社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE.

構文種別に着目したDockerfileのコードクローン検出手法

鶴　　智秋† 　中川　　将† 　柗本　真佑† 　肥後　芳樹† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{t-tsuru,t-nakagw,shinsuke,higo,kusumoto}@ist.osaka-u.ac.jp

あらまし　本論文では，Docker に対する Type-2 コードクローンの検出手法を提案する．Docker とは，コンテナ型仮想環境を実現するプラットフォームであり，サーバインフラを支える技術として注目されている．Docker では仮想環境実現の手順を，Dockerfile と呼ばれる一種のソースコードの形式で記述する．そのため，似た構造の繰り返しや重複といったコードクローンが必ず含まれる．本研究は，Dockerfile における Type-2 クローンの検出を目的として， Dockerfile 固有のネスト構造という性質に着目した検出手法を提案する．提案手法では，構文要素に対して適切な正規化を行い， Dockerfile 構文と Shell Script 構文に分離し，それぞれの構文ごとに接尾辞配列アルゴリズムを用いてコードクローン検出を行う．GitHub 上に公開されている約 2,000 個の Dockerfile を対象に適用実験を行い，高い適合率で Type-2 クローンを検出した．また，Dockerfile における定型処理を発見した．

キーワード　Docker，Dockerfile，コードクローン，コードクローン検出

1. はじめに

コンテナ仮想化と呼ばれるリソース効率に優れた仮想環境を実現するプラットフォームとして，Docker が着目されている [1]．Docker はインフラの構成自動化，IaC（Infrastructure as Code）を実現する技術の一つであり，コンテナ構築手順を Dockerfile と呼ばれるソースコードに記述する．インフラ構築手順のコード化により，誰がいつ実行しても全く同じインフラ環境を迅速に再現することが可能となる [2]．Docker は近年注目されている技術であり，多くの企業や OSS で採用されて始めている [3] [4]．一方で，Docker をはじめとする IaC の周辺技術は黎明期にあり，研究が未熟な領域が存在することも指摘されている [5]．

本研究では，Dockerfile で発生するコードクローン（以後，クローン）について考える．クローンとはソースコード内における一致，または類似コード片のことであり，様々な研究者によって盛んに研究されている [6] [7] [8]．クローンの検出により，利用 API の推薦 [9] や，冗長な記述に対するリファクタリングの推薦 [10]，同一変更を要する箇所に対する変更推薦 [11]，ライセンス違反の検出 [12]，といった様々な応用が可能となる．

これまでに Java や C などの手続き言語に限らず，様々な言語やファイルを対象としたクローン検出技術が研究されてきたが，Dockerfile に対するクローン検出の提案は少ない．一部，Oumaziz らによって Dockerfile を対象とした重複コード片の検出手法が提案されている [13]．この手法では，Dockerfile のトークンを分離し，転置索引と呼ばれるアルゴリズムを用いて重複箇所を検出する．しかし，この手法は完全一致する重複コード片（Type-1 クローン）のみを対象としており，変数名の違いを許容したクローン（Type-2）の検出は不可能である．

本研究では，Dockerfile に対する Type-2 コードクローンの検出手法を提案する．本研究の貢献は以下の通りである．

- Dockerfile における Type-2 クローンの検討と定義：Type-2 クローンでは，プログラムの振る舞いに影響を与えないトークンの違いを許容する．よってその検出には，変数名や関数名，定数の適切な正規化が必須である．Dockerfile では変数以外にも，ファイルパスやポート番号などの様々な定数要素が含まれる．さらには，Shell Script 構文内でのパラメタ順序やオプション名のエイリアスなどのように，振る舞いに影響を与えない記法が多数存在する．本研究では，正規化対象とするトークン要素を整理することで，Dockerfile における Type-2 クローンを検討する．
- 構文種別を考慮した Dockerfile におけるクローン検出手法の提案：Dockerfile は，単一ファイル内に複数言語の構文を記述可能なネスト言語である [1] [14]．コンテナ内部での処理を Shell Script 構文で記述し，それ以外のコンテナの外界との処理（ホスト OS からのファイルコピーやコンテナの公開ポート番号の指定など）を Dockerfile 構文で記述する．提案するクローン検出手法では，まず与えられた Dockerfile から抽象構文木を作成し，トークンの正規化を行う．さらに 2 種類の構文を分離し，接尾辞配列アルゴリズムを適用することで構文種別を考慮したクローン検出を実現する．
- 公開されている Dockerfile に対する Type-2 クローン検出，及び Dockerfile における定型処理の発見：GitHub 上の 223 リポジトリ内に存在する 1,897 個の Dockerfile を対象として，提案手法を用いた Type-2 クローン検出を行う．提案手法を用いた結果，適合率 95 % で Type-2 クローンを検出できた．また，空のコンテナからディストリビューションを構成するといった処理のように，Dockerfile における定型処理を発見した．

2. 準　　備

2.1 コードクローン

コードクローン（クローン）とは，ソースコード内における一致コード片または類似コード片を指す．一致コード片及び類似コード片の集合はクローンセットと呼ばれる．クローンは，一般的にその類似度の違いに基づき，以下の 3 種類に分類される [6].

- Type-1: 空白や改行，及びコメントの差異を除いて完全一致するクローン
- Type-2: 変数名や関数名などの識別子名の差異を除いて一致するクローン
- Type-3: 命令文単位での挿入や削除，変更が行われたクローン

これまでに，一般的なプログラミング言語において，多数のクローン検出手法が提案されている [6] [7] [8]．また，ビルドツールや要求定義書などといった，プログラミング言語以外の言語においてもクローン検出研究が行われている [15] [16] [17].

2.2 Docker

Docker は，コンテナと呼ばれる OS レベルの仮想化を実現するプラットフォームである．Docker は近年注目されている技術であり，IT 企業の 87 % で使用されているほか，多様な OSS で採用されている [3] [4].

Docker は IaC (Infrastructure as Code) と呼ばれるインフラの構成自動化を実現する重要な技術の一つであり，コンテナ構築手順をソースコードとして記述できる [2]．Docker コンテナ構築手順を記したソースコードを Dockerfile と呼び，Dockerfile から構成されるコンテナのテンプレートをイメージと呼ぶ．Dockerfile は表 1 に示す主な Dockerfile 構文で記述される．また，Dockerfile は RUN 命令に Shell Script 構文を内包できるという性質を持つ，ネスト言語でもある [1] [14].

2.3 Dockerfile における重複コード検出手法

Oumaziz らは，Dockerfile に対する重複コード検出手法を提案している [13]．彼らは，Dockerfile を構文解析したのち，Dockerfile 命令 6 種を 1 単位とした転置索引アルゴリズム [18] を適用して，重複コードを検出する．なお，Dockerfile において，単一の RUN 命令内部に複数の Shell Script 命令を記述するプラクティスが存在する．彼らの手法では，Shell Script 命令に内在する重複コードも検出するために，構文解析時に RUN 命令を内部の Shell Script 命令単位で分離させる．

表 1: 主な Dockerfile 構文の一覧

命令名	概要
FROM	コンテナの基となるイメージの指定
ARG	Dockerfile 内部における一時変数定義
ENV	コンテナ内部における環境変数定義
RUN	Shell Script で記述されるコンテナ構築手順
COPY	ホスト OS 上のファイルをコンテナへコピー
ADD	ホスト OS 上のファイルをコンテナへコピー tar アーカイブであるならばコンテナ上で展開
WORKDIR	ワーキングディレクトリ変更
USER	ユーザ名・グループ名の指定
EXPOSE	コンテナ外部に開放するポート番号の指定
ENTRYPOINT	コンテナ起動時に実行するコマンドの指定
CMD	コンテナ起動時に実行するコマンド引数の指定

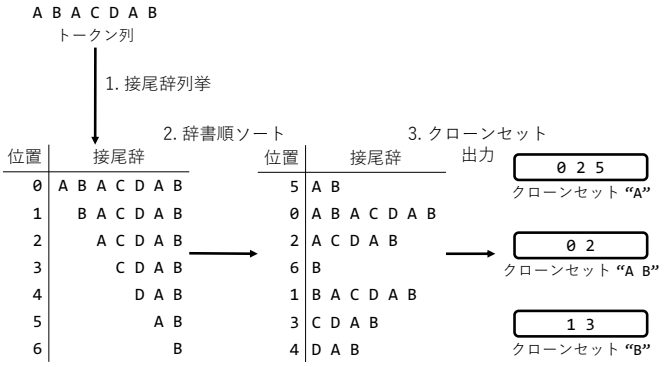

図 1: 接尾辞配列の構成手順及びクローン検出方法

Oumaziz らの手法は，トークンの正規化を適用していないため，文の類似ではなく完全一致，すなわち Type-1 クローンのみを検出対象としている．また，Dockerfile 構文と Shell Script 構文とを区別していないため，例えば，Shell Script 構文（RUN 命令の中身）の内容が本質的にクローンの関係にあるものの，片方の Dockerfile では RUN 命令の途中に ENV 命令等の Dockerfile 構文が挿入されている場合，クローンとして検出することはできない．さらに，転置索引アルゴリズムでは最短クローン長を固定しているため，長さがそれ未満のクローンを検出できない [18].

2.4 接尾辞配列アルゴリズム

接尾辞配列は，文字列検索アルゴリズムで使用されるデータ構造の一種であり，検索文字列における接尾辞を辞書順に並べ替えて得られる配列である [19]．接尾辞配列を用いることにより，長さが n の文字列から，長さが m である検索対象文字列の出現位置を $O(m + \log n)$ の時間計算量で求められる [19]．また，SA-IS 法を用いることで，接尾辞配列を $O(n)$ の時間計算量で構築できる [20]．接尾辞配列は，クローン検出の分野においても採用されている [21]．図 1 に，トークン列 “A B A C D A B” を基にした接尾辞配列の構成手順及びクローン検出方法を示す．まず，接尾辞列挙により入力トークン列の各接尾辞を配列に格納する（手順 1）．次に接尾辞配列を辞書順にソートする（手順 2）．最後に，接尾辞配列内の接尾辞同士に対し，前方一致する部分トークン列同士をトークンセットとして出力する（手順 3）．トークン列 “A B A C D A B” の例では，“A”，“A B”，“B” の 3 種類がクローンである．

3. 提 案 手 法

3.1 概　　要

提案手法の概要を図 2 に示す．提案手法の入力は複数の Dockerfile であり，出力は検出したクローンセットである．手法の流れについて図 2 に従って説明する．まず，入力として

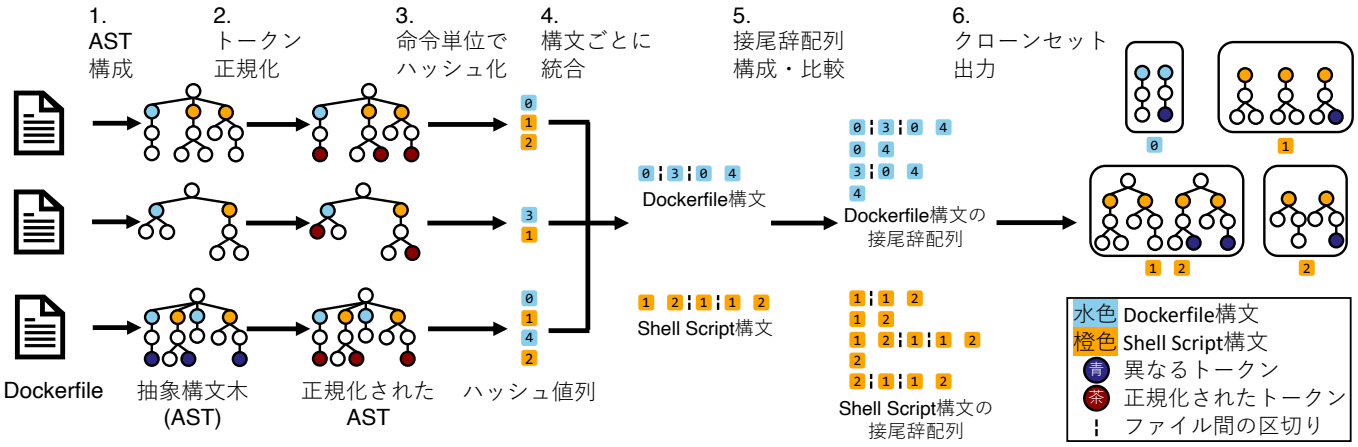

図 2: 提案手法の概要

与えた複数の Dockerfile から，それぞれに対応する抽象構文木を生成する（手順 1）．次に，各トークンの正規化を行う（手順 2）．この正規化処理により Type-1 クローンの検出から Type-2 クローン検出への拡張が可能となる．さらに命令単位のハッシュ化（手順 3），及び構文毎の統合（手順 4）を適用する．Dockerfile 構文と Shell Script 構文をあらかじめ分離し，続く接尾辞配列アルゴリズム（手順 5）とクローンセットの検出（手順 6）を行うことで，構文毎のクローンの検出を実現する．

以降の節では，手順 2 で正規化するトークン（3.2 節），手順 4 で Dockerfile 構文と Shell Script 構文にあらかじめ分離する理由（3.3 節），そして手順 5 で接尾辞配列アルゴリズムを採用する理由（3.4 節）について述べる．

3.2 正規化対象トークンの検討

Type-2 クローンの検出のためには，ソースコード内のトークンを適切に正規化する必要がある．Java や C 等を対象とした一般的なコードクローン研究においては，変数名や関数名などの識別子名を正規化対象とすることが多い [22]．変数名や関数名は一種のラベル情報であり，そのラベルが異なっても同一処理を実現可能なためである．Dockerfile でも一時変数（ARG 命令で宣言される変数）などが存在しており，正規化の対象となる．さらに，ソースコードの処理内容の同一性という観点では，Shell Script でのオプション順序（`ls -l .` と `ls . -l`）や，オプション名のエイリアス（`ls -list` と `ls -l`）など，同一処理を実現可能な別記法が多数存在する．これらを適切に正規化することで Dockerfile の Type-2 クローンが検出可能となる．

本研究では，正規化対象とするトークンを表 2，正規化対象としないトークンを表 3 の通り定義した．図 3 は，提案手法によるトークン正規化の例である．正規化対象トークンの一つに，Shell コマンドにおけるパラメタ・オプション順序がある．Shell コマンドは，コマンドの種別ごとにパラメタやオプションの順序が定められている．コマンドによっては，パラメタやオプションの記述順序の違いを許容している．そのため，パラメタやオプションが異なるにもかかわらず同じ動作を行うコマンドが，複数の Dockerfile 内で発生する．例えば，`RUN apt-get install --yes --no-install-recommends` と `RUN apt-get install --no-install-recommends --yes` はパラメタ・オプション順序が異なるクローン片の一例で

(注1)：`https://github.com/docker-library/rabbitmq/blob/master/Dockerfile-ubuntu.template`

表 2: 正規化対象とするトークン

トークン名	理由	正規化の例
一時変数名	コンテナ内の外部プロセスに影響を与えないため	`ARG tag=bionic` → `ARG $$VAR0=bionic`
パラメタ・オプション順序	意味的に同値なコマンドを実行できるため	`set -xe` → `set -e -x`
オプションのエイリアス	意味的に同値なコマンドを実行できるため	`rm -fr` → `rm --force --recursive`
ファイルパス	パスの差異が振る舞いに影響を与えないため	`cd webapps/convertigo` → `cd $$PATH0`
ユーザ・グループ	ユーザやグループの差異が振る舞いに影響を与えないため	`USER spark:spark` → `USER $$USER0:$$GROUP0`
コンテナのポート番号	ポート番号の差異が振る舞いに影響を与えないため	`EXPOSE 80` → `EXPOSE $$PORT0`
FROM 命令のタグ	タグによるコンテナの機能差がほとんど無いため	`FROM ubuntu:16.04` → `FROM ubuntu:$$TAG0`

表 3: 正規化対象としないトークン

トークン名	理由
環境変数名	コンテナ内の外部プロセスに影響を与えるため
FROM 命令のイメージ名	イメージごとに内包されるパッケージマネージャが異なるため

```
3  FROM ubuntu:18.04
   ...
16 ARG PGP_KEYSERVER=ha.pool.sks-keyservers.net
   ...
59 RUN set -eux; ¥
60     ¥
61     savedAptMark="$(apt-mark showmanual)"; ¥
62     apt-get update; ¥
63     apt-get install --yes --no-install-recommends ¥
       ...
72     ; ¥
73     rm -rf /var/lib/apt/lists/*; ¥
       ...
83     for key in $OPENSSL_PGP_KEY_IDS; do ¥
84         gpg --batch --keyserver "$PGP_KEYSERVER" --recv-keys "$key"; ¥
85     done; ¥
...
255    chown -R rabbitmq:rabbitmq "$RABBITMQ_HOME"; ¥
...
292 EXPOSE 4369 5671 5672 15691 15692 25672
```

(a) 正規化前

```
3  FROM ubuntu:$$TAG0
   ...
16 ARG $$VAR0=ha.pool.sks-keyservers.net
   ...
59 RUN set -e -u -x; ¥
60     ¥
61     $$VAR1="$(apt-mark showmanual)": ¥
62     apt-get update; ¥
63     apt-get install --no-install-recommends --yes ¥
       ...
72     ; ¥
73     rm --force --recursive $$PATH0/*; ¥
       ...
83     for $$VAR2 in $OPENSSL_PGP_KEY_IDS; do ¥
84         gpg --batch --keyserver "$$VAR0" --recv-keys "$$VAR2"; ¥
85     done; ¥
...
255    chown -R $$USER0:$$GROUP0 "$RABBITMQ_HOME"; ¥
...
292 EXPOSE $$PORT0 $$PORT1 $$PORT2 $$PORT3 $$PORT4 $$PORT5 $$PORT6
```

茶色 抽象概念への変換
水色 オプションのエイリアス
下線 オプションの分離・順序変更

$OPENSSL_PGP_KEY_IDSは環境変数であるため，正規化を施さない．

(b) 正規化後

図 3: 実際の Dockerfile (注1) に対するトークン正規化の例

```
21 # set up ssh keys
22 RUN mkdir -p /var/run/sshd ¥
23     && ssh-keygen -A ¥
       ...
30     && chmod 700 /root/.ssh ¥
31     && cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys
32
33 # download and install mlnx
34 RUN wget -q -O - http://www.mellanox.com/downloads/ofed/
   MLNX_OFED-4.6-1.0.1.1/MLNX_OFED_LINUX-4.6-1.0.1.1-rhel7.6-x86_64.tgz
    | tar -xzf - ¥
35     && ./MLNX_OFED_LINUX-4.6-1.0.1.1-rhel7.6-x86_64/mlnxofedinstall
    --user-space-only --without-fw-update --all --force ¥
36     && rm -rf MLNX_OFED_LINUX-4.6-1.0.1.1-rhel7.6-x86_64
```

(a) OpenFOAM イメージの Dockerfile (注3) から抽出したクローン片

```
11 # set up ssh keys
12 RUN mkdir -p /var/run/sshd ¥
13     && ssh-keygen -A ¥
       ...
20     && chmod 700 /root/.ssh ¥
21     && cp /root/.ssh/id_rsa.pub /root/.ssh/authorized_keys
22
23 # set up workdir
24 ENV INSTALL_PREFIX=/opt
25 WORKDIR /tmp/mpi
26
27 # download and install mlnx
28 RUN wget -q -O - http://www.mellanox.com/downloads/ofed/
   MLNX_OFED-4.6-1.0.1.1/MLNX_OFED_LINUX-4.6-1.0.1.1-rhel7.6-x86_64.tgz
    | tar -xzf - ¥
29     && ./MLNX_OFED_LINUX-4.6-1.0.1.1-rhel7.6-x86_64/mlnxofedinstall
    --user-space-only --without-fw-update --all --force ¥
30     && rm -rf MLNX_OFED_LINUX-4.6-1.0.1.1-rhel7.6-x86_64
```

(b) mpibench イメージの Dockerfile (注2) から抽出したクローン片

図 4: Shell Script 命令列に Dockerfile 命令列が挿入された Type-1 クローンの一例

ある．

3.3 構文種別の分離

Dockerfile は Shell Script 構文を内包したネスト言語である [1] [14]．そのため，Dockerfile でトークンを正規化するならば，Dockerfile 構文における抽象構文木と Shell Script 構文の抽象構文木をそれぞれ構成しなければならない．

また，Dockerfile は，一方の構文で記述された命令列に対して，もう一方の構文で記述された命令及びその構文の命令列を挿入できる．例えば，Dockerfile 命令列に Shell Script 構文を内包した RUN 命令を挿入できる．一方，RUN 命令が内包する複数の Shell Script 命令列は，Shell Script 命令単位で分離可能である．したがって，RUN 命令を Shell Script 命令単位で分離して，Shell Script 命令列に Dockerfile 命令及び Dockerfile 命令列を挿入できる．一般的なプログラミング言語と同様に，Dockerfile における命令文単位の修正で生じる上記のような類似コードは，Type-3 クローンとして定義可能である．しかし，各構文に着目すると，構文ごとで命令文単位での修正は行われていない．したがって，本研究では，Dockerfile がネスト言語である特徴に着目し，構文種別が異なる命令列単位の修正によって発生する類似コード片は Type-2 クローンに分類する．

図 4 は，Shell Script 命令列に Dockerfile 命令列が挿入されたクローンの一例である．mpibench イメージ Dockerfile (注2) (図 4(b)) の 23 行目 ～ 25 行目には，OpenFORM イメージ Dockerfile (注3) (図 4(a)) で出現しなかった Dockerfile 命令列が挿入されている．提案手法では，図 4 のようなクローンが検出可能である．

3.4 接尾辞配列の構成及び比較

Oumaziz らは，Dockerfile 開発者が長さが 6 未満の命令列も必要としていると述べている [13]．本研究では，接尾辞配列は可変長クローンを検出可能であるという特徴に着目した [21]．提案手法では，Dockerfile 命令及び Shell Script 命令 1 個ごとにハッシュ化を施しているため，任意の長さにおける Dockerfile 命令列や Shell Script 命令列をクローン片として検出できる．

なお，提案手法では，全ての Dockerfile のハッシュ値列を構文種別ごとに統合した巨大なハッシュ値列に対して，接尾辞配列アルゴリズムを適用する．単純な接尾辞配列アルゴリズムでは，異なるファイル間で統合されたコード片をクローンとして検出するため，ハッシュ値ごとに，そのハッシュ値の基となった構文が含まれている Dockerfile の情報を含める必要

(注2)：https://github.com/Azure/batch-shipyard/blob/master/recipes/mpiBench-Infiniband-MPICH/docker/Dockerfile
(注3)：https://github.com/Azure/batch-shipyard/blob/master/recipes/OpenFOAM-Infiniband-OpenMPI/docker/Dockerfile

```
FROM scratch
...
ADD rootfs.tar.xz /
CMD ["/bin/bash"]
```

(a) CRUX イメージ (注4) の Dockerfile から抽出したクローン片

```
FROM scratch
...
ADD smgl-stable-0.62-docker-x86_64.tar.xz /
CMD ["/bin/bash"]
```

(b) Source Mage イメージ (注5) の Dockerfile から抽出したクローン片

図 5: 提案手法で発見した定型処理（Type-2）

がある．

4. 実　　験

4.1 実験目的

実験の目的は，提案手法による Dockerfile の Type-2 クローン検出能力の確認である．比較対象としては，Oumaziz らによる Type-1 クローン検出手法 [13] を用いる．以降では，この手法を既存手法と呼ぶ．計測指標は，検出クローンの個数と適合率であり，Type-2 クローンは Type-1 クローンを包含するため提案手法の検出数が増えると期待される．適合率とは，検出したクローンのうち，実際にクローンである候補の割合である．適合率が高いほど，よいクローン検出手法と言える．なお対象とした Dockerfile における真のクローンの集合は未知であるため，検出したクローンセットから各手法ごとでランダムに 101 個抽出し，目視確認により適合率を算出する．

4.2 実験対象

GitHub 上で人気がある 223 リポジトリのうち，それらのリポジトリ内に存在する 1,897 ファイルを対象とする．なお，Docker ではテンプレートファイルにコンテナ構築の全体の流れを記述し，ディストリビューションやバージョンの細かな差異を可変変数で定義する，というプラクティスが存在する．このテンプレートを実行，あるいはコンパイルすることで目的の Dockerfile を自動生成する．この自動生成された Dockerfile は Type-2 クローンの関係にあるが，自動生成を理由としたクローンであり検出対象とすべきではない．よって，テンプレートファイルが存在するリポジトリに対しては，そのリポジトリに存在する Dockerfile ではなく，テンプレートそのものを対象とする．

多くの Dockerfile では Shell Script 構文として Bash shell を採用している [1] [14]．したがって，本実験では Shell Script 構文を Bash に限定する．また，抽象構文木構成のために構文解析を行う Bash コマンドを，Henkel らが調査した Dockerfile で頻繁に用いられる上位 50 種類に限定する [1] [14]．なお，クローン検出において，転置行列の大きさは通常 7 または 10 以下でなければならない [18]．そのため，既存手法における転置索引の大きさは，彼らの論文で採用された値である 6 とする．

4.3 実験結果

4.3.1 検出できるクローンの個数及び精度

表 4 は，実験で検出できたクローン片やクローンセット，適合率を載せた実験結果である．まず，既存手法と正規化ありの提案手法について比較すると，提案手法が検出したクローン片の数及びクローンセット数は，既存手法で検出できたクローン片及びクローンセット数よりも多い．提案手法では，トークンを適切に正規化して，接尾辞配列を用いて任意の長さを持つクローン片を検出できる．そのため，提案手法が既存手法より多くのクローンを検出できるという期待にそった結果となった．次に，正規化を施した提案手法と正規化を施さない提案手法を比較すると，正規化を施した提案手法の方が，正規化を施さない提案手法よりも検出できるクローン片の数及びクローンセット数が少ない．正規化ありの提案手法（Type-2）は正規化なしの提案手法（Type-1）を包含するためクローンの検出数が多い，という期待を反する結果となった．その原因として，正規化前のトークン名が同じであるにも関わらず，正規化処理によって抽象化されたトークン名に差異が生じたためと考えられる．そのため，クローン片の数及びクローンセット数が減少しないように改善した正規化処理の提案は今後必須である．最後に，既存手法と正規化ありの提案手法の適合率について比較すると，正規化を施した提案手法の適合率は既存手法の適合率より低い．一般的に正規化を施すほど適合率は下がりやすい．Type-1 クローンよりも Type-2 クローンが，Type-2 クローンよりも Type-3 クローンの方が検出は困難となる．しかしながら，提案手法は適合率 95 % と高く，Type-2 クローン検出手法として有用であると言える．

4.3.2 検出した Type-2 クローンの一例

本実験では，提案手法で検出した Type-2 クローンの内，Dockerfile における定型処理を発見した．図 5 は，提案手法で発見した，定型処理を含む Type-2 クローンの一例である．どちらの Dockerfile も，scratch と呼ばれる空のコンテナへ，tar アーカイブをホスト OS からコピーし，その tar アーカイブを

(注4)：https://github.com/cruxlinux/docker-crux/blob/master/Dockerfile
(注5)：https://github.com/vaygr/docker-sourcemage/blob/master/stable/Dockerfile

表 4: 検出できたクローン数及び適合率

	既存手法 [13]	提案手法（正規化なし）		提案手法（正規化あり）	
		Dockerfile 構文	Shell Script 構文	Dockerfile 構文	Shell Script 構文
クローン片数	24,687	93,435	211,104	94,153	207,748
クローンセット数	4,630	18,884	56,118	18,757	54,369
適合率	100.00 %	100.00 %	100.00 %	95.05 %	98.02 %

コンテナ上で展開して，新たなディストリビューションを構成する．

5. おわりに

本研究では，Dockerfile における Type-2 クローンの定義，及び Dockerfile 固有の文法構造に着目した Type-2 クローンの検出手法を提案した．提案手法を用いて，223 個の GitHub リポジトリ上に存在する 1,897 ファイルに対して実験を行ったところ，Type-2 クローンを検出できた．また，提案手法の適合率が Dockerfile 構文及び Shell Script 構文のそれぞれで 95 % 以上であったため，提案手法が有効であるといえた．

本研究の今後の課題としては以下が考えられる．

- トークン正規化による検出クローンの減少：4.3.1 節で述べた通り，本研究の実験では，トークン正規化によって検出できるクローンが減少した．そのため，この問題を改善する手法の提案は今後必須である．1 つの解決策として，一次変数同士を同一視してクローン検出するというように，正規化した同種トークンを区別しない方法が考えられる．しかし，この解決策ではトークン種別以外を同一視するため，適合率が低くなると考えられる．
- 構文を分離しない Type-2 クローン検出：提案手法では，両構文を分離して各構文ごとにクローンを検出しているため，Dockerfile 命令列と Shell Script 命令列の双方を含むクローンを検出できない．Dockerfile 構文と Shell Script 構文が一式となるクローンも活用できると考えられるため，両構文を含有するクローンも検出する必要がある．
- Type-3 クローンへの拡張：Dockerfile には，パッケージマネージャの違いにより発生するクローンが存在する．パッケージマネージャによって必要な Shell Script コマンドが異なるため，文単位で修正されたクローンを検出する手法が必要である．

謝辞 本研究の一部は，日本学術振興会科学研究費補助金基盤研究（B）（課題番号：18H03222）の助成を得て行われた．

文　献

[1] J. Henkel, C. Bird, S.K. Lahiri, and T. Reps, “A dataset of dockerfiles,” In Proceedings of International Conference on Mining Software Repositories, pp.528–532, 2020.

[2] Y. Jiang and B. Adams, “Co-evolution of infrastructure and source code-an empirical study,” In Proceedings of Working Conference on Mining Software Repositories, pp.45–55, 2015.

[3] Portworx, “Annual container adoption report,” `https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf`, 2019. [Online; accessed 18. Jan. 2021].

[4] J. Cito, G. Schermann, J.E. Wittern, P. Leitner, S. Zumberi, and H.C. Gall, “An empirical analysis of the docker container ecosystem on github,” In Proceedings of International Conference on Mining Software Repositories, pp.323–333, 2017.

[5] R. Jabbari, N. binAli, K. Petersen, and B. Tanveer, “What is devops? a systematic mapping study on definitions and practices,” In Proceedings of the Scientific Workshop Proceedings of XP2016, pp.1–11, 2016.

[6] 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.

[7] A. Sheneamer and J. Kalita, “A survey of software clone detection techniques,” International Journal of Computer Applications, vol.137, no.10, pp.1–21, 2016.

[8] H. Min and Z. Li Ping, “Survey on software clone detection research,” In Proceedings of International Conference on Management Engineering, Software Engineering and Service Sciences, pp.9–16, 2019.

[9] I. Keivanloo, J. Rilling, and Y. Zou, “Spotting working code examples,” In Proceedings of International Conference on Software Engineering, pp.664–675, 2014.

[10] N. Yoshida, S. Numata, E. Choiz, and K. Inoue, “Proactive clone recommendation system for extract method refactoring,” In Proceedings of International Workshop on Refactoring, pp.67–70, 2019.

[11] M. Mondal, C.K. Roy, and K.A. Schneider, “An exploratory study on change suggestions for methods using clone detection,” In Proceedings of International Conference on Computer Science and Software Engineering, pp.85–95, 2016.

[12] A. Monden, S. Okahara, Y. Manabe, and K. Matsumoto, “Guilty or not guilty: Using clone metrics to determine open source licensing violations,” IEEE software, vol.28, no.2, pp.42–47, 2010.

[13] M.A. Oumaziz, J.-R. Falleri, X. Blanc, T.F. Bissyandé, and J. Klein, “Handling duplicates in dockerfiles families: Learning from experts,” In Proceedings of International Conference on Software Maintenance and Evolution, pp.524–535, 2019.

[14] J. Henkel, C. Bird, S.K. Lahiri, and T. Reps, “Learning from, understanding, and supporting devops artifacts for docker,” In Proceedings of International Conference on Software Engineering, pp.38–49, 2020.

[15] S. McIntosh, M. Poehlmann, E. Juergens, A. Mockus, B. Adams, A.E. Hassan, B. Haupt, and C. Wagner, “Collecting and leveraging a benchmark of build system clones to aid in quality assessments,” In Proceedings of International Conference on Software Engineering, pp.145–154, 2014.

[16] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit, “Can clone detection support quality assessments of requirements specifications?,” In Proceedings of International Conference on Software Engineering, pp.79–88, 2010.

[17] C. Domann, E. Juergens, and J. Streit, “The curse of copy&paste—cloning in requirements specifications,” In Proceedings of International Symposium on Empirical Software Engineering and Measurement, pp.443–446, 2009.

[18] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, “Index-based code clone detection: incremental, distributed, scalable,” In Proceedings of International Conference on Software Maintenance, pp.1–9, 2010.

[19] U. Manber and G. Myers, “Suffix arrays: a new method for on-line string searches,” Siam Journal on Computing, vol.22, no.5, pp.935–948, 1993.

[20] G. Nong, S. Zhang, and W.H. Chan, “Two efficient algorithms for linear time suffix array construction,” IEEE Transactions on Computers, vol.60, no.10, pp.1471–1484, 2010.

[21] H.A. Basit and S. Jarzabek, “Efficient token based clone detection with flexible tokenization,” In Proceedings of European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp.513–516, 2007.

[22] 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.