Dockerfileプリプロセッサの自動抽出に向けたファイルアーキテクチャの調査

馬渕 航　柏 祐太郎　柗本 真佑　飯田 元　楠本 真二

Docker におけるコンテナ開発では，コンテナで利用する OS や提供するサービスのバージョンなど，様々な環境や利用形態に対応したコンテナが提供されている．構築するコンテナの要素が異なると，Dockerfile 内に記述すべき内容も異なる．多くのコンテナ開発プロジェクトでは，Dockerfile プリプロセッサ（DPP）を各自で開発し，Dockerfile の差異を管理している．各プロジェクトで多種多様な DPP が実装される一方，どのような実装が存在し，どの程度 Docker コンテナの管理に役立つかなど，明らかになっていないことが多い．本研究では，DPP のパターン分類および DPP ファイルアーキテクチャの調査を行う．Docker Hub から 60 件のコンテナ開発リポジトリを抽出し，目視分類を行った．その結果，既存研究で明らかにされていない DPP パターンを観察した．また，定量的分析を実現するために，DPP の自動検出方法を検討および評価した．評価実験の結果，Dockerfile テンプレート名による抽出が最も精度が高かった．

1　はじめに

Docker はコンテナ仮想化を実現するプラットフォームである．コンテナ仮想化はホストマシンに依存しない仮想環境の複製が容易であり，高い可搬性を実現する．IT 企業の 79%以上が Docker を利用しているとの報告があり [14]，コンテナ仮想化におけるデファクトスタンダードと言える．また，コード記述により再現可能な仮想環境インフラの構築が可能なことから，ソフトウェアの配布方法やチーム共通の開発環境としても活用されている [3] [11].

Docker におけるコンテナ開発においては，複数の環境や利用形態を想定し，サポートすることが一般的である．例えば，コンテナのベースとなる OS（Ubuntu や CentOS）や，ホストマシンのアーキテクチャ（AMD364 や i386），提供するサービスのバージョン（1.0 や 2.0）など，様々な利用形態の組み合わせが存在する [12]．各要素が異なれば，Dockerfile（コンテナ構築手順が記載されたソースコード）内の記述内容も異なる．例えば，Linux ディストリビューションにおいては，CentOS では `yum`，Alpine では `apk` をパッケージマネージャとして用いるため，利用形態に応じて Dockerfile を書き換える必要性がある．

利用形態で異なる Dockerfile を管理するために，Dockerfile のプリプロセッサ（DPP）が広く採用されている．DPP では，生成対象となる Dockerfile の枠組みにベース OS やバージョン等の可変な情報を入力として埋め込むことで，複数種類の Docker コンテナを自動生成することができる．DPP の利用により，様々な利用形態に対応する Docker コンテナの管理コスト削減が期待できる．

近年では，DPP の普及を背景に，DPP を分析対象とした研究も実施されている．Oumaziz ら [12] は，重複コードを含む Dockerfile を持つリポジトリのうち，52%のリポジトリで DPP が利用され，DPP により重複コードが削減されていることを確認した．また，DPP の実装方法の分類を行い，DPP の実現に複数の実装方法が存在することを示している．

An Empirical Investigation on File Architecture of Dockerfile Preprocessors

Wataru Mabuchi, Yutaro Kashiwa, Hajimu Iida, 奈良先端科学技術大学院大学, Nara Institute of Science and Technology.

Shinsuke Matsumoto, Shinji Kusumoto, 大阪大学, Osaka University.

```
1 FROM alpine
2 ..
3 ENV PYTHON_VERSION 3.11.1
4 RUN ..
5     apk add --no-cache --virtual .
          build-deps
6     ..
7     wget python.tar.xz "https://../
          $PYTHON_VERSION.tar.xz";
8 ..
9 CMD ["python3"]
```

図 1　ベース OS を Alpine，Python3.11 の実行環境コンテナを想定した Dockerfile[†1]

既存研究により DPP の普及状況や実装方法は明らかになってはいるものの，依然として DPP 導入が Docker コンテナの管理にどの程度役立つか等，定量的な観点において明らかになっていないことが多い．定量的な分析が未実施の理由として，DPP を所有するリポジトリの抽出が容易ではないことが考えられる．例えば，先行研究 [12] では，目視作業の負荷を軽減する目的として，複数の Dockerfile を保有するリポジトリに対象を絞っている．この方法を応用することで，間接的に DPP を特定し，定量分析に繋げることも可能と思われる．しかし，実際の Docker コンテナ開発では，`docker buildx` コマンド等を用いることで，複数の Dockerfile を生成することなくコンテナを生成することも多い．したがって，複数の Dockerfile を保有するリポジトリのみを分析対象とすると，複数の Dockerfile を生成しない DPP を検出できない可能性が高い．DPP を確実かつ漏れなく検出するためには，生成物（つまり，複数の Dockerfile）に着目するのではなく，DPP を構成する要素（DPP ファイルや入力ファイル等）から検出する必要がある．

本研究では，分析対象を「複数の Dockerfile を保有するリポジトリ」に限定することなく分析することで新たな DPP パターンが存在するかを確認する．その後，DPP を構成するファイルアーキテクチャを明らかにし，DPP を構成するファイル名から DPP を自動で検出することを目指す．

†1 `https://github.com/docker-library/python/blob/master/3.11/alpine3.16/Dockerfile`

2　背景

本章では，Dockerfile および Dockerfile プリプロセッサ（DPP）を紹介する．その後，Oumaziz らの DPP 分類 [12] およびその問題点について紹介する．

2.1　Dockerfile

Dockerfile とは，コンテナ生成手順が記載されたソースコードである．同一ファイル内に，Dockerfile 固有の構文と RUN 命令に内包される Shell Script 構文 [6] を用いて，ベース OS やコンテナ内で実行する命令を記述する．図 1 に Python 実行環境を提供するコンテナを生成する際の Dockerfile の例を示す．図 1 では，まず FROM 命令でベース OS となる Alpine を選択している．そして，RUN 命令を用いて，パッケージのインストールや http リクエストによるファイルの取得など，コンテナ内で実行するコマンド列を記述している．最後に，コンテナ起動時に実行するコマンドを CMD 命令を用いて記述している．以上のように，Dockerfile を作成し配布することで，誰がいつ実行しても Python 実行環境をサービスとするコンテナを再現可能である [7].

2.2　複数環境対応のための Dockerfile

Dockerfile の作成や配布においては，複数の利用形態を想定しサポートすることが一般的である．利用形態の例としては，コンテナのベースとなる OS やサービスのバージョン等，様々な組み合わせが考えられる．これらの組み合わせによって Dockerfile で記述する内容は変化する．図 1 で示したのは，ベース OS が Alpine，Python のバージョンが 3.11 の場合の Dockerfile である．ここで，ベース OS を Buster（Debian 系 OS），Python のバージョンを 3.7 にした場合の Dockerfile の例を図 2 に示す．3 行目では ENV で指定する変数の値が異なっている．また，5 行目ではパッケージマネージャが `apk` から `apt-get` に変わっていることが読み取れる．配布者は以上のような記述内容の違いを反映させ，各種組み合わせに対応した Dockerfile を用意する必要がある．

```
FROM buster
..
ENV PYTHON_VERSION 3.7.16
RUN ..
    apt-get update;
    apt-get install -y --no-install-
        recommends
    patchelf;
    wget python.tar.xz "https://../
        $PYTHON_VERSION.tar.xz";
..
CMD ["python3"]
```

図 2　ベース OS を Buster，Python3.7 の実行環境コンテナを想定した Dockerfile†2

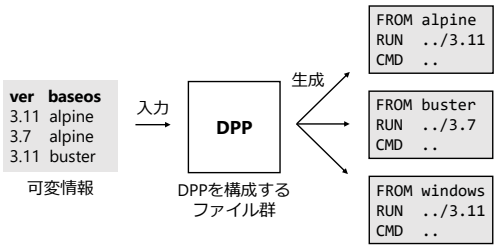

図 3　DPP の概要

2.3　Dockerfile プリプロセッサ（DPP）

複数の利用形態に応じた Dockerfile の作成にあたっては，Dockerfile プリプロセッサ（DPP）が利用されることがある．開発者が DPP に対してバージョンの数値やベースとなる OS を指定する．DPP は入力された可変情報が示す利用形態に対応した Dockerfile を自動生成する．Python コンテナの Dockerfile を生成する DPP の概略を図 3 に示す．図 3 では，Alpine 等のベース OS や 3.11 等のバージョン情報等の可変情報を入力としている．DPP は入力情報を基に処理を行い，出力として複数の Dockerfile を生成する．以上のような仕組みを採用することで，開発者は複数の環境に対応した Dockerfile を一元に管理でき，手間の軽減やバグの低減に繋がる．

```
sed -i \
  -e 's,%%ALPINE_VERSION%%,'"
      $alpinever"',' \
  -e 's,%%DEBIAN_VERSION%%,'"
      $debianver"',' \
  ..
```

図 4　Template processor での Dockerfile 生成の例 †3

2.4　Oumaziz らの DPP 分類

DPP は複数 Dockerfile の管理において有効であるが，DPP 実現のための統一的な方法や Docker 公式の示すスタンダードは存在せず，DPP の実現方法はプロジェクトによって異なる．多種多様な DPP が存在する状況下で，開発者は手探りでの DPP 開発が求められる．Oumaziz ら [12] は，Dockerfile 開発者の 56%が DPP の実現は難しいと感じていると報告している．また，Dockerfile を複数持つリポジトリに焦点を当て，Dockerfile の重複コードの存在を発見し，その管理方法として DPP を調査している．DPP の実現手法として Template processor，Find and replace，Generator の 3 種類のパターンが示されている．以下では，それぞれパターンを紹介する．

Template processor:　Template processor は，Dockerfile のテンプレート内の変数を可変情報に応じて置換しながら Dockerfile を生成する方法である．テンプレートと生成スクリプト 2 種のファイルを持ち，生成スクリプトからテンプレートの変数を “sed” 等の命令で置換する．テンプレート置換の例として，nginx コンテナ管理プロジェクトの例を図 4 に示す．テンプレート内の変数（%%ALPINE_VERSION%%等）を，生成スクリプト内の可変情報データ（$alpinever 等）に “sed” 命令を用いて置換しながら Dockerfile を生成している．

Find and replace:　Find and replace では，Template processor の単純な置換と同様に “sed” 等による置換を用いて Dockerfile を生成する．しかし，テンプレートとして用いるファイルが異なる．Template processor では予め用意されたテンプレー

†2 https://github.com/docker-library/python/blob/master/3.7/buster/Dockerfile

†3 https://github.com/nginxinc/docker-nginx/blob/d039609e/update.sh

```
print_ubuntu_java_install() {
  ..
  cat >> $1 <<'EOI'
RUN set -eux; \
    ARCH="$(dpkg␣--print-architecture)
        "; \
    case "${ARCH}" in \
EOI
  print_java_install ${file} ${srcpkg}
      ${dstpkg};
}
```

図 5　Generator での Dockerfile 生成の宣言部分 †4

```
if [ "${os}" == "ubuntu" ]; then
    print_ubuntu_java_install ${file}
        ${srcpkg} ${dstpkg};
elif [ "${os}" == "alpine" ]; then
    print_alpine_java_install ${file}
        ${srcpkg} ${dstpkg};
elif..
```

図 6　Generator での Dockerfile 生成の呼出部分 †5

```
build:
  docker build $(DOCKER_BUILD_ARGS) --
      build-arg OWNER=$(OWNER)
push:
  docker push --all-tags $(OWNER)/$(
      notdir $@)
```

図 7　docker build を用いた可変情報対応の例 †6

トを用いたが，Find and replace では以前に作られた Dockerfile を用いる．既存の Dockerfile から，正規表現を用いることで置換箇所を探し出し，新しく変更される情報に置換する．

Generator:　Generator は，生成スクリプト内で Dockerfile の生成メソッドを宣言し，そのメソッドの組み合わせによって Dockerfile を生成する方法である．Dockerfile の全文を，FROM 文やパッケージマネージャ関連のコマンドのような可変情報に伴って変わる記述で分けて，それぞれの生成メソッドを宣言する．そして，可変情報に合わせたメソッドの使い分けや，メソッド内での変数処理を用いて，Dockerfile を生成する．ibmjava プロジェクトでの生成スクリプトの例を図 5 と図 6 に示す．図 5 はメソッドの宣言部分である．メソッド内では，ベース OS によって記述が異なるパッケージマネージャの命令を “cat” で Dockerfile に書き込む．図 6 はメソッドの呼び出し部分である．ベース OS に合わせて実行するメソッドを区別して呼び出す．以上のようにして，白紙のファイルに可変情報に応じた文を “cat” 命令によって書き込み，Dockerfile を生成する．

2.5　Oumaziz らの分類の問題点

上述の Oumaziz ら [12] の分類は，複数の Dockerfile を含むリポジトリのみを対象にしていた．しかし，Dockerfile を 1 つしか持っていないリポジトリにも，複数の利用形態に応じた Docker コンテナの生成を自動化した例がある．jupyter プロジェクトのコンテナ生成を図 7 に示す．図に示すように，docker build 命令を活用して，複数のベース OS やバージョンに対応する例が見られる．このような手法は，docker buildx などのコンテナビルドツールの進化で実現しやすい傾向にある．docker build 命令を用いた手法はベースとなる Dockerfile 一つで複数のコンテナを生成することができる．このようなケースは，Oumaziz らの研究では調査対象に含まれていない．DPP の抽出において，docker build を利用した手法の特徴も調査し，抽出のための情報を収集する必要がある．

3　Research Questions

本研究では，Oumaziz ら [12] の DPP 分類の補完および DPP を構成するファイルの把握と，特徴の整理を行い，最後に DPP の自動抽出評価を行う．具体的には，以下の 3 つの Research Questions に取組む．

RQ1: DPP の実現方法にどのようなパターンが存在するか？

Oumaziz ら [12] の研究では，複数の Dockerfile が存在するリポジトリを分析対象として DPP を観察していた．しかし，多種のバージョンやベース OS

†4 https://github.com/ibmruntimes/ci.docker/blob/master/ibmjava/update.sh

†5 https://github.com/ibmruntimes/ci.docker/blob/master/ibmjava/update.sh

†6 https://github.com/jupyter/docker-stacks/blob/main/Makefile

をサポートするコンテナ生成は 1 つの Dockerfile からでも可能である．RQ1 では，Oumaziz らの研究の *Conceptual Replication* [8] として，Docker Hub に存在するプロジェクトから，「複数の Dockerfile が存在するリポジトリ」の制限をせずに DPP を目視で分類することで，異なるパターンの DPP 検出を試みる．

RQ2: DPP の特定に必要な情報は何か？

本研究の目的である DPP の自動抽出を行うには，DPP のファイル名や DPP の記載内容から特定する必要がある．Mazinania ら [10] は，CSS のプリプロセッサを特定するために “*.less” などの拡張子の種類を用いた抽出を実施している．しかしながら，DPP のファイル名や記載内容は自由度が高く，容易に抽出することが難しいと考えられる．そのため，DPP だけでなく DPP の関連ファイル（入力）を含めた DPP 全体のアーキテクチャから特定することが効率的と考えられる．RQ2 では，DPP を用いて Docker コンテナを生成する際に利用されるファイルおよび DPP を含むファイル間の関係性を整理する．

RQ3: DPP をどの程度の精度で自動抽出できるか？

RQ3 では，本研究の目的である DPP の自動抽出を行う．具体的には，RQ2 において特定したファイル構成から頻出するファイル名に対して正規表現を用いて抽出し，どの程度の DPP が特定できるかを評価する．DPP が自動抽出できることで，DPP を利用するプロジェクトと利用しないプロジェクトの比較など定量的な分析の実現が期待できる．

3.1 データ収集方法

本研究では，DPP を保有するリポジトリを収集するために，まず Docker Hub から Docker コンテナ開発プロジェクトを特定する．具体的には， Docker Hub 内における Docker official image, Verified Publisher，Sponsored OSS の 3 種類の公認マークのついたコンテナからそれぞれダウンロード数上位 20 件ずつ，計 60 プロジェクトを対象とする．これらは高品質なコンテナプロジェクトと Docker から評価されている †7 †8 †9.

次に，GitHub の URL を Docker Hub から取得する．ほとんどコンテナは，Docker Hub 内のコンテナ概要欄ページに Dockerfile のソースコードを示す GitHub の URL を公開している．調査対象のコンテナ概要欄から URL を探し，取得する．ただし，URL を取得できなかったコンテナは調査対象から外し，新たなコンテナを調査対象に追加する．また，一部のコンテナでは，管理者が同じ複数のコンテナが存在する．この場合，管理方法も同様であるため，2 件目以降は調査対象から外し，新たなコンテナを追加する．

3.2 分析方法

次に，対象となった各リポジトリに含まれるファイルのそれぞれを目視し，DPP を構成するファイルが含まれるかを確認する．DPP を構成するファイルの判断にはファイル名や記述内容を用いた．DPP を構成するファイル内では，複数のベース OS やバージョンの数字を定義する記述や，“Dockerfile” の文字列を含む記述が含まれる．また，そのファイルと入出力の関係にあたるファイルも記述から読み取れる．以上の判断基準から目視による DPP 特定を実施した．その結果，53 件のリポジトリで DPP を特定した．これらの DPP を対象に，それぞれの RQ に従って調査を実施する．

RQ1 では，収集した DPP を Oumaziz ら [12] を参考に分類を行う．Oumaziz らの定義した DPP パターンにあてはまらない DPP が存在した場合は，新たなカテゴリを追加する．最後に，各 DPP パターンの採用数および割合を集計する．

RQ2 では，リポジトリ内に存在する DPP 構成ファイルの特徴を明らかにする．まず，DPP 構成ファイルを精査し，高頻度で利用されるファイル名や複数のファイルに共通して現れる命令を調査する．そして，DPP を構成するファイルの役割を整理し，DPP の

†7 https://docs.docker.com/docker-hub/official_images/

†8 https://docs.docker.com/docker-hub/dvp-program/

†9 https://docs.docker.com/docker-hub/dsos-program/

表 1　公認マーク毎の DPP パターン採用の分布

	Docker official	Verified publisher	Sponsored OSS	Total
Template processor	8	3	3	14 (26%)
Find and replace	8	4	3	15 (28%)
Generator	1	0	0	1 (1.9%)
Container build	0	10	13	23 (43%)

ファイルアーキテクチャを明らかにする．

RQ3 では，RQ2 で特定した DPP のファイルアーキテクチャパターンを利用し，各プロジェクトの Git リポジトリから DPP の特定を行い，DPP を用いているリポジトリをどの程度抽出できるかを評価する．精度の指標には，適合率（DPP を含むリポジトリとして抽出したものの内，実際に DPP を含むリポジトリであった割合）および再現率（データセット内の DPP を含むリポジトリの内，抽出できた割合），F1 値（適合率と再現率の調和平均）を利用する．また，比較対象として，Oumaziz ら [12] の DPP 抽出方法である「複数の Dockerfile が存在する」リポジトリを抽出する手法と比較する．

4　調査結果

RQ1: DPP の実現方法にどのようなパターンが存在するか？

表 1 に Oumaziz ら [12] の分類に従って発見した DPP の内訳を示す．DPP 採用プロジェクト全 53 件の内，Template Processor が 26%（53 件中 14 件），Find and replace が 28%（53 件中 15 件），Generator が 1.9%（53 件中 1 件）存在した．表 1 に示すように，Oumaziz ら [12] も調査を実施している Docker official では，彼らの分析と同様の結果が得られた．一方で，Verified publisher や Sponsored OSS のプロジェクトでは，Oumaziz ら [12] の分類には存在しない，複数の Dockerfile を生成しない DPP を発見した．本研究では “Container build” パターンの DPP と呼ぶ．Container build は全 DPP の 53 件中 23 件であった．これは全体の 43%であり，全 4 パターンのうち最も大きな割合を占めている．“Container build” 型の DPP は，コンテナのビルド命令を通して複数の利用形態に対応する方法である．可変情報が変数となっている Dockerfile を使い，docker build 等の命令でコンテナビルドを行う．コンテナのビルド方法には種類があり，2 章で紹介したような docker build 命令の “–build-arg” オプションを用いる方法や buildx 等の拡張ツールを用いる方法がある．newrelic/infrastructure-bundle プロジェクトでの例を図 8 に示す．buildx を用いることで，3 行目から 6 行目のような従来の docker build でも見られるバージョン指定に加え，2 行目のように変数 “${DOCKER_PLATFORMS}” を通した CPU アーキテクチャ指定 (AMD364 や i386) が可能となる．

```
1 docker buildx build
2   --platform="${DOCKER_PLATFORMS}"
3   --build-arg agent_version=
4       "$AGENT_VERSION"
5   --build-arg jre_version=
6       "$JRE_VERSION"
7   -t "${DOCKER_IMAGE}:${
        DOCKER_IMAGE_TAG}"
```

図 8　buildx を用いた可変情報対応の例 †10

RQ1: Oumaziz らの分類に加えて，複数の Dockerfile を生成しない DPP が 43%のプロジェクトで利用されていた．

RQ2: DPP の特定に必要な情報は何か？

DPP はプロジェクトや 4 種類のパターン（Template processor, Find and replace, Generator, Container build）によって，入力として与えるファイルの種類や入力方法が異なる．DPP を特定するための情報として，各 DPP のファイル構成，ファイル名，ファイル内の記述に共通点が見られた．各パターン毎にそれらの共通点を述べる．

Template processor の全体像を図 9 に示す．入力

†10 https://github.com/newrelic/infrastructure-bundle/blob/master/docker-build.sh

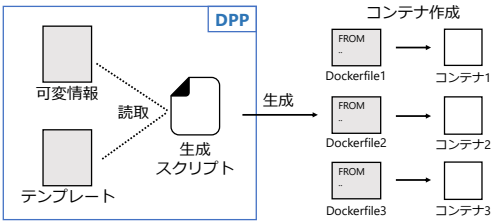

図 9　Template processor の全体像

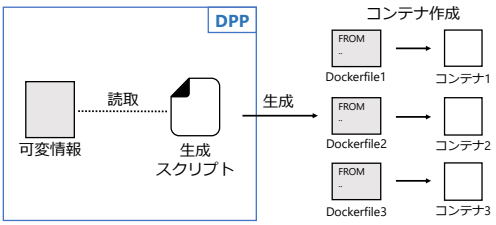

図 11　Generator の全体像

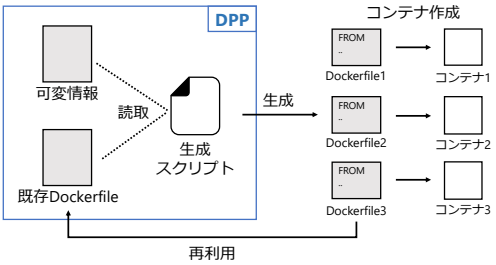

図 10　Find and replace の全体像

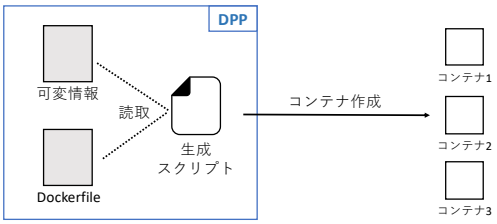

図 12　Container build の全体像

として可変情報とテンプレートを用い，“sed” 等の命令によるテンプレート置換で Dockerfile を生成する生成スクリプトが存在する．テンプレートのファイル名は “Dockerfile.*” が最も多く，収集したテンプレートの 36% （14 件中 5 件）がこのフォーマットであった．次に，“Dockerfile-BASEOS.template” が多く，収集したテンプレートの 29% （14 件中 4 件）がこのフォーマットであった．生成スクリプトは “update.*”(*は任意文字列) が多く，発見した生成スクリプトの 50% （14 件中 7 件）がこのファイル名を利用していた．残りの生成スクリプトは，“render-template.*” や “generate.*” などが数件ずつ見られた．ファイルの記述内容として，生成スクリプト内の置換に用いる命令に分布が見られた．複数のプロジェクトで利用されていた命令としては，“sed” 命令による置換が 43%（14 件中 6 件），“jinja” による置換が 14%（14 件中 2 件）であった．可変情報の形式は 2 種類存在し，JSON 等のファイルで定義される場合と，生成スクリプトの変数や実行時の引数で定義される場合がある．前者の形式は 5 件存在し，“version.json” のようなファイルが見られた．

Find and replace の全体像を図 10 に示す．Template processor と同様に置換のための生成スクリプトを持つ．しかし，入力テンプレートは以前に作成された Dockerfile を用いるため，テンプレートのみを目的としたファイルは存在しない．従って，テンプレートとして扱われるファイルは全て，“Dockerfile” あるいは “*.Dockerfile” であった．生成スクリプトのファイル名は Template processor と同様 “update.*” が最も多く，発見した生成スクリプトの 33% （15 件中 5 件）がこのファイル名を使用していた．残りは固有のファイル名を持つ生成スクリプトが多く，“common.sh” や “build.sh”，“update_version.sh” などが見られた．Find and replace でも生成スクリプト内の置換に用いる命令に分布が見られた．複数のプロジェクトで利用されていた命令としては，“sed” 命令による置換が 47%（15 件中 7 件），その他は Go 言語等の各採用言語による命令であった．Find and replace でも Template processor と同様に．可変情報の形式は 2 種類存在した．可変情報だけを定義するファイルが存在するプロジェクトは 3 件であった．

Generator を構成するファイルの全体像を図 11 に示す．入力は可変情報のみであり，白紙ファイルを作成し Dockerfile の内容を書き込む生成スクリプトが存在する．Dockerfile のひな形となる要素は生成スクリプト内に，“cat” 等で書き込むテキストとして存在する．Generator は 1 件しか存在しないため，ファイ

ル名やファイルの記述内容の分布を調査できるほどの十分なデータは無かった．

Container build を構成するファイルの全体像を図 12 に示す．コンテナのビルドに用いる Dockerfile と生成スクリプトを持つ．Dockerfile は “Dockerfile” あるいは “*.Dockerfile” といった名前で存在する．生成スクリプトのファイル名は採用言語として Make の利用が多いことから，“Makefile” が多く，Container build の生成スクリプトの 43% （23 件中 10 件）であった．次に “build” を含むファイル名（“build.sh” や “docker-build.sh” 等）が多く，Container buiild の生成スクリプトの 22%（23 件中 5 件）がこのファイル名であった．Container build でも，可変情報の形式は 2 種類存在した．可変情報の定義ファイルが存在するプロジェクトは 7 件存在し，コンテナビルドのための条件定義ファイル（“docker-compose.yml” や “docker-bake.hcl”）がよく見られた．

RQ2: DPP のパターンによって構成ファイルは異なり，パターンに応じたファイルの特定を要する．また，ファイル名や記述内容には共通点が存在する．

RQ3: DPP をどの程度の精度で自動抽出できるか？

DPP の各パターンのうち最も頻出した生成スクリプトとテンプレートのファイル名を用いて以下の正規表現を用意した．

Regex1: `update\..+`

Regex2: `.*build\..+`

Regex3: `Makefile`

Regex4: `(.+\.[Dd]ockerfile)|([Dd]ockerfile[-.].+)`

各検索表現において，データセットに含まれる 60 件のリポジトリから DPP を検出する精度を評価した．その結果を表 2 に示す．

まず，Template processor と Find and replace の生成スクリプトのファイル名として多く見られた `update\..+`（Regex1）での検索について述べる．分析対象のリポジトリ中 20 件のリポジトリで正規表現に一致するファイルを検出した．20 件のリポジトリのうち，12 件は実際に DPP を利用しており，適合率は 0.60 （=12/20）であった．DPP を利用している全リポジトリ 53 件のうち 12 件の DPP を特定しているため，再現率は 0.23（=12/53）であった．

表 2　DPP を利用するリポジトリの抽出精度

Regex	適合率	再現率	F1 値
`update\..+`	0.60	0.23	0.33
`.*build\..+`	0.26	0.13	0.17
`Makefile`	0.50	0.28	0.36
`(.+\.[Dd]ockerfile)|([Dd]ockerfile[-.].+)`	**0.92**	0.62	0.74
既存手法	0.90	**0.68**	**0.77**

Container build の生成スクリプトのファイル名として見られた.`*build\..+`（Regex2）では，27 件のリポジトリが検索に該当した．その内，DPP の構成ファイルを正しく検出できたリポジトリは 7 件であった．つまり，適合率は 0.26 と再現率は 0.13 で，他の表現と比較し，精度が最も低い結果であった．

3 つ目の `Makefile` での検索（Regex3）では，30 件のリポジトリが抽出された．そのうち，DPP の構成ファイルを正しく検出できたリポジトリは 15 件（適合率: 0.50）であり，全リポジトリの 28%（=再現率）であった．

4 つ目の Dockerfile のテンプレート名での検索（Regex4）では 36 件のリポジトリが得られた．その内，33 件のリポジトリが実際に DPP を利用していた．検出精度では，適合率は 0.92，再現率が 0.62 と他のファイル抽出方法と比較し最も高かった．

最後に，本研究で最も精度の良かったテンプレートのファイル名から抽出する方法（Regex4）と既存手法である Oumaziz らの手法を比較する．Regex4 は，既存手法と比較し，僅差の精度で適合率が高かった（Regex4: 0.92, 既存手法: 0.90）．しかし，再現率で比較すると既存手法の方が Regex4 よりも高い精度であった（Regex4: 0.62, 既存手法: 0.68）．それに伴い，調和平均である F1 値でも，既存手法の方が高い結果となった．ただし，既存手法では，DPP により生成されていない Dockerfile が偶然，複数存在したことにより DPP を所有するリポジトリを検出できた例が散見された （詳細は次節で述べる）．

```
./tools/../Dockerfile
./test/../Dockerfile
```

図 13　異なる機能の **Dockerfile** が抽出される例

RQ3: ファイル名による抽出では，Dockerfile テンプレート名での検索が最も精度が高かった（適合率：0.92，再現率：0.62）．

5　議論

5.1　既存手法と提案手法の検出性能に関する考察

RQ3 では，Dockerfile を含む 3 種類のファイル名の検索による抽出と既存手法で同程度の適合率, 再現率が得られた．しかし，既存手法では偶然複数の Dockerfile が抽出される例が見られた．複数の Dockerfile を含むリポジトリを抽出した際に見られた例を図 13 に示す．図中では，tools ディレクトリと test ディレクトリの中にそれぞれ Dockerfile が含まれている．DPP は，同機能であるが依存パッケージのバージョンやベースとなる OS によって記述が変わる複数の Dockerfile を自動生成するための方法である．しかし，図 13 の 2 種の Dockerfile はそれぞれ tools と test で異なる機能を目的としており，1 つの DPP が生成した複数の Dockerfile ではない．複数の Dockerfile が 1 つの DPP から生成されているかを厳密に評価すると，既存手法の精度は適合率が 63%，再現率が 47%となり，Dockerfile テンプレートを用いる方法よりも大きく下回った．

追加分析：既存手法は，厳密に DPP の構成ファイルを特定できず，同じ DPP から複数生成された Dockerfile が存在するリポジトリを検索した場合，精度は提案手法の検出精度より大きく下回った．

5.2　複数の自動抽出手法の適用

RQ3 では，Dockerfile テンプレートのファイル名に見られる正規表現を用いた方法が最も高い精度を示した．しかし，この方法でも，再現率が 0.62 と改善の余地が大きく残る．本節では，Dockerfile テンプレートを用いる方法に加えて，複数の手法を併用することで再現率を向上させることを検討する．

Dockerfile テンプレートの正規表現による抽出は，Template processor を全て検出できている．しかし，テンプレートを持たない Find and replace や Container build の再現率は 0.27 と 0.61 と低く，これらのパターンの DPP を更に検出する必要がある．

そこで，テンプレート名による方法で抽出しなかったリポジトリに対して，さらに他の正規表現を利用することで精度向上を試みる．例として，Find and replace の生成スクリプトのファイル名で見られた，`update\..+`による抽出を，テンプレート名が抽出しなかったリポジトリに対しておこなった．このとき，新たに 5 件のリポジトリを得た．このうち，3 件が正しく検出された DPP 採用リポジトリであり，残り 2 件のリポジトリが，DPP 以外のファイルを検出したリポジトリであった．テンプレート名による方法と組み合わせると，再現率が 66%，適合率が 88%となる．数種類のパターンの組合わせだけでは，すべての DPP を検出できないものの，テンプレート名による単体抽出での精度と比較し，一定の精度向上は見られた．今後は，様々なパターンを追加していくことで漏れなく DPP を検出できることを目指す．

6　関連研究

6.1　Dockerfile の保守に関する研究

Ye ら [16] は，Docker コンテナにインストールするソフトウェアに応じた Dockerfile の自動作成ツールを提案した．そして，59%の精度で目的の Dockerfile の自動生成に成功した．ただし，バージョン更新等の Dockerfile 生成後のメンテナンス方法については述べられていない．Hassan ら [4] は，Docker コンテナ内のソフトウェアのアップデート頻度や，その影響を調査し，Docker コンテナのアップデートの重要性を示している．また，Dockerfile の更新履歴の調査は活発に行われている [15] [2]．Gholami ら [2] は，Dockerfile の更新を自動検出し，多くのパッケージ変更やアップデートが見られることを示した．Tanaka ら [15] は，Dockerfile を含む 7914 件の GitHub リポジトリを調査し，Dockerfile の更新活動におけるメタメンテナンスの重要性を示した．以上のように，Dockerfile の更新活動の調査を行なっているが，更新方法としての DPP の存在は示していない．

6.2 プリプロセッサに関する研究

プリプロセッサの研究は，C 言語や HTML，CSS などの言語で研究がされている [1] [9] [10]．Ernst ら [1] は，C 言語プリプロセッサの実証的調査を実施し，C プリプロセッサの使用方法やマクロ定義や使用における不整合とエラーに関する報告を行っている．Tatsubori ら [9] は，テンプレートの使用による，HTML や XML ファイルの生成方法を提案している．Mazinanian ら [10] は，150 の CSS プリプロセッサを調査し，プリプロセッサの特徴を 4 分割し，それぞれの使用パターンやメンテナンス時の応用例を示している．その他にも，Parr ら [13] は，Web アプリケーションの作成におけるテンプレートの活用法を調査した．Hata ら [5] は，Python や JavaScript 等の言語が採用された 30,000 以上の GitHub リポジトリを対象に，重複箇所や変数の管理を調査し，メタメンテナンスの重要性を示した．

7 おわりに

本研究では，Dockerfile プリプロセッサ（DPP）の自動抽出を目的として，DPP のパターン分類および DPP ファイルアーキテクチャの調査を行った．まず，先行研究で未調査のリポジトリを含めた上で目視調査を実施し，新しい DPP パターンを発見した．そして各 DPP パターンにおけるファイル名やファイル内容の特徴を調査した上で，ファイル名での DPP の抽出を実施し，既存手法を含め精度を比較した．その結果，Dockerfile テンプレート名を利用した正規表現により，既存手法を上回る精度で DPP を抽出できた．

今後の展望として，更なる DPP 検出精度の向上を目指す．今回ファイル名だけで検出できなかった DPP から，ファイル内の命令における特徴を調査し，検出に利用する．そして，DPP 検出ツールを用いて，DPP を利用する・利用しないプロジェクトに分類し，DPP 導入の目安となる条件や DPP の利用による効果を明らかにする．

謝辞 本研究は，JSPS 科研費（JP21H03416, JP21K17725, JP21H04877, JP20H04166, JP21K18302, JP21K11829）の助成を受けた．

参 考 文 献

[1] Ernst, M. D., Badros, G. J., and Notkin, D.: An Empirical Analysis of C Preprocessor Use, *Trans. Software Engineering*, Vol. 28, No. 12(2002), pp. 1146—1170.

[2] Gholami, S., Khazaei, H., and Bezemer, C.-P.: Should you Upgrade Official Docker Hub Images in Production Environments?, *Proc. of ICSE-NIER*, 2021, pp. 101–105.

[3] Haque, M. U., Iwaya, L. H., and Babar, M. A.: Challenges in Docker Development: A Large-Scale Study Using Stack Overflow, *Proc. of ESEM*, 2020, pp. 1–11.

[4] Hassan, F., Rodriguez, R., and Wang, X.: RUDSEA: Recommending Updates of Dockerfiles via Software Environment Analysis, *Proc. of ASE*, 2018, pp. 796—801.

[5] Hata, H., Kula, R. G., Ishio, T., and Treude, C.: Same File, Different Changes: The Potential of Meta-Maintenance on GitHub, *Proc. of ICSE*, 2021, pp. 773–784.

[6] Henkel, J., Bird, C., Lahiri, S. K., and Reps, T.: A Dataset of Dockerfiles, *Proc. of MSR*, 2020, pp. 528—532.

[7] Jiang, Y. and Adams, B.: Co-evolution of Infrastructure and Source Code - An Empirical Study, *Proc. of MSR*, 2015, pp. 45–55.

[8] Kitchenham, B. A.: The role of replications in empirical software engineering - a word of warning, *Empir. Softw. Eng.*, Vol. 13, No. 2(2008), pp. 219–221.

[9] Kristensen, A.: Template resolution in XML/HTML, *Trans. Computer Networks and ISDN Systems*, Vol. 30, No. 1(1998), pp. 239–249.

[10] Mazinanian, D. and Tsantalis, N.: An Empirical Study on the Use of CSS Preprocessors, *Proc. of SANER*, Vol. 1, 2016, pp. 168–178.

[11] Morris, D., Voutsinas, S., Hambly, N., and Mann, R.: Use of Docker for deployment and testing of astronomy software, *Astronomy and Computing*, Vol. 20(2017), pp. 105–119.

[12] Oumaziz, M. A., Falleri, J.-R., Blanc, X., Bissyandé, T. F., and Klein, J.: Handling Duplicates in Dockerfiles Families: Learning from Experts, *Proc. of ICSME*, 2019, pp. 524–535.

[13] Parr, T. J.: Enforcing Strict Model-View Separation in Template Engines, 2004.

[14] Portworx: Annual Container Adoption Report, 2019. `https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf` (accessed 2023-01-31).

[15] Tanaka, T., Hata, H., Chinthanet, B., Kula, R. G., and Matsumoto, K.: Meta-Maintanance for Dockerfiles: Are We There Yet?, 2023.

[16] Ye, H., Zhou, J., Chen, W., Zhu, J., Wu, G., and Wei, J.: DockerGen: A Knowledge Graph based Approach for Software Containerization, *Proc. of COMPSAC*, 2021, pp. 986–991.