに可視化するアプリケーションである．本研究で用いた Grafana のバージョンは 7.3.6 であり，全ファイル数は 4,951 個，LoC は 570,890 である，そのうち JavaScript，JSX，TypeScript，TSX それぞれのファイル数と LoC の内訳を表 1 に示す．ファイル数および LoC の計測には cloc 1.8.8[2] を用いた．

3.3 調査手順

本研究では Grafana のソースコードから以下の手順で必要なデータを得た．

[2] https://github.com/AlDanial/cloc

図 3: SourcererCC の動作

ソースコード 2: 変換前のソースコード

```
 function supportsTags(version: string): boolean {
    return isVersionGtOrEq(version, '1.1');
}
```

3.3.1 手順 1：対象プロジェクトのトークン化

調査のためにまず Grafana からコードクローンを検出するが，既存の検出ツールの多くは JavaScript や TypeScript，TSX に対応していない，また表 1 に示すように Grafana は複数の言語を用いて記述されており，より正確な調査のためにはこれらの言語間のコードクローンを検出する必要がある．そこで本研究では SourcererCC[17] を用いてコードクローンを検出する．SourcererCC は高い精度でタイプ 1 からタイプ 3 までのコードクローンを検出可能なツールである．SourcererCC の動作の概略を図 3 に示す．SourcererCC はソースコードを入力として与えるとソースコードをトークン列に変換しコードクローンを検出する．この場合，Java，C，C# のソースコードに対応する．また，あらかじめソースコードをトークン列に変換しそれを入力として与えることでコードクローンを検出することも可能である．この場合，ソースコードがどの言語で記述されていてもそれに依存することなくコードクローンを検出する．

本研究では Grafana のソースコードを SourcererCC 独自のフォーマットのトークン列に変換するトークナイザを作成した．Grafana に存在するソースコードの一部をソースコード 2 に，ソースコード 2 を

ソースコード 3: 変換後のトークン列

```
function@@::@@1,supportsTags@@::@@1,version@@::@@2, … ,'1.1'@@::@@1
```

トークナイザで変換した結果をソースコード 3 に示す．SourcererCC の入力として要求されるトークン列はソースコード中の各トークンとその出現回数の組によって構成される．例えば，ソースコード 3 先頭の `function@@::@@1` の場合 `@@::@@` をデリミタとしてその前に位置する `function` がトークンを，後に位置する 1 が `function` の出現回数を示す．また，SourcererCC はトークン列の 1 行を最小単位としてコードクローンを検出する．つまり，ソースコード中の各関数を 1 行のトークン列に変換した場合関数単位でコードクローンが検出され，ソースコード中の各ファイルを 1 行のトークン列に変換した場合ファイル単位でコードクローンが検出される．本研究では関数単位でコードクローンを検出するため，Grafana のソースコードの各関数を 1 行のトークン列に変換した．

トークナイザの作成には TypeScript の Compiler API[3] を用いた．Compiler API を用いることで JavaScript や JSX，TypeScript，TSX で記述されたソースコードに対して構文解析を行い抽象構文木を得る．抽象構文木中の各ノードは対応するトークンの情報をそれぞれ保持しており，抽象構文木をトラバースすることでトークナイザは各関数に含まれるトークンとその出現回数を計測する．そしてその結果をソースコード 3 のような形式で出力する．

3.3.2　手順 2：SourcererCC によるコードクローン検出

手順 1 でトークン化したソースコードを SourcererCC に入力し，コードクローンを検出する．SourcererCC はパラメータとしてコードクローンとして検出するコード片の最小類似度を指定できるが，本研究では 80% を指定した．

3.3.3　手順 3：発生原因の特定

手順 2 で検出されたコードクローンをクローンペア単位で 2.4.2 節で述べた発生原因で分類する．これに際して，各クローンペアについて以下の観点で詳細を調査し適切な分類を決定した．

- ソースコード中のコメント
- ソースファイルやソースファイルの存在するディレクトリの名前
- コミットメッセージ
- 編集履歴
- 関数の呼び出し元や呼び出し先といった参照関係
- コードクローンの前後のソースコード

[3]https://github.com/Microsoft/TypeScript/wiki/Using-the-Compiler-API

3.3.4　手順 4: コードクローンの長さおよび分布の測定

手順 2 で検出されたコードクローンについて長さおよび分布を測定する．コードクローンの長さはコードクローンとして検出されたコード片の長さを表すものである．コード片の長さは最後の行の行番号から最初の行の行番号を引いた結果に等しい．また，コードクローンの分布はクローンペアとなる 2 つのコード片の位置関係を表すものである．本研究ではコードクローンの分布として以下の 3 つを用いる．

同一ファイル内のクローンペア

2 つのコード片が同一ファイル内に存在するようなクローンペア.

同一ディレクトリ内のクローンペア

2 つのコード片が同一ディレクトリの異なるファイル内に存在するようなクローンペア.

異なるディレクトリ内のクローンペア

2 つのコード片が異なるディレクトリのファイル内に存在するようなクローンペア.

コードクローンの長さおよび分布は機械的に決定可能である．検出結果においてコードクローンは，コード片の存在するファイルのパス，開始行番号および終了行番号を用いて表現されるが，終了行番号から開始行番号を引くことでコードクローンの長さを得る．またコードクローンの分布については以下の手順で得る．

1. クローンペアとなる 2 つのコード片のファイルパスが互いに一致する場合，同一ファイルのクローンペアである．
2. 1 に当てはまらないクローンペアについて，2 つのコード片のファイルパスがファイル名を除いて互いに一致する場合，同一ディレクトリ内のクローンペアである．
3. 1 および 2 に当てはまらないクローンペアは異なるディレクトリ内のクローンペアである．

表 2: Grafana のコードクローンの発生原因

分類	クローンペア数	割合
1. ハードウェアバリエーション	0	0%
2. プラットフォームバリエーション	0	0%
3. 実験バリエーション	0	0%
4. 言語の表現力不足によるボイラープレート	2	1.7%
5.API/Library プロトコル	7	5.9%
6. 言語やアルゴリズムの一般的なイディオム	2	1.7%
7. パラメータ化されたコード	57	47.1%
8. バグ回避	0	0%
9. 複製と専門化	30	24.8%
10. 横断的関心事	0	0%
11. 逐語的コード片	23	19.0%
合計	121	100%

4 結果

調査の対象とした Grafana 7.3.6 には 4,141 個の関数が含まれていたため，それぞれを 1 行のトークン列に変換し合計で 4,141 行のトークン列を入力として SourcererCC に与えた．SourcererCC は 122 個のクローンペアを出力したが 1 つは設定ファイル間のクローンペアであったため，それを除外した残りの 121 個のクローンペアについて 3.3 節の手順に従い調査した．

4.1 発生原因

調査対象とした 121 個のクローンペアを発生原因で分類した結果を表 2 に示す．全クローンペアのうちパラメータ化されたコードに分類されたものが多く，次いで複製と専門化に分類されたものが多い．一方でハードウェアバリエーションやプラットフォームバリエーション，実験バリエーションなどはどのクローンペアも分類されなかった．Grafana に存在するコードクローンの発生原因の特徴を考察するため，この結果を既存研究の結果と比較する．Kasper らは C 言語で記述されたソフトウェアである Apache HTTP Server に存在するコードクローンの発生原因を分類した [13]．コードクローンの検出には本研究とは異なるツールを用いておりパラメータ設定などの条件も異なるため結果を直接比較することはできないが，本項では分類されたコードクローンの有無のみを比較した．その結果，特徴的な 2 つの相違点について述べる．

まず 1 つ目の相違点として，プラットフォームバリエーションに分類されたコードクローンが Apache HTTP Server には存在するが Grafana には存在しないことが挙げられる．プラットフォームバリエーションには新たなプラットフォームに移植する際既存ソースコードをコピーすることで発生するコードクロー

ンが分類される．Apache HTTP Server のような一般的なアプリケーションにおいては OS がプラットフォームに相当する．一方，Grafana においては Web ブラウザがプラットフォームに該当する．Web ブラウザは Web サイトの JavaScript コードに対し以下の 2 つの機能を提供する．

- DOM やデータ保存のような Web ブラウザの持つ機能を利用するための Web API[6]
- JavaScript コードを解析し実行する JavaScript エンジン

Web API は World Wide Web Consortium および Web Hypertext Application Technology Working Group によって，JavaScript エンジンが実行する JavaScript の言語仕様については ECMAScript として Ecma International によって標準化されているが，どの Web API をサポートしどのバージョンの JavaScript 言語仕様に対応するかは各ブラウザベンダが決定する．例えば Web サイトにおける音声操作の標準規格として Web Audio API が標準化されているが，全てのバージョンの Internet Explorer はこれをサポートしない [10]．一方，Chrome は Web Audio API を構成する機能のうち多くをサポートするが，サポートしないものもいくつか存在する [10]．また，ECMAScript 2018 では非同期オブジェクトに対して反復的な処理を行う構文として `for await...of` が追加されたが，Edge バージョン 18 の JavaScript エンジンはこの構文をサポートしない．このようにサポートされていない Web API や文法を JavaScript コードが利用しようとした場合，実行時にエラーが発生し Web サイトの正常な動作を妨げる要因となりうる．Web サイトはあらゆる種類の Web ブラウザやそのバージョンから利用される可能性があり，多くの環境で正常に動作させるためには前述のような Web ブラウザの差異に対処する必要がある．そのため，プラットフォームバリエーションに分類されるコードクローンが発生する可能性は高いが，Grafana には存在していない．

この理由として Polyfill と Babel[7] の利用が挙げられる．Polyfill は Web ブラウザがサポートしない Web API の代替実装を提供する JavaScript コードである．Polyfill を利用するためには設定された対象ブラウザに応じて事前に Polyfill を注入する方法と実行時に User Agent から不足している機能を判別し必要な Polyfill を外部サービスから取得する方法の 2 通りがある．Grafana では前者の方法を採用しており，様々な Polyfill を提供するライブラリの 1 つである `core-js`[8] を利用するようソースコードを改変する．これにより Web ブラウザごとのサポートされる Web API の差異に対処している．Babel は上位バージョンの JavaScript で記述されたソースコードを下位バージョンの JavaScript に変換するトランスパイラである．Grafana では Babel を用いてソースコードを主要ブラウザの最新の 3 バージョンでサポートされる JavaScript コードに変換する．これにより Web ブラウザごとのサポートされる JavaScript のバージョンの差異に対処している．

このように Grafana では Polyfill および Babel を利用することで Web ブラウザの差異を意識することなく開発されている．そのためプラットフォームバリエーションに分類されるコードクローンが存在し

なかったと考えられる．

次に，2 つ目の相違点として，実験バリエーションに分類されたコードクローンが Apache HTTP Server には存在するが Grafana には存在しないことが挙げられる．実験バリエーションにはソフトウェアの一部を改変する際，安定性を維持するため開発者が既存ソースコードをコピーして安定版と実験版を作成することで発生するコードクローンが分類される．

実験バリエーションに分類されるコードクローンが存在しない理由として Grafana が Git を用いて開発されていることが挙げられる．開発者がソースコードの一部を改変する際，安定版のブランチから派生して開発用のブランチを作成することができる．これにより開発用ブランチに対する変更は安定版ブランチには反映されないため，既存部分の安定性を維持したまま開発を続けることができる．Grafana では master ブランチを開発用ブランチとして，master ブランチから派生してマイナーバージョンのリリースごとに安定版ブランチが作成される．master ブランチに対する変更は他のブランチに反映されないため，マイナーバージョン用ブランチのソースコードの安定性は維持される．

また，Grafana が GitHub 上で開発されていることも実験バリエーションに分類されるコードクローンが存在しない理由として挙げられる．GitHub には他ユーザのリポジトリをコピーし自分のリポジトリとして使用するフォークと呼ばれる機能が存在する．フォークしたリポジトリに対する変更はそのリポジトリにのみ適用されるため，フォーク元のリポジトリが変更されることはない．フォークしたリポジトリに対する変更をフォーク元のリポジトリに反映させる場合，フォーク元リポジトリの所有者に変更箇所を通知しマージを依頼するプルリクエストと呼ばれる機能を用いる．Grafana リポジトリへのアクセス権限を持たない一般の開発者がバグ修正やドキュメントの更新，新機能の追加などのためソースコードを改変する際，Grafana リポジトリをフォークしそれに対して変更を加える．その後プルリクエストを作成し，Grafana リポジトリへのアクセス権限を持つ開発者に変更箇所のマージを依頼する．プルリクエストが承認されるまでフォークしたリポジトリに対する変更は Grafana リポジトリには反映されないため，Grafana リポジトリのソースコードの安定性は維持される．

このように Grafana では Git および GitHub を利用することで安定版と実験版のソースコードを分離している．そのため実験バリエーションに分類されるコードクローンが存在しなかったと考えられる．

4.2　発生原因とコードクローンの長さ

発生原因ごとのコードクローンの長さの分布を図 4 に示す．コードクローンが存在しない分類は図 4 において省略している．中央値は複製と専門化に分類されたコードクローンが最も長く，次いでパラメータ化されたコード，API/Library プロトコルが長い．逆に，言語の表現力不足によるボイラープレートや言語やアルゴリズムの一般的なイディオム，逐語的コード片に分類されるコードクローンは短い傾向にある．複製と専門化やパラメータ化されたコードなどに長いコードクローンが分類される傾向があるこ

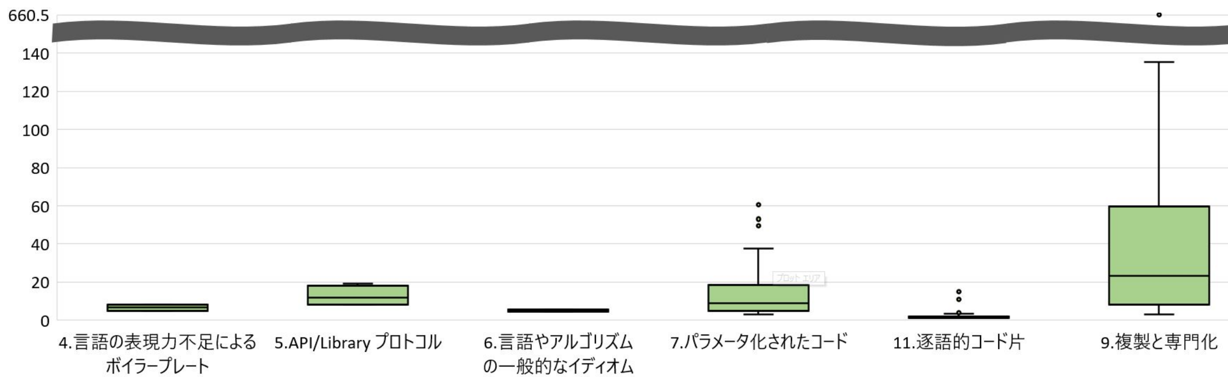

図 4: 発生原因ごとのコードクローンの長さの分布

とは，それぞれの定義より明らかである．

長いコードクローンについて詳細を調査したところ，多くが既存ソースコードをファイルごとコピーすることで発生したと推測された．その理由として，コードクローンとして検出されたコード片の前後も互いに類似する傾向があることが挙げられる．また，もう 1 つの理由としてコードクローンを含むファイルの初回コミット時の内容とコピー元ファイルの内容を比較したところほぼ全てが一致していたことが挙げられる．このようなコードクローンの例をソースコード 4 およびソースコード 5 に示す．これらはともに DB からログを取得するためのクラスであり，ソースコード 5 をコピー・修正してソースコード 4 が作成された．更にソースコード 4 とソースコード 5 は頻繁に同時に修正されており，保守コストの増大に繋がるコードクローンであることを確認した．

4.3 発生原因とコードクローンの分布

発生原因ごとのコードクローンの分布を表 3 に示す．この結果からパラメータ化されたコードに分類されるコードクローンは同一ファイル内に存在する傾向があり，複製と専門化に分類されるコードクローンは異なるディレクトリに存在する傾向があることが分かる．

本研究では，パラメータ化されたコードに分類されるクローンペアが同一ファイル内に多く存在する理由について確認することはできなかった．一方，複製と専門化に分類されるコードクローンが異なるディレクトリに存在する傾向がある理由として，ディレクトリごとソースコードをコピーし修正して利用していることが挙げられる．ソースコード 6 とソースコード 7 は複製と専門化に分類されたコードクローンであり，ソースコード 6 はディレクトリ `mysql` 内の，ソースコード 7 はディレクトリ `mssql` 内のファイルに存在する．これらについてコミットログなどをもとに詳細を調査したところ，ある機能に関連するファイルを集めたディレクトリ `mysql` をコピーし，類似機能のためのディレクトリ `mssql` が作成

ソースコード 4: 既存ファイルをコピーして作成されたファイル

```
  1: import _ from 'lodash';
  2: import appEvents from 'app/core/app_events';
...
 27: export class PostgresQueryCtrl extends QueryCtrl {
...
640:   handleQueryError(err: any): any[] {
641:     this.error = err.message || 'Failed to issue metric query';
642:     return [];
643:   }
644: }
```

されたことでこの 2 つのディレクトリ間のコードクローンが多く発生していた．また，ソースコード 6 とソースコード 7 は頻繁に同時に修正されており，保守コストの増大に繋がるコードクローンであることを確認した.

ソースコード 5: コピー元となったファイル

```
  1: import _ from 'lodash';
  2: import appEvents from 'app/core/app_events';
...
 27: export class MysqlQueryCtrl extends QueryCtrl {
...
640:   handleQueryError(err: any): any[] {
641:     this.error = err.message || 'Failed to issue metric query';
642:     return [];
643:   }
644: }
```

表 3: 発生原因ごとのコードクローンの分布

分類	同一ファイル	同一ディレクトリ	別ディレクトリ
1. ハードウェアバリエーション	0	0	0
2. プラットフォームバリエーション	0	0	0
3. 実験バリエーション	0	0	0
4. 言語の表現力不足によるボイラープレート	0	1	1
5.API/Library プロトコル	2	2	3
6. 言語やアルゴリズムの一般的なイディオム	1	0	1
7. パラメータ化されたコード	37	11	9
8. バグ回避	0	0	0
9. 複製と専門化	0	2	28
10. 横断的関心事	0	0	0
11. 逐語的コード片	0	6	17
合計	40	22	59

ソースコード 6: ディレクトリのコピーにより発生したコードクローンの例 1

```
  1: import _ from 'lodash';
  2: import { Observable, of } from 'rxjs';
  3: import { catchError, map, mapTo } from 'rxjs/operators';
  4: import { getBackendSrv } from '@grafana/runtime';
  5: import { ScopedVars } from '@grafana/data';
...
219:     rawSql = rawSql.replace('$__', '');
219:
219:     return this.templateSrv.variableExists(rawSql);
219:   }
219: }
```

ソースコード 7: ディレクトリのコピーにより発生したコードクローンの例 2

```
  1: import _ from 'lodash';
  2: import { Observable, of } from 'rxjs';
  3: import { catchError, map, mapTo } from 'rxjs/operators';
  4: import { getBackendSrv } from '@grafana/runtime';
  5: import { ScopedVars } from '@grafana/data';
...
192: targetContainsTemplate(target: any) {
193:     const rawSql = target.rawSql.replace('$__', '');
194:     return this.templateSrv.variableExists(rawSql);
195:   }
196: }
```

5　妥当性への脅威

5.1　発生原因の分類

本研究では，検出された全てのコードクローンについて筆者の主観により発生原因を分類したため誤りが含まれる可能性がある．しかし，より正確に発生原因を分類するため 3.3.3 項 で述べたように各クローンペアについて詳細に調査し適切な分類を決定するよう努めた．一方，コードクローンの長さおよび分布については 3.3.4 項で述べたように検出結果から機械的に得ることが可能であるため，分析者の違いが調査結果に影響を与えることはない．

5.2　異なるコードクローン検出ツール

本研究ではコードクローンを検出するために SourcererCC を用いた．SourcererCC 以外のツールを用いて本研究と同様の調査を行った場合，検出されるコードクローンの違いから異なる調査結果を得る可能性がある．しかし，SourcererCC は他の既存ツールと比較して高い精度でコードクローンを検出可能であることが報告されている [17].

6 まとめと今後の課題

本研究では React アプリケーションの 1 つである Grafana を対象にコードクローンの発生原因について調査した．調査にあたって Grafana は JavaScript や TypeScript，TSX といった複数の言語で実装されており既存の検出ツールでは対応不可能であったため，SourcererCC を拡張することでそれらの言語間のコードクローンを検出可能なツールを作成した．また，調査の結果 Polyfill や Babel の利用によりプラットフォームバリエーションに分類されるコードクローンが存在しないことを確認した．更にいくつかのコードクローンについて実際に保守コストの増大に繋がる例を確認した．

今後の課題として，以下の 2 点が挙げられる．

調査対象の追加

本研究では Grafana のみを対象として調査を行った．同様の手順で様々な React アプリケーションを対象に調査を行うことで，React アプリケーションに存在するコードクローンの傾向について確認できると考えている．

有害性の調査

本研究では一部のコードクローンについて有害性を調査し，実際に保守コストの増大に繋がる例を確認した．今後全てのコードクローンについて有害性を調査することで発生原因と有害性の関連を検証したいと考えている．また，Kasper らは C 言語で記述された 2 つのアプリケーションに対してコードクローンの発生原因とその有害性の関連を調査した [13]．この結果と比較することで React アプリケーション特有のコードクローンの有害性を明らかにし，保守コストの抑制に貢献できると考えている．

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 井上克郎 教授にはご多忙の中，研究活動及び研究室での生活に貴重な御意見及び御指導を賜りました．井上 教授に厚く御礼申し上げます．

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 春名 修介 特任教授には研究において適切な御助言を賜りました．春名 教授に厚く御礼申し上げます．

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 准教授には研究や発表における問題点の提示など，多くの御助言を賜りました．松下 准教授に厚く御礼申し上げます．

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 助教には研究や発表における問題点の提示など，多くの御助言を賜りました．神田 助教に厚く御礼申し上げます．

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 松島 一樹 氏には研究の方針から本論文の執筆に至るまで，直接の御指導及び御助言を賜りました．松島 氏に厚く御礼申し上げます．

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 伊藤 薫 氏，嶋利 一真 氏には研究室での生活や，発表について御指導していただくなど，日々研究に関するご協力を賜りました．両氏に厚く御礼申し上げます．

最後に，様々な御指導，ご助言を頂いた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井上研究室の皆様に厚く御礼申し上げます．

参考文献

[1] The State of JavaScript 2019. https://2019.stateofjs.com/front-end-frameworks/

[2] Ecma International. ECMAScript® 2020 Language Specification 11th edition (June 2020), 2020.

[3] C - Project status and milestones. http://www.open-std.org/jtc1/sc22/wg14/www/projects#9899

[4] edenhill/librdkafka. https://github.com/edenhill/librdkafka

[5] mui-org/material-ui. https://github.com/mui-org/material-ui

[6] JavaScript Web APIs. https://www.w3.org/standards/webdesign/script.html

[7] Babel · The compiler for next generation JavaScript. https://babeljs.io

[8] zloirock/core-js. https://github.com/zloirock/core-js

[9] grafana/grafana. https://github.com/grafana/grafana

[10] Web Audio API. https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

[11] M. Fowler. Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[12] A. Rastogi, N. Swamy, C. Fournet, C. Bierman, and P. Vekris. Safe & Efficient Gradual Typing for TypeScript, In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15), pp. 167–180, 2015.

[13] C. J. Kapser and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software, Empirical Software Engineering, Vol. 13, 645, 2018.

[14] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, Science of Computer Programming, Vol. 74, No. 7, pp. 470–495, 2009.

[15] C. K. Roy and J. R. Cordy. Nicad: Accurate Detection of Near-miss Intentional Clones Using Flexible Pretty-printing and Code Normalization. In Proceedings of the 16th International Conference on Program Comprehension (ICPC'08), pp. 172–181, 2008.

[16] C. Mullins. Responsive, Mobile App, Mobile First: Untangling the UX Design Web in Practical Experience. In Proceedings of the 33rd Annual International Conference on the Design of Communication (SIGDOC'15), pp. 1-6, 2015.

[17] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes. SourcererCC: Scaling Code Clone Detection to Big-Code, In Proceedings of the 38th International Conference on Software Engineering (ICSE'16), pp. 1157-1168, 2016.

[18] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and Accurate Tree-based Detection of Code Clones. In Proceedings of the 29th International Conference on Software Engineering (ICSE'07), pp. 96–105, 2007.

[19] M. Asaduzzaman, C. K. Roy, and K. A. Schneider. VisCad: Flexible Code Clone Analysis Support for NiCad. In Proceeding of the 5th International Workshop on Software Clones (IWSC'11), pp. 77–78, 2011.

[20] S. Bellon, R. Koschke, G. Antoniol, J. Krinke and E. Merlo. Comparison and Evaluation of Clone Detection Tools. IEEE Transactions on Software Engineering, vol. 33, No. 9, pp. 577-591, 2007.

[21] T. Kamiya, S. Kusumoto, and K. Inoue. A Multi-linguistic Token-based Code Clone Detection System for Large Scale Source Code. IEEE Transactions on Software Engineering, Vol. 28, No. 7, pp. 654–670, 2002.

[22] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In Proceedings of the 8th International Symposium on Software Metrics (METRICS ' 02), pp. 67–76, 2002.