CCX: SaaS 型コードクローン分析システム

松島 一樹　小池 耀　井上 克郎

コードクローンを検出する際，検出ツールの種類や設定するパラメータの値によって得られる検出結果が大きく変化することが知られている [5] [22]．より正確なコードクローン分析を行うためには, 一つの対象に対し異なる検出ツールやパラメータ値を用いて検出を行い, それらの検出結果を比較することが重要である．しかし個々のコードクローン検出ツールの実行環境を整備し実行結果を比較することは容易ではない．本研究では多様な検出ツールに対応した SaaS 型コードクローン分析システム CCX を開発した．CCX を利用することで，開発者は個別にインストールすることなく現在 5 つの検出ツールを利用でき，その検出結果の比較や分析を容易に行うことができる．

It is reported that the results of the code clone detection are drastically affected by clone detection tool and/or their detection parameters [5] [22]. However, it is not easy to set up execution environments for clone detection tools with different parameters and to compare the execution results. In this study, we have developed a SaaS-based code clone analysis environment, named CCX. By using CCX, developers can use currently five detection tools without installing them individually, and can easily compare and analyze their detection results.

1　はじめに

ソフトウェア開発の保守工程における大きな問題の 1 つとしてコードクローンの存在が指摘されている [4] [14]．コードクローンとはソースコード中に存在する互いに類似したコード片のことであり，主に既存ソースコードの複製によって発生する [9]．開発者はコードクローンを管理し，適切に修正することでコードクローンによる悪影響を軽減させる事ができる．

しかし，ソースコード中から手作業でコードクローンを検出することは困難であるため，コードクローンを自動で検出するための様々な手法が研究されている [11] [15] [16] [23]．また，検出結果を可視化することでコードクローン分析を支援するツールも開発されている [1] [2] [21]．開発者はこのようなツールを利用することで効率的なコードクローンの管理が可能である．

一方で，複数の既存研究において検出ツールの特性の違いやそのパラメータの違いによって，得られる検出結果が大きく変化することが明らかにされている [5] [22]．よって，プログラムの保守を行う際には, 複数の検出ツールやパラメータを組み合わせてコードクローンを検出し, 修正の影響範囲を慎重に特定することが重要である．しかし，以下のような問題から複数の検出ツールや分析ツールを同時に利用することは困難である．

問題点 1　インストールする際の導入コストと動作環境の違い．

問題点 2　検出パラメータの違い．

問題点 3　入出力仕様の違い．

本研究ではこれら既存ツールの問題点を解決した SaaS 型コードクローン分析システム CCX を開発した．CCX は Web クライアントを提供しておりブラウザからアクセスするだけで利用できるため，ローカルマシンにコードクローンの検出や分析のためのソフトウェアをインストールする必要がない．また，

CCX: SaaS-based Code Clone Analysis Environment.

Kazuki Matsushima, Yo Koike, 大阪大学大学院情報科学研究科, Osaka University.

Katsuro Inoue, 南山大学理工学部, Nanzan University.

コンピュータソフトウェア, Vol.39, No.4 (2022), pp.129–143.

[ソフトウェア論文] 2021 年 1 月 12 日受付.

CCX はサーバ上でコードクローン検出を行い，バックエンドとして複数の検出ツールを使用可能である．更に CCX の仕様に沿ったプラグインを実装しアップロードすることで，新たな検出ツールをバックエンドに追加できる．

上述した 3 項目の既存ツールの問題点を解決するために CCX に導入した様々な要素についてその設計の妥当性を調査した．

以降，2 では本研究の背景としてコードクローンついて述べる．3 では既存ツールの問題点について述べる．4 では CCX とその詳細について述べる．5 では CCX の設計の妥当性について調査する．6 では妥当性への脅威を述べ, 最後に 7 では本研究のまとめと今後の課題について述べる．

2 コードクローンと関連研究

ソースコード中に存在する，全く同一あるいは互いに類似したコード片をコードクローンと呼ぶ [16]．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やコード生成ツールによる自動生成によって発生する．一般的に互いにコードクローンとなる 2 つのコード片の組をクローンペアと呼び，コードクローンの同値類をクローンセットと呼ぶ．

複数の既存研究において，コードクローンはソフトウェア保守を困難にする要因のひとつとして指摘されている．Barbour らは論文 [4] において一貫した修正が行われていないコードクローンにはバグが混入する可能性が高いことを明らかにした．また，Mondal らは論文 [14] において複数のプロジェクトを対象にコードクローンの変更履歴を調査し，過去にバグ修正が行われたコードクローンのうち 18.42% がバグ伝播に関与していることを明らかにした．

開発者がコードクローンを管理し修正することで，上記のようなコードクローンによる影響を抑制しソフトウェアの保守性を維持できる．そのため，開発者がコードクローンに関する情報を認識することは重要である．

2.1 コードクローン検出・分析ツール

コードクローンを管理・修正することは，ソフトウェアの保守性を維持するために重要である．しかし，全てのコードクローンを手作業で検出することは非現実的であるため，効率的なコードクローンの検出・分析を目的としたツールに関する研究が数多く行われている [16]．

コードクローンの検出に関して CCFinder [11], NiCad [15]，SourcererCC [17] といったツールが提案されている．一般的に検出ツールはコードクローンをファイル名や行番号などの組み合わせで出力するため，検出結果を直感的に理解することは難しい．

そこで，検出結果を可視化して効率的なコードクローンの分析を支援するツールが提案されている．植田らは CCFinder の検出結果を散布図やメトリクスグラフとして可視化するツール Gemini を提案している [21]．また，Asaduzzaman らは NiCad などの検出結果を散布図やラジアルマップ，ツリーマップとして可視化するツール VisCad を提案している [1]．

2.2 検出結果の変化

同一のソースコード集合に対してであっても，コードクローン検出ツールごとの特性や検出パラメータによって検出されるコードクローンは大きく変化することが既存研究から明らかになっている．

Bellon らは論文 [5] において，CCFinder のデフォルトパラメータと調整したパラメータで検出されるコードクローンの正解集合に対する適合率と再現率を比較した．その結果，調整後のパラメータで得られた検出結果はデフォルトパラメータのものと比較して全く同じ再現率で適合率が 3 倍以上向上していた．

また，Wang らは論文 [22] において，6 つの検出ツールで検出されたコードクローンを行単位で比較した．その結果，ある 1 つの検出ツールで検出されたコードクローンの多くはほかの検出ツールでは検出されないものであった．また，全検出ツールで共通して得られるコードクローンは 10% 程度であった．

このように，ただ 1 つの検出結果だけではコードクローンの正確な分析は困難であり，異なる検出ツールやそのパラメータから得られた複数の検出結果を

分析することが重要である．

BigCloneEval は，IjaDataset と呼ばれる Java プログラムのデータセットに対して，コードクローン検出ツールの検出結果の再現率を計算するためのフレームワークである [19] [20]．このフレームワークは，特定のデータに対する再現率の算出に特化しており，データセットの変更や検出結果の詳細な調査などを行うことは困難である．

2.3 クラウドベースの検出システム

CloneSwarm は，AWS クラウド環境上で NiCad を検出ツールとして稼働させ，一般ユーザにコードクローン検出を提供するサービスである [2] [3]．検出結果はコードクローンの存在を大局的に可視化したり，また，詳細にコードレベルで表示することができる．git レポジトリを検出対象としてユーザに指定させるなど CCX との共通点もあるが，検出ツールが NiCad に固定されている，検出結果の比較ができない，など CCX に比べて機能が限られている．

3 既存ツールの問題点

正確な分析のためには異なる検出ツールやそのパラメータでコードクローンを検出することが重要であるが，様々な理由から複数の検出ツールを同時に利用することは困難である．

3.1 問題点 1: ツールをインストールする際の問題

ツールをインストールする際の問題としてツールの導入コストと動作環境の制約がある．

導入コストに関連する問題として，実行可能ファイルが配布されていない，またはドキュメントに動作要件が明記されていないという問題が挙げられる．表 1 の例では NiCad は実行可能ファイルが配布されておらず，利用するためにはユーザがソースコードをビルドしなければならない．また，Deckard ではビルドに必要なライブラリやラインタイムがドキュメントに記載されていないため，エラーメッセージから不足しているパッケージを読み取り適切にインストールする必要がある．

動作環境の制約に関連する問題として，ツールが特定の OS 上でのみ動作する，または共存不可能なバージョンのランタイムを要求するという問題が挙げられる．NiCad は Windows 環境をサポートしておらず，Cygwin や VirtualBox のような Windows 上に Linux 環境を構築するソフトウェアを別途インストールし，その中に NiCad をインストールする必要がある．また，CCFinderX の GUI フロントエンドである GemX はランタイムとして Java 6 を要求する一方で CCVolti は Java 8 以上 を要求する．異なるバージョンの Java は同時に利用できないため GemX と CCVolti も同時に利用できない．

3.2 インストールされたツールを利用する際の問題

3.2.1 問題点 2: 検出パラメータの違い

インストールされたツールを利用する際の問題の 1 つとして検出パラメータの違いが挙げられる．一般的に検出ツールは実行時に検出のためのパラメータを設定でき，例えば NiCad には類似度が閾値を下回るコードクローンを検出結果から除外するパラメータや出力単位を変更するパラメータなどが存在する．検出パラメータの種類や定義は検出ツールごとに異なるため GemX は CCFinderX 専用の，Clone Swarm [2] は NiCad 専用のパラメータ設定フォームを実装しており，オプションなどでフォームを変更し他の検出ツールに対応するような仕組みは備えていない．

3.2.2 問題点 3: 入出力仕様の違い

インストールされたツールを利用する際のもう 1 つの問題として入出力仕様の違いがある．

検出ツールへの入力には起動コマンドと検出パラメータの 2 つがあり，出力にはコード片の位置を示すファイルパスと範囲，コードクローンの出力単位，出力フォーマットの 4 つがある．

表 1 に挙げた 5 つの検出ツールについて入力仕様をまとめたものを表 2 に，出力仕様をまとめたものを表 3 に示す．入力仕様・出力仕様共にほとんどのツールで互換性がない．また，独自フォーマットの出力は構文が複雑で単純な正規表現によるパターンマッチでは必要なデータを抽出することが困難であ

表 1　検出ツールの配布形態と動作環境

検出ツール	配布形態	動作環境	ランタイム
CCFinderX+GemX [10]	実行可能ファイル	Windows, Ubuntu 9	Python 2.6, Java 6
CCVolti [23]	ソースコード	Windows	Java 8 以上
CCFinderSW [18]	実行可能ファイル	JVM	Java 8 以上
NiCad [15]	ソースコード	Linux	TXL
Deckard [8]	ソースコード	Linux	Python 2

る．VisCad では出力フォーマットごとにパーサを用意することで複数の検出ツールに対応しているが，検出パラメータの設定や起動コマンドの入力はユーザが手作業で行う必要がある．

4 CCX

4.1 概要

本研究では 3 で述べた問題点を踏まえ SaaS 型コードクローン分析システム CCX を開発した．

CCX は公開された Git リポジトリに対してコードクローンの検出および分析を行う．コードクローンの検出には様々な検出ツールおよびそのパラメータを設定することが可能である．検出結果は任意に参照でき，Web クライアントからコードクローンの分析が可能である．

CCX は問題点 1 について Web サービス化により解決する．従来の検出ツールは利用するためにまずインストールする必要があった．CCX はバックエンドでコードクローン検出を行うため，ツールをインストールすることなくブラウザからアクセスするだけで様々な検出ツールを利用可能である．

次に，問題点 2 については検出パラメータ定義ファイルの導入により解決する．検出パラメータ定義ファイルは検出ツールに与える検出パラメータの定義を記述したテキストファイルである．CCX はこの検出パラメータ定義ファイルから検出パラメータの設定フォームを動的に生成することで，検出ツールの種類に関わらず統一された UI を提供する．

最後に，問題点 3 については検出ツールプラグインの導入により解決する．コードクローン検出プラグインは既存の検出ツールを利用してコードクローン検出を行う実行可能ファイルである．コードクローン検出プラグインには共通の入出力仕様が定められており，内部では検出ツールの起動コマンドの生成や検出結果の変換などを行う．

4.2 アーキテクチャ

本節では CCX のアーキテクチャとその詳細について述べる．

図 1 に CCX のアーキテクチャを示す．CCX はコントローラノードとワーカノードという 2 種類のノードと Web クライアントから構成される．

コントローラノードは CCX のシステム内に 1 つだけ存在し API サーバやデータベース，検出ツールプラグインといったコンポーネントを含む．ワーカノードは検出ツールプラグインを利用して実際にコードクローン検出を行うためのノードであり，CCX のシステム内に 1 つ以上存在する．ワーカノード上ではプラグインマネージャが検出ツールプラグインを実行するための前処理と実行後の後処理を行う．Web クライアントは Web ブラウザから CCX を利用するためのフロントエンドである．

以降，CCX を構成する主なコンポーネントについて詳細を述べる．

4.2.1 API サーバ

API サーバはコントローラノードの中心となるコンポーネントであり，受信した API リクエストを実行するサーバである．API サーバが提供する API の例として，外部リポジトリのインポートや検出履歴の取得，検出結果の参照などがある．その他にコードクローン検出を実行するための API も提供するが，これに対するリクエストは API サーバではなくワーカ

表 2 検出ツールの入力仕様

検出ツール	起動コマンド	検出パラメータ
CCFinderX	./ccfx.exe d java ...	起動コマンドで設定
CCVolti	java -jar CCVolti.jar ...	起動コマンドで設定
CCFinderSW	./CCFinderSW d ...	起動コマンドで設定
NiCad	./nicad6 blocks java ...	起動コマンドおよび設定ファイルで設定
Deckard	./deckard.sh	設定ファイルで設定

表 3 検出ツールの出力仕様

検出ツール	ファイルパス	範囲	出力単位	フォーマット
CCFinderX	絶対パス	トークン番号	クローンペア	独自フォーマット (バイナリ)
	相対パス	トークン番号	クローンペア	独自フォーマット (バイナリ)
CCVolti	絶対パス	行番号	クローンペア	CSV
			クローンセット	独自フォーマット (テキスト)
CCFinderSW	絶対パス	行番号	クローンペア	JSON
		トークン番号	クローンペア	CCFinderX フォーマット
NiCad	絶対パス	行番号	クローンペア	XML
			クローンセット	XML
Deckard	相対パス	トークン番号	クローンセット	独自フォーマット (テキスト)

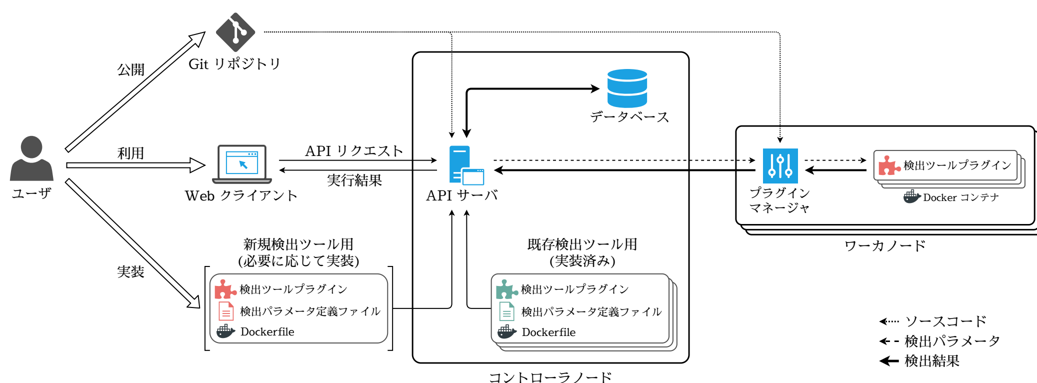


図 1 CCX のアーキテクチャ

ノード上で実行される．また，CCX を複数人で共有するために必要なユーザ認証機能も API サーバが提供する．

API は Web クライアント以外にも公開され，外部サービスと連携したコードクローンの分析が可能である．

4.2.2 検出パラメータ定義ファイル

検出パラメータ定義ファイルは検出ツールのパラメータ定義のリストを記述したテキストファイルである．

NiCad の検出パラメータの 1 つである `threshold` パラメータの定義を記述したものソースコード 1 に示す．1 行目では検出パラメータの表示名を設定している．2 行目では検出パラメータの種類を設定して

おり，float は threshold パラメータが実数値をとることを表す．ここには float の他に，ディレクトリ名をとることを表す directory や文字列をとることを表す input などが設定可能である．3–6 行目では検出パラメータの制約を設定しており，ここではデフォルト値が 0.3，最小値は 0 より大きく最大値が 1 に等しいという制約を設定している．7 行目では threshold パラメータの説明文を設定している．説明文は Web クライアントの検出パラメータ設定フォームに表示されるため，不慣れなユーザであっても意味を確認しながらパラメータ設定が可能である．

ソースコード 1　threshold パラメータの定義

```
1 label: Threshold
2 type: float
3 rule:
4   default: 0.3
5   max: 1
6   minExclusive: 0
7 description: Maximum difference threshold
        you are interested in.
```

4.2.3　プラグインマネージャ

プラグインマネージャはワーカノードに存在するコンポーネントであり，検出ツールプラグインを起動してコードクローン検出を行う．また，その前処理として検出対象のソースコードの取得や検出ツールプラグイン用 Docker コンテナの作成を，後処理として API サーバへ検出結果の送信などを行う．

API サーバとのデータの送受信は全てネットワーク経由で行われるため コントローラノードとワーカノードを異なるマシンに割り当てることも可能である．

4.2.4　検出ツールプラグイン

検出ツールプラグインは既存検出ツールを起動してコードクローン検出を行うコンポーネントである．CCX では検出パラメータと検出結果の共通フォーマットを JSON 形式で定めており，検出ツールプラグインはこの共通フォーマットと検出ツール独自のフォーマット間の変換を行う．また，この仕様に沿ったプラグインをユーザが実装しアップロードすることで，新たな検出ツールを CCX のバックエンドに追加できる．

図 2　検出パラメータ設定画面

検出ツールプラグインは同梱された Dockerfile をもとに作成された Docker コンテナ内で実行される †1．Docker を用いることによりランタイムの有無やそのバージョンといったワーカノード環境の制約を排除し，独立した専用の仮想環境内で検出ツールプラグインを実行できる．

4.3　Web クライアント

Web クライアントは Web ブラウザから CCX を利用するためのフロントエンドである．

Web クライアントの画面の例を図 2 に示す．

図 2 は Git リポジトリに対し，検出パラメータを設定しコードクローン検出を実行するための画面である．Target revision セクションでは，分析対象となる任意の版を，コミットのハッシュ値またはブランチ名で指定できる (デフォルトは HEAD)．Detector セ

†1 Dockerfile が同梱されない場合ワーカノード上のプロセスとして実行される．

クションから使用する検出ツールとそのバージョンを，Parameters セクションで検出パラメータを設定する．ユーザが選択する検出ツールとそのバージョンに応じて Parameters セクションに表示されるフィールドは変化する．

従来のツールでは特定の検出ツール専用のパラメータ設定用フォームを実装しており，他の検出ツールに対応させるために外部からフォームを変更するような仕組みを備えていなかった．CCX では検出パラメータ定義ファイルの記述をもとに動的にフォームを生成することで，ソースコードを改変することなく新たな検出ツールに対応する．例えばソースコード 1 の検出パラメータ定義からは図 2 の Threshold フィールドが生成される．

検出ツールやパラメータの異なる実行を簡単にまとめてできるよう，CCX では複数の検出パラメータ定義ファイルの逐次作成・実行する機能や，全分析ツールのデフォルト実行する機能を設けて，ユーザのパラメータ設定の手間の軽減を図っている．

4.4 機能

本節では CCX が提供する機能について説明する．

4.4.1 コードクローンの検出

前述のようにユーザは設定画面を通じて検出ツールの選択，およびその検出パラメータの設定を行い，コードクローン検出を実行させることができる．

CCX は，API サーバがクライアントからコードクローン検出の API リクエストを受信すると，検出パラメータ設定ファイルを参照しパラメータ値のバリデーションを行う．制約が満たされていれば，API サーバはプラグインマネージャへ検出パラメータを送信する．プラグインマネージャは検出パラメータを受信すると，検出ツールプラグインを起動してコードクローン検出を行う．

検出結果は API サーバを経由してデータベースに保存される．

4.4.2 検出結果の表示

CCX では検出結果のコードクローンの対（クローンペア）を ソースコード上に表示でき，視覚的に存在を確認することができる．表示のために API サーバは Git リポジトリからソースコードを，データベースから検出結果を取得する．Web クライアントは API サーバから受信したソースコードと検出結果から分析画面を表示する．

検出結果の表示画面の例を図 3 に示す．画面左上の a にはファイルツリーが表示される．a でファイルを 1 つ選択するとそのファイルに関連するクローンペアの一覧が b に，そのソースコードが c に表示され，存在するコードクローンは d に示すような縦のラインでハイライトされる．e には d と対になるコードクローンのソースコードが表示される．

4.4.3 2 つの検出結果の比較

CCX では 2 つの検出結果を比較，分析する以下のような機能を有する．

機能 1 文献[13] の方法により，検出結果の差異の大きさを，ファイル単位で散布図上に，赤 (100%) から黄色 (1%) までの階調色，または緑 (0%) で表示する．

機能 2 各ファイル単位に 2 つの検出結果に共通に現れるコードクローン，一方のみに出現するコードクローンのリストを表示する．

機能 3 散布図上の点（ファイルに対応）をクリックすることにより，対応するファイル中に存在するコードクローンのソースコードを，その対となるコードクローンとともに並列表示する．

図 4 は ある Java ライブラリに対する 2 種類の分析結果を比較する散布図の例で，左上を原点として縦横両軸に同じ順番でファイルが並べられ，各マス目（点と呼ぶ）が 1 つのファイルに対応する．有色点はいずれかの分析結果にコードクローンが存在することを示し，無色の点は存在しないことを示す．この例では，有色点のうち半分程度が赤で，対応するそれらのファイルでは検出結果の差異が大きいことが分かり，これらの点をクリックしてソースコードを精査することができる．また，有色点の残り半分程度は緑で，２つの検出結果が完全に一致しているのがわかる．

上記機能 1 により，２つの検出結果の差異の概要を視覚的に認識できるようになる．赤い点が多い場合は，差異が大きいファイルが多数あることを示している．一般にコードクローン部分のコードを変更す

```
Files
 > extras/src                                a
 v gson/src
   v main/java/co...
     > internal
     > stream
       Default...   1
       Gson.j...    7
       JsonEl...    2
       JsonSt...    1
       TypeA...     1
   > test/java/com/...
 > metrics/src/main...

Clones                                       b
Ln 401-407 #18
Ln 401-407 #19
Ln 680-687 #20
Ln 695-716 #21
Ln 739-746 #20
Ln 776-796 #21
```

```
gson/src/main/java/com/google/gson/Gson.java                         c
680   public void toJson(Object src, Type typeOfSrc, Appe
681     try {
682       JsonWriter jsonWriter = newJsonWriter(Streams.
683       toJson(src, typeOfSrc, jsonWriter);                 d
684     } catch (IOException e) {
685       throw new JsonIOException(e);
686     }
687   }
688
689   /**
690    * Writes the JSON representation of {@code src} o
691    * {@code writer}.
692    * @throws JsonIOException if there was a problem 
693    */
694   @SuppressWarnings("unchecked")
695   public void toJson(Object src, Type typeOfSrc, Json
696     TypeAdapter<?> adapter = getAdapter(TypeToken.ge
697     boolean oldLenient = writer.isLenient();
698     writer.setLenient(true);
699     boolean oldHtmlSafe = writer.isHtmlSafe();
700     writer.setHtmlSafe(htmlSafe);
701     boolean oldSerializeNulls = writer.getSerializeN
702     writer.setSerializeNulls(serializeNulls);
703     try {
704       ((TypeAdapter<Object>) adapter).write(writer, 
705     } catch (IOException e) {
706       throw new JsonIOException(e);
707     } catch (AssertionError e) {
708       AssertionError error = new AssertionError("Ass
709       error.initCause(e);
710       throw error;
```

```
gson/src/main/java/com/google/gson/Gson.java                         e
739   public void toJson(JsonElement jsonElement, Appe
740     try {
741       JsonWriter jsonWriter = newJsonWriter(Stream
742       toJson(jsonElement, jsonWriter);
743     } catch (IOException e) {
744       throw new JsonIOException(e);
745     }
746   }
```

図 3　検出結果表示画面の例

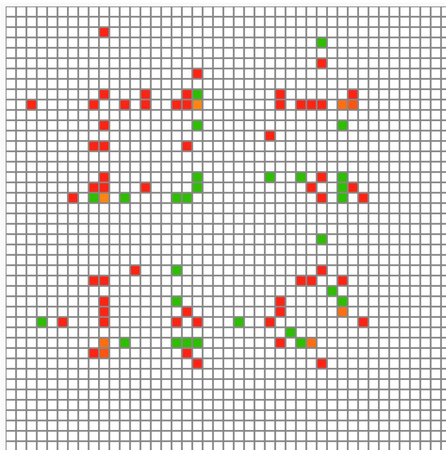


図 4　検出結果を比較した散布図の例

る場合は，コードクローン対の両方のコードを同時に変更する必要があり，細心の注意を要する [4] [14] が，検出結果の差異の大きいファイルでは，一種類のコードクローンの検出結果だけを頼りにコードの修正箇所を特定すると，必要な変更箇所を見落とす可能性が高まる．従って，赤い点のファイルの修正には注意が必要である．一方，緑の点が多い散布図は分析結果が安定しており，一種類の検出結果の参照で十分であることを示している．

機能 2 および機能 3 は、図 3 に示した単独の検出結果の表示画面と同様な画面構成を用いて実現しているが，両方の検出結果に現れるクローンペア，比較の基礎となる検出結果のみに現れるクローンペア，そして，基礎には現れずにもう一方の比較対象の検出結果のみに現れるクローンペアの 3 種に分けてリスト表示し，それぞれのクローンペアのファイル中での位置や行数がわかるようになっている．これによって，機能 1 で赤く表示されたファイルに修正を加えようとする場合，機能 2 や 3 を用いることによって，変更予定箇所がどこの分析結果のコードクローンに含まれているか，対になっているコード片はどこにあるか，それは具体的にどのようなコードで修正の必要があるか，などを知るのに役立つ．

5　設計の妥当性

CCX では Web サービス化，検出パラメータ定義ファイルの導入，検出ツールプラグインによる入出力フォーマットの統一により，3 で挙げた既存ツールの 3 つの問題点に対処した．本章では表 1 で挙げた 5 つの既存ツールのプラグインをそれぞれ実装し，問題点に対する CCX の設計の妥当性を調査する．これら 5 つのツールを選んだ理由は，我々が開発し，インストールや利用方法の知見がある（CCFinderSW, CCFinderX, CCVolti），長年にわたり利用され続け

ソースコード 2 **CCFinderX** をインストールする **Powershell** スクリプト

```
1 Invoke-WebRequest https://www.python.org/
      ftp/python/2.6.5/python-2.6.5.msi -
      OutFile python-2.6.5.msi
2 msiexec /quiet /i python-2.6.5.msi
3 Invoke-WebRequest http://www.ccfinder.net
      /download/ccfx-win32-en.zip -OutFile
       ccfx-win32-en.zip
4 Expand-Archive ccfx-win32-en.zip -
      DestinationPath ccfx-win32-en
```

ておりツールとして成熟している（Deckard, NiCad）などである．調査には CPU が Xeon E5-1660，メインメモリが 16GB のマシンを使用し，OS として Windows 10 Pro および Ubuntu 18.04 を利用した．

5.1 問題点 1 の解決: Web サービス化

既存ツールには，ツールをインストールする際の導入コストと動作環境の制約という問題があった．CCX では Web サービス化により Web ブラウザからアクセスするだけでコードクローンの検出や分析が可能である．本節では複数の検出ツールを利用する際の導入コストを単純なモデルで表現して定量化し，Web サービス化が導入コストの低減にどの程度貢献しているのか調査する．

5.1.1 導入コストの計測

導入コストを計測するメトリクスの 1 つとして *total number of distinct steps* (以下 NS) が提案されてる [12]．NS にはファイルのコピーやディレクトリの作成，設定の変更といったソフトウェアのインストール時に必要となる全ユーザ操作が含まれ，値が小さいほど良い．

計測のためにまず，ドキュメントに記載された手順に沿って各検出ツールをインストールするシェルスクリプトを記述した．ユーザ操作 1 回はシェルスクリプト 1 行に相当するものとして扱う．各ユーザ操作は以下の 4 種類に分類し，それぞれの行数を合計した全体の行数を NS とする．

$\boldsymbol{NS_{dist}}$ 配布物の取得に必要な操作

$\boldsymbol{NS_{build}}$ 検出ツールのビルドに必要な操作

$\boldsymbol{NS_{rt}}$ ランタイムのインストールに必要な操作

$\boldsymbol{NS_{others}}$ 上記に当てはまらない操作

Windows 10 Pro における CCFinderX のインストール手順を Powershell スクリプトとして記述した例をソースコード 2 に示す．

ソースコード 2 の 1–2 行目では CCFinderX の動作に必要なランタイムとして Python 2.6.5 をインストールし，3–4 行目では CCFinderX の配布物を zip ファイルとして取得し展開する．よって Windows 10 Pro における CCFinderX のインストールではユーザ操作の数が 4，つまり $NS = 4$ であり，その内訳は $NS_{dist} = 2$，$NS_{build} = 0$，$NS_{rt} = 2$，$NS_{others} = 0$ である．

5.1.2 各既存検出ツールの NS

まず，各検出ツールを単独でインストールする際の NS の値を計測する．計測のために検出ツールをインストールするシェルスクリプトを Windows 10 Pro 環境では Powershell スクリプトで，Ubuntu 18.04 環境では Bash スクリプトで記述した．

Windows 10 Pro 環境における各既存検出ツールの NS の値を表 4 に示す．NiCad と Deckard は Windows 10 Pro を直接サポートしないため，仮想マシン上に作成した Ubuntu 18.04 環境にインストールするために必要なユーザ操作の数を計測した． 表中の NS_u は VMware Workstation Player 16 をインストールし Ubuntu 18.04 仮想マシンを作成後，起動が完了するまでに必要なユーザ操作の数を表す．

Ubuntu 18.04 環境における各既存検出ツールの NS の値を表 5 に示す．CCFinderX と CCVolti は Ubuntu 18.04 を直接サポートしないため，表 4 の場合と同様に，仮想マシン上に作成した Windows 10 Pro 環境にインストールするために必要なユーザ操作数を計測した．表中の NS_w は VMware Workstation Player 16 をインストールし Windows 仮想マシンを作成後，起動が完了するまでに必要なユーザ操作の数を表す．

5.1.3 複数の検出ツールを同時に利用する際の NS

次に 5.1.2 で記述したシェルスクリプトを元に，CCX を使用せずに複数の検出ツールを同時に利用する際の NS の値を計測する．同じランタイムをイン

表 4 **Windows 10 Pro** 環境で検出ツールを利用する際の NS の値

検出ツール	NS_{dist}	NS_{build}	NS_{rt}	NS_{others}	$\boldsymbol{NS}$
CCFinderX	2	0	2	0	**4**
CCVolti	2	7	0	1	**10**
CCFinderSW	2	0	2	0	**4**
NiCad	2	7	0	NS_u	$\boldsymbol{9+NS_u}$
Deckard	2	3	0	NS_u	$\boldsymbol{5+NS_u}$

表 5 **Ubuntu 18.04** 環境で検出ツールを利用する際の NS の値

検出ツール	NS_{dist}	NS_{build}	NS_{rt}	NS_{others}	$\boldsymbol{NS}$
CCFinderX	2	0	2	NS_w	$\boldsymbol{4+NS_w}$
CCVolti	2	7	0	1+NS_w	$\boldsymbol{10+NS_w}$
CCFinderSW	2	0	2	0	**4**
NiCad	2	7	0	0	**9**
Deckard	2	3	0	0	**5**

ストールしているなどシェルスクリプト間で重複したユーザ操作がある場合は 1 操作とした．また，対象の既存検出ツールのプラグインが導入済みの CCX をローカルマシンにホストして利用する際の NS の値 NS_{ccx} も同時に計測する．ホスト OS をサポートしない検出ツールのプラグインを利用する場合は，ホスト OS にコントローラノードを，ゲスト OS にワーカノードをインストールし CCX システムを構築する．

Windows 10 Pro における計測結果を表 6 に，Ubuntu 18.04 における計測結果を表 7 にそれぞれ示す．各表の NS と NS_{ccx} を比較すると分かるように，多くの検出ツールの組み合わせで CCX を利用しない場合の NS の値が 10 を超えている．特に CCVolti と NiCad や NiCad と Deckard のようなソースコードをユーザがビルドする必要があり，更にホスト OS をサポートしない検出ツールの組み合わせでは，同時に利用するために非常に多くのユーザ操作が必要である．

CCX をローカルマシンにホストして利用すると，ほぼ全ての検出ツールの組み合わせで CCX を使用しない場合と比較して必要なユーザ操作数が少ない．これは Docker が検出ツールプラグインに同梱された Dockerfile の記述に沿って実行可能ファイルのビルドやランタイムのインストールを自動で行うため，ユーザ操作による検出ツールのセットアップが不要となったことが大きな要因である．

また，CCX をローカルマシンにホストせず既にインターネット上で公開されている CCX をブラウザから利用する場合，どの検出ツールの組み合わせでも利用するために必要なユーザ操作は，

1. CCX の Web サイトにアクセスする．
2. アカウントを作成する．

の 2 操作であり $NS = 2$ で簡便に利用できることが分かる．

上記のことから，CCX をローカルマシンにホストして利用する場合や CCX をブラウザから利用する場合はいずれも複数の既存検出ツールを利用する場合よりも導入コストが低く，より容易に利用可能であるといえる．

5.2 問題点 2 の解決: 検出パラメータ定義ファイルの導入

既存ツールにはインストールされた検出ツールを利用する際の問題の 1 つとして検出パラメータの違い (問題点 2) があった．CCX では，検出パラメータ定義ファイルに記述された各ツールのパラメータ定義か

表 6 **Windows 10 Pro** 環境で複数の検出ツールを同時に利用する際の NS の値

検出ツールの組み合わせ	NS_{dist}	NS_{build}	NS_{rt}	NS_{others}	$\boldsymbol{NS}$	$\boldsymbol{NS_{ccx}}$
CCFinderX + CCVolti	4	7	2	1	**14**	**6**
CCFinderX + CCFinderSW	4	0	4	0	**8**	**6**
CCFinderX + NiCad	4	7	2	NS_u	$\boldsymbol{13 + NS_u}$	$\boldsymbol{10 + NS_u}$
CCFinderX + Deckard	4	3	2	NS_u	$\boldsymbol{9 + NS_u}$	$\boldsymbol{10 + NS_u}$
CCVolti + CCFinderSW	4	7	0	1	**12**	**6**
CCVolti + NiCad	4	14	0	$1 + NS_u$	$\boldsymbol{19 + NS_u}$	$\boldsymbol{10 + NS_u}$
CCVolti + Deckard	4	10	0	$1 + NS_u$	$\boldsymbol{15 + NS_u}$	$\boldsymbol{10 + NS_u}$
CCFinderSW + NiCad	4	7	2	NS_u	$\boldsymbol{13 + NS_u}$	**7**
CCFinderSW + Deckard	4	3	2	NS_u	$\boldsymbol{9 + NS_u}$	**7**
NiCad + Deckard	4	10	0	NS_u	$\boldsymbol{14 + NS_u}$	**7**

表 7 **Ubuntu 18.04** 環境で複数の検出ツールを同時に利用する際の NS の値

検出ツールの組み合わせ	NS_{dist}	NS_{build}	NS_{rt}	NS_{others}	$\boldsymbol{NS}$	$\boldsymbol{NS_{ccx}}$
CCFinderX + CCVolti	4	7	2	$1 + NS_w$	$\boldsymbol{14 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCFinderX + CCFinderSW	4	0	4	NS_w	$\boldsymbol{8 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCFinderX + NiCad	4	7	2	NS_w	$\boldsymbol{13 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCFinderX + Deckard	4	3	2	NS_w	$\boldsymbol{9 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCVolti + CCFinderSW	4	7	0	$1 + NS_w$	$\boldsymbol{12 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCVolti + NiCad	4	14	0	$1 + NS_w$	$\boldsymbol{19 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCVolti + Deckard	4	10	0	$1 + NS_w$	$\boldsymbol{15 + NS_w}$	$\boldsymbol{12 + NS_w}$
CCFinderSW + NiCad	4	7	2	0	**13**	**8**
CCFinderSW + Deckard	4	3	2	0	**9**	**8**
NiCad + Deckard	4	10	0	0	**14**	**8**

ら設定用フォームを動的に生成することで統一されたUI をユーザに提供する．また，検出パラメータの説明の表示や入力値に対するバリデーションを行うことでユーザビリティを高めている．このように検出パラメータ定義ファイルは使いやすさに直結するコンポーネントであり，検出ツールのパラメータ定義を正確に記述することが重要である．本節では，検出パラメータ定義ファイルがどの程度正確に既存ツールの検出パラメータの制約を再現可能であるか調査する．

5.2.1 調査手順

まず，各検出ツールのドキュメントや CLI のヘルプ表示，検出パラメータ設定用ファイルのコメントの記載から検出パラメータとその制約を抽出した．抽出されたパラメータのうち，実行スレッド数を設定するパラメータや一時ファイルの保存先を指定するパラメータなど出力される検出結果に影響を与えないものは除外した．次に抽出された検出パラメータの制約を可能な限り正確に検出パラメータ定義ファイルに記述した．そして最後に各パラメータについて制約を再現できているか検証する．

5.2.2 結果

調査結果を表 8 に示す．

CCFinderX，CCVolti，NiCad の 3 つの検出ツールでは全てのパラメータの制約を正確に再現できた．一方，それ以外の検出ツールでは再現できなかったパラメータが 1 つ以上存在する．

CCFinderSW では b パラメータと antlr パラメータに関する制約が再現できなかった．これらは共に字句解析時の動作を指定するパラメータであり，同時に有効化できない．検出パラメータ定義ファイルではこのような排他的なパラメータの組み合わせを定義できないため，制約を再現できなかった．

Deckard では MIN_TOKENS，STRIDE，SIMILARITY の 3 つのパラメータに関する制約が再現できなかった．これらは整数値，整数値，実数値をとるパラメータで，それぞれ複数の値を指定することが可能である． その場合，各値を組み合わせたパラメータセットでコードクローンを検出する．例えば MIN_TOKENS に 20 を，STRIDE に 10 と 20 を，SIMILARITY に 0.8 と 0.9 を指定した場合，Deckard は (MIN_TOKENS, STRIDE, SIMILARITY) = (20, 10, 0.8), (20, 10, 0.9), (20, 20, 0.8), (20, 20, 0.9) の 4 パターンのパラメータセットでコードクローンを検出し，4 つの結果を出力する．CCX では検出パラメータセット 1 つに対して検出結果 1 つのみが紐づけられるため，複数の検出結果を出力するパラメータには対応できない．

上記のことから，検出パラメータ定義ファイルでは多くのパラメータの制約を再現可能である．一方で，CCX の設計上の都合から排他的なパラメータや複数の検出結果を出力するパラメータには対応できない．

5.3 問題点 3 の解決: 検出ツールプラグイン

既存ツールにはインストールされた検出ツールを利用する際の問題の 1 つとして入出力仕様の違い (問題点 3) があった．CCX では検出ツールプラグインが CCX の共通フォーマットと検出ツール独自のフォーマットを変換することで，様々な検出ツールをバックエンドに追加できる．本節では 5 つの検出ツールに対して実装した検出ツールプラグインの実装コストを調査する．

5.3.1 実装コストの計測

検出ツールごとの実装コストを調査するため各ツールの検出ツールプラグインを TypeScript で実装した．更に，言語ごとの実装コストを調査するため Deckard 用検出ツールプラグインを Kotlin，Go，Rust で実装した．

実装コストは cloc 1.88†2 を用いてコード行数から計測した．コーディングスタイルがコード行数を変化させないよう，セマンティクスに影響を与えない範囲で事前に改行や括弧の有無などを統一した．

5.3.2 結果

各ツールの検出ツールプラグインを TypeScript で実装したときの実装コストを表 9 に示す．

次に，Deckard 用検出ツールプラグインを TypeScript，Kotlin，Go，Rust で実装したときの実装コストを表 10 に示す．検出ツールプラグインの実装コストは，どの検出ツール・言語であっても空行やコメントを除いておおよそ 100–200 行程度であった．

CCX は共通フォーマットの記法として JSON を採用している．JSON は広く用いられているデータフォーマットの 1 つで，多くの言語でシリアライズ/デシリアライズのためのライブラリが存在する．CCX では，独自のデータフォーマットを定義するのではなく標準的なデータフォーマットを用いることで検出ツールプラグインの入出力処理の実装コストを削減している．また，検出ツールプラグインは基本的なファイル IO と検出ツールを起動するための外部プロセス実行機能で構成されるため様々な言語で実装可能である．上記のことから，検出ツールプラグインは言語を問わず実装可能であり，実装コストは多くとも 200 行程度である．

コードクローン検出ツールとして，以下の条件を満たせば，CCX のプラグインとして実装可能である．

- Docker 環境下の Winodws または Linux 上で稼働するスタンドアロン型コードクローン検出ツールである．
- 分析対象をファイルまたはディレクトリとして指定し，分析結果はクローンペアもしくはクローンセットの情報として出力する．
- 検出パラメータは実行コマンドのオプションとして指定するか，またはファイルとして読み込む機能がある．

例えば SourcererCC などもこの条件に適合し，CCX に組み込むことは可能である．

†2 https://github.com/AlDanial/cloc

表 8 制約を再現できなかったパラメータ数

検出ツール	抽出したパラメータ数	再現できなかったパラメータ数
CCFinderX	7	0
CCVolti	10	0
CCFinderSW	14	2
NiCad	12	0
Deckard	5	3

表 9 各検出ツールプラグインの TypeScript による実装コスト

検出ツール	LoC
CCFinderX	168
CCVolti	136
CCFinderSW	121
NiCad	155
Deckard	144

表 10 Deckard 用プラグインの実装コスト

言語	LoC
TypeScript	154
Kotlin	186
Go	143
Rust	197

6 妥当性への脅威

本研究の妥当性への脅威として次の 2 点が挙げられる．

1 点目は限られた種類の検出ツールを調査対象とした点である．本研究では 5 の結果を基に CCX は僅かな実装コストで既存検出ツールに対応可能であり，導入コストにおいて既存検出ツールより優れていることを示した．しかし，対象外の検出ツールではこれに反する結果が得られる可能性がある．

2 点目は文献[12] に基づいて 5.1 で導入コストをシェルスクリプトの行数というモデルで定量化した点である．調査においてシェルスクリプト 1 行をユーザ操作 1 回に相当するものとして扱っているが，1 行で実行される処理の複雑度は統一されていない．このため，導入コストが実際よりも過大に，あるいは過少に計測されている可能性がある． 導入コストに関するメトリクスとして以下のような ISO/IEC 9126 で定義されている．より正確に導入コストを計測するためには NS だけでなくこれらのメトリクスの利用も検討する必要があろう．

Ease of installation [6] ユーザの都合で手順を変更した場合にインストールが成功した割合．

Installation effort [7] 全インストール手順のうち自動化されているものの割合．

しかしこれらのメトリクスを具体的に実装し適用している例は知られていない．

7 まとめと今後の課題

本研究では SaaS 型コードクローン分析システム CCX を開発した．ユーザは CCX の Web サイトにアクセスするだけで複数の検出ツールを用いてコードクローンの分析ができる．また，ユーザが検出ツールプラグインを実装し CCX にアップロードすることで新たな検出ツールをバックエンドに追加することが可能である．検出ツールプラグインは基本的な言語機能で構成されるため様々な言語で実装可能である．

今後の課題として分析機能のプラグイン化が挙げられる．CCX のバックエンドはユーザが検出ツールプラグインを実装することで拡張できるが，フロントエンドである Web クライアントはそのような仕組みを備えていない．現在の設計では，ユーザは新たな分析機能を CCX の API を利用する外部サービスとして開発できるが，ユーザ認証やデータ取得といった分析以外の機能も開発する必要があり実装コストが増大する．分析機能をプラグイン化し Web クライアン

トに埋め込むことが可能になれば，Web クライアントが取得したデータを元に分析を行い結果を可視化する処理を実装するだけで CCX に分析機能を追加できる．また処理の一部をワーカノード上で実行することで複雑で大規模な分析に対応できる．

この機能を実現するためには既存の分析手法がどのように検出結果を利用するか調査し，分析プラグインのための共通仕様を定める必要がある．

CCX は https://sel.ist.osaka-u.ac.jp/webapps/ccx/ で公開されており，GitHub などでホストされた Git リポジトリからコードクローンを検出できる．また，そのソースコードは https://github.com/koi-y/ccx で公開されている．

謝辞 本研究は JSPS 科研費 18H04094 の助成を受けた．また本研究に関していろいろな助言いただいた関係者や査読者に深謝いたします．

参 考 文 献

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

[2] Bandi, V., Roy, C. K., and Gutwin, C.: Clone Swarm: A Cloud Based Code-Clone Analysis Tool, in *Proceedings of the 2020 IEEE 14th International Workshop on Software Clones (IWSC'20)*, IEEE, 2020.

[3] Bandi, V.: CloneSwarm, (2020). https://cloneswarm.usask.ca/.

[4] Barbour, L., Khomh, F., and Zou, Y.: An Empirical Study of Faults in Late Propagation Clone Genealogies, *J. Softw.: Evol. Process*, Vol. 25, No. 11 (2013), pp. 1139–1165.

[5] Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E.: Comparison and Evaluation of Clone Detection Tools, *IEEE Trans. Softw. Eng.*, Vol. 33, No. 9 (2007), pp. 577–591.

[6] ISO/IEC: Software Engineering – Product Quality – Part 2: External Metrics, Technical Report ISO/IEC TR 9126-2, ISO/IEC, 2003.

[7] ISO/IEC: Software Engineering – Product Quality – Part 3: Internal Metrics, Technical Report ISO/IEC TR 9126-3, ISO/IEC, 2003.

[8] Jiang, L., Misherghi, G., Su, Z., and Glondu, S.: DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones, in *Proceedings of the 29th international conference on Software Engineering (ICSE'07)*, IEEE Computer Society, 2007, pp. 96–105.

[9] Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S.: Do Code Clones Matter?, in *Proceedings of the 2009 IEEE 31st International Conference on Software Engineering (ICSE '09)*, IEEE Computer Society, 2009, pp. 485–495.

[10] Kamiya, T.: AIST CCFinderX. http://www.ccfinder.net/ccfinderxos.html.

[11] Kamiya, T., Kusumoto, S., and Inoue, K.: CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. Softw. Eng.*, Vol. 28, No. 7 (2002), pp. 654–670.

[12] Lenhard, J., Harrer, S., and Wirtz, G.: Measuring the Installability of Service Orchestrations Using the Square Method, in *Proceedings of the 2013 IEEE 6th International Conference on Service-Oriented Computing and Applications (SOCA'13)*, IEEE Computer Society, 2013, pp. 118–125.

[13] Matsushima, K. and Inoue, K.: Comparison and Visualization of Code Clone Detection Results, in *Proceedings of the 2020 IEEE 14th International Workshop on Software Clones (IWSC'20)*, IEEE, 2020, pp. 45–51.

[14] Mondal, M., Roy, B., Roy, C. K., and Schneider, K. A.: An Empirical Study on Bug Propagation through Code Cloning, *J. Syst. Softw.*, Vol. 158 (2019), pp. 110407.

[15] Roy, C. K., and Cordy, J. R.: NiCad: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization, in *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension (ICPC'08)*, IEEE Computer Society, 2008, pp. 172–181.

[16] Roy, C. K., Cordy, J. R., and Koschke, R.: Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach, *Science of Computer Programming*, Vol. 74, No. 7 (2009), pp. 470 – 495.

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

[18] Semura, Y., Yoshida, N., Choi, E., and Inoue, K.: CCFinderSW: Clone Detection Tool with Flexible Multilingual Tokenization, in *Proceedings of the 24th Asia-Pacific Software Engineering Conference (APSEC 2017)*, IEEE, 2017, pp. 654–659.

[19] Svajlenko, J.: IJaDataset 2.0, (2013). https://github.com/jeffsvajlenko/BigCloneEval.

[20] Svajlenko, J. and Roy, C. K.: BigCloneEval: A Clone Detection Tool Evaluation Framework with BigCloneBench, in *2016 IEEE International Con-*

ference on Software Maintenance and Evolution (ICSME), 2016, pp. 596–600.

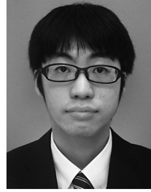
[21] Ueda, Y., Kamiya, T., Kusumoto, S., and Inoue, K.: Gemini: Maintenance Support Environment Based on Code Clone Analysis, in *Proceedings of the 8th International Symposium on Software Metrics (METRICS '02)*, IEEE Computer Society, 2002, pp. 67–76.

[22] Wang, T., Harman, M., Jia, Y., and Krinke, J.: Searching for Better Configurations: A Rigorous Approach to Clone Evaluation, in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, Association for Computing Machinery, 2013, pp. 455–465.

[23] 横井一輝, 崔恩瀞, 吉田則裕, 井上克郎: 情報検索技術に基づく細粒度ブロッククローン検出, コンピュータソフトウェア, Vol. 35, No. 4 (2018), pp. 16–36.

松 島 一 樹

2019年大阪大学基礎工学部情報科学科卒業．2021年同大学大学院情報科学研究科博士前期課程修了．同年楽天グループ株式会社入社．在学中，コードクローンに関する研究に従事.



小 池 耀

2021年大阪大学基礎工学部情報科学科卒業．現在，大阪大学大学院情報科学研究科博士前期課程2年．コードクローンに関する研究に従事.

井 上 克 郎

1979年大阪大学基礎工学部情報工学科卒業．1984年博士課程了．同年同大学基礎工学部助手．2002年情報科学研究科教授．2022年南山大学理工学部教授．工学博士．ソフトウェア工学, 特にソフトウェア開発手法, プログラム解析, 再利用技術の研究に従事.