修士学位論文

題目

OverlayFS の解析による Dockerfile のテスト自動生成

指導教員

楠本 真二 教授

報告者

岩瀬 匠

令和 6 年 2 月 1 日

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

令和 5 年度 修士学位論文

OverlayFS の解析による Dockerfile のテスト自動生成

岩瀬 匠

内容梗概

リソースの効率的利用やコスト削減のためにコンテナ化と呼ばれるサーバーの仮想化が広く利用されている．コンテナ化プラットフォームの 1 つである Docker は事実上の標準化コンテナ化プラットフォームであり，多くの企業で利用されている．Docker では Dockerfile と呼ばれるファイルを作成し，それを基に Docker イメージを構築する．その Docker イメージの実行によりコンテナが起動する．Dockerfile の中身は一種のソースコードで，専用の Docker 命令を記述する．この命令の組み合せにより，コンテナの構成情報の設定が可能となる．Dockerfile の記述内容の正しさを確認するためにはテストを行うべきであり，Container Structure Test という Dockerfile を対象としたテストフレームワークが存在する．しかし，開発者がプログラムの動作を網羅するテストを用意するのは困難である．Java や C 言語等の手続き型言語において，テストを自動で生成する技術が多数存在するが，Dockerfile は一般的なソースコードとその性質が大きく異なるため，既存の技術は適用不可である．そのため，本研究では Dockerfile を対象としたテストの自動生成を提案する．提案手法では，Docker イメージで使用されている OverlayFS のレイヤ情報を解析することでテストを生成する．提案手法が開発者の作成したテストと同じ内容のテストを生成できるか評価実験を行った結果，開発者が作成したテストの内半数以上のテストの生成に成功した．

主な用語

Docker，ソフトウェアテスト，テスト自動生成

目次

図目次

1　はじめに

リソースの効率的利用やコスト削減のためにサーバーの仮想化が広く利用されている [1]．コンテナ化は仮想化技術の一種であり，仮想化の主流となっている [2]．コンテナ化によりアプリケーションの優れた移植性と相互運用性，迅速な提供が可能となる [3]．コンテナ化プラットフォームの 1 つである Docker は事実上の標準コンテナ化プラットフォームであり，多くの企業で利用されている [4]．Docker ではまず Dockerfile と呼ばれるファイルに命令文を記述する．次に，作成した Dockerfile のビルドにより，Docker イメージを構築する．最後に Docker イメージの実行により，コンテナが起動する．Dockerfile を作成し，Docker イメージを構築，それに基づいてコンテナを起動する流れの中で，ソフトウェア開発の知見を活用する研究，ツールの開発が多く実施されている [5,6,7]．

Dockerfile のようにソースコードを通してインフラストラクチャを設定する方法は Infrastructure as Code と呼ばれる [8,9]．Dockerfile では専用の Docker 命令を記述し，各命令の実行により Docker イメージが構築される．また，Docker 命令の 1 つ，RUN 命令ではディストリビューションの Shell コマンドが実行可能である．各 Docker 命令の実行結果としては apt や apk 等のパッケージマネージャを用いたインストールや設定ファイルへの追記命令等がある．これら命令の組み合せによりコンテナの構成情報を設定する．これにより再現性のあるコンテナの作成が可能となる [10]．

Dockerfile の記述内容の正しさを確認するために，他のソースコードと同様にテストを行うべきである．Dockerfile を対象としたテストフレームワークとして Container Structure Test が存在する．Container Structure Test では作成したコンテナに対してテストが実行できる．テスト可能な内容は，コンテナ内でコマンドを実行した結果や，コンテナ内に特定のファイルが存在するかの確認，コンテナのメタ情報等がある．テストファイルは yaml や json 形式であり，記法に沿って 1 つずつテストケースを用意する必要がある．しかし，プログラムの挙動全てを網羅するテストを開発者が用意するのは困難である．

Java や C 言語等の手続き型言語ではソースコードから単体テストを自動で生成する技術が多数存在する [11,12]．これらの手法はメソッドに与える引数をランダムに与え実行し，その結果を基にテストを生成する手法や，静的，動的解析の結果を活用する手法や，遺伝的アルゴリズムを活用し，テストケースの網羅率を最大化するようにテストを生成する手法である．これらの技術は，テスト作成における開発者支援を目的としている．同様に Dockerfile に対してもテスト自動生成による開発者支援ができると考える．

本研究では Dockerfile を対象としたテスト自動生成を提案する．既存のテスト生成技術は多く存在するが，Dockerfile に適用できない．これは，Dockerfile は単なる命令の逐次実行であり，分岐やループを内包する一般的なソースコードとはその性質が大幅に異なる [1] ためである．手続き型言語ではプロ

グラムの実行結果が返り値やメモリであるのに対し，Dockerfile の実行結果は Docker イメージというファイルの集合であるという違いも存在する．これらの違いから既存のテスト自動生成手法を適用することはできない．提案手法では Docker イメージが使用している OverlayFileSysytem（OverlayFS）に着目し，Docker イメージ内の各レイヤの集合を解析し，各レイヤから自動でテストを生成する．

提案手法が Github 上にある Dockerfile の開発者が用意したテストと同じ内容のテストを生成できるかの確認のため，評価実験を実施した．その結果，提案手法が半数以上のテスト生成に成功した．また，開発者が用意したテストには無く提案手法のみが生成できたテストも多く生成された．この結果から，提案手法が開発者の用意するような内容のテストの生成が可能であると考えられる．

以降，2 節ではテスト自動生成と Docker 及び Container Structure Test について説明し，3 節でテスト自動生成の関連研究について述べる．4 節では提案手法について説明し，5 節で実施した評価実験について述べる．6 節で実験結果について考察し，7 節で妥当性の脅威について述べる．最後に 8 節で本研究のまとめと今後の課題について述べる．

2 準備

2.1 ソフトウェアテスト

ソフトウェアテスト [13,14] はソフトウェアの正しさを検証する主要な方法である．ソフトウェア開発ではテストに数十億ドルが費やされることもあり，ソフトウェア開発コストの大半を占める [15]．また，ソフトウェアテストのインフラの改善によってこのコストの三分の一を削減できる可能性があると推定されている [16]．通常，ソフトウェア開発で実施されるテストは単体テスト，結合テスト，総合テストの三段階に分けられる．単体テスト [17] はシステムの個々の機能単体で正しく動作するのかを検証するテストである．結合テストは機能を組み合せて正しく動作するのかを検証する．モジュール同士の組み合せや外部のモジュールとの結合を検証する場合も存在する．最後に総合テストはシステム全体の振舞いをテストする．機能要求が満たされているか，仕様通りかを検証する．単体テストの役割は各機能が単体で動作するのか，つまり全ての機能の動作を検証する必要があり，100% の網羅率が必要である．しかし，単体テストは非常に難しく，コストがかかるため適切に実施されることはほとんど無い [16,18]．この問題を解決するためにテストケースを自動的に生成する手法が多く研究されている．

2.2 テスト自動生成

テスト自動生成の手法にはランダムテストを生成する手法 [19] や対象のソースコードを解析して生成する手法 [20]，探索ベースのアルゴリズムを利用する手法 [21] 等がある．

2.2.1 ランダムテスト生成

ランダムテストを生成する手法は，テスト対象のソースコードに含まれるメソッドや関数のテストを生成する．メソッドに与える入力をランダムに生成し，テストケースを生成する．生成したテストケースが実行時に例外を投げるか，といった基準をあらかじめ用意し，テストが通ったか判断する．この手法は入力をランダムに決めるため自動化に適しているが生成したテストの網羅率が低い，という課題点が存在する．この課題を克服するために適応型ランダムテスト [22,23] とフィードバック指向型ランダムテスト [19,24] と呼ばれる手法も研究されている．

2.2.2 ソースコード解析によるテスト生成

ソースコード解析によるテスト生成手法ではシンボリック実行 [25] を活用する．シンボリック実行は，はじめにソースコードを解析し，実行可能なパスを抽出する．次に，それぞれのパスを通るために必要な入力値の条件を収集する．最後にパスごとの条件を制約ソルバを用いて解く．シンボリック実行によって得られた解の値をそれぞれのテストの入力とする手法である．この手法の課題点として制約ソ

```
1  FROM ubuntu
2  RUN apt-get update \ &&
3      apt-get install -y clamav python3-pip jq \ &&
4      freshclam \ &&
5      pip3 install awscli
6  COPY process_file.py /
7  WORKDIR /viruscheck
8  CMD aws s3 sync s3://tdr-upload-files-dev/$CONSIGNMENT_ID . && clamscan > out.
       log || : && python3 /process_file.py
```

図 1: Ubuntu 上で Python，AWS を動作できるよう設定する Dockerfile

ルバで解くことができない場合にはテストが生成されない点が挙げられる．現在ではシンボリック実行を改善した動的シンボリック実行を用いたテスト生成も研究されている [26].

2.2.3 探索ベースのアルゴリズムを利用した生成

探索ベースの生成手法では，あらかじめテストを評価する評価関数を用意する．生成したテストに評価関数を実行し値を取得する．値の良いテストに対し，探索アルゴリズムを用いて新たなテストを生成する．新たに生成されたテストに対し，再び評価関数を実行し値を取得する．これらの手順を評価関数の実行結果がある閾値を超えるまで繰り返し実行し，テストを生成する手法である．

2.3 Docker

Docker はアプリケーションを開発，実行するためのコンテナ化プラットフォームである [4]．Docker ではコンテナによる仮想化を実現し，アプリケーションを開発環境から分離する．これによりアプリケーションの迅速な提供，可搬性の向上，効率的なリソースの利用を可能にする．これらの特徴から Docker は事実上の標準コンテナ化プラットフォームとなり IT 企業の 87% 以上が Docker を利用し，数多くの OSS コミュニティでも採用されている [27, 28].

Docker ではコンテナを稼働するためにまず Dockerfile を作成する．作成した Dockerfile のビルドにより Docker イメージが構築される．Docker イメージは複数のイメージレイヤと呼ばれるレイヤから構成される．イメージレイヤの中身は Dockerfile の各命令の実行結果により生じた変更差分であり，各レイヤにインストールされたバイナリや追加されたファイルの情報が保存される．作成した Docker イメージを実行することでコンテナが起動する．

Dockerfile の例を図 1 に示す．Dockerfile では初めにベースイメージと呼ばれるイメージレイヤの最初の層を設定する必要がある．1 行目の FROM 命令がこれに該当する．2 行目の RUN 命令では対応

するディストリビューションの Shell コマンドを入力することで必要なバイナリのインストールを行っている．また Shell コマンドは “&&” という文字列でつなげることで複数の Shell コマンドをひとつの RUN 命令にまとめ，上から順に実行できる．今回の例では clamav，python3-pip，jq をインストールしたあと，インストールされた freshclam の実行，pip3 による awscli のインストールが実行されている．6 行目の COPY 命令ではローカルファイルである “process_file.py” をコンテナ内で実行できるようにコピーしている．7 行目の WORKDIR 命令ではコンテナ内で作業する際のベースディレクトリを指定している．8 行目の CMD 命令ではコンテナ実行時に自動的に実行するコマンドを指定している．このように，Dockerfile は一種のソースコードであり，中身は手続き的な命令文の列挙である．これらの命令を通してコンテナの構成情報を設定することで再現性のあるコンテナの作成が可能である [10].

次に，Docker イメージについて述べる．Docker イメージでは UnionFileSystem [29] の 1 つである OverlayFS が利用されている．OverlayFS とは複数のファイルやディレクトリの集合を透過的に重ね合わせ，1 つのファイルシステムとみなす仕組みである．Docker イメージでは各 Docker 命令ごとに，実行結果によってインストールされたバイナリや追加されたファイルがイメージレイヤに保存される．各イメージレイヤの内容を透過的に重ね合わせることで，1 つのファイルシステムを構築している．Docker イメージの例を図 2 に示す．この Docker イメージは図 1 によってビルドされた．レイヤ 1 には FROM 命令の実行によって生成されたファイル群が保存される．レイヤ 2 には RUN 命令で実行されたバイナリのインストールや freshclam コマンドの実行結果が保存される．レイヤ 3 には COPY 命令により追加された “process_file.py” が保存され，レイヤ 4 には WORKDIR 命令により作成された viruscheck ディレクトリが保存されている．CMD 命令はコンテナ実行時のコマンドを設定しているだけで，変更差分が生じないためレイヤが生成されない．図 2 では説明のためレイヤ 1，レイヤ 2 の中身は一部のみ抜粋をした形だが，実際には数千ファイルもの膨大な数のファイル差分が保存されている．また，レイヤはホスト OS 上の `/var/lib/docker/overlay2/mcvhqrs8ektas9jc2i6dmwfg2/diff/` に保存される．mcv から始まる文字列はレイヤ ID である．

現在，Dockerfile の作成からコンテナ起動までの一連の流れの中で，ソフトウェア開発の知見を活用する研究，ツールの開発が盛んに実施されている．Dockerfile に関してコード補完や，リファクタリング，データセットの作成等の研究がある [5, 30, 31]．また，作成したイメージのセキュリティに関する調査も多く実施されている [32, 33]．本研究では，Docker におけるテストに着目する．

2.4 Container Structure Test

Container Structure Test は作成したコンテナに対してテストを実行するフレームワークであり，yaml や json 形式で書いたテストファイルの内容を満たしているかどうかテストする．テスト可能な項目を以下に示す．

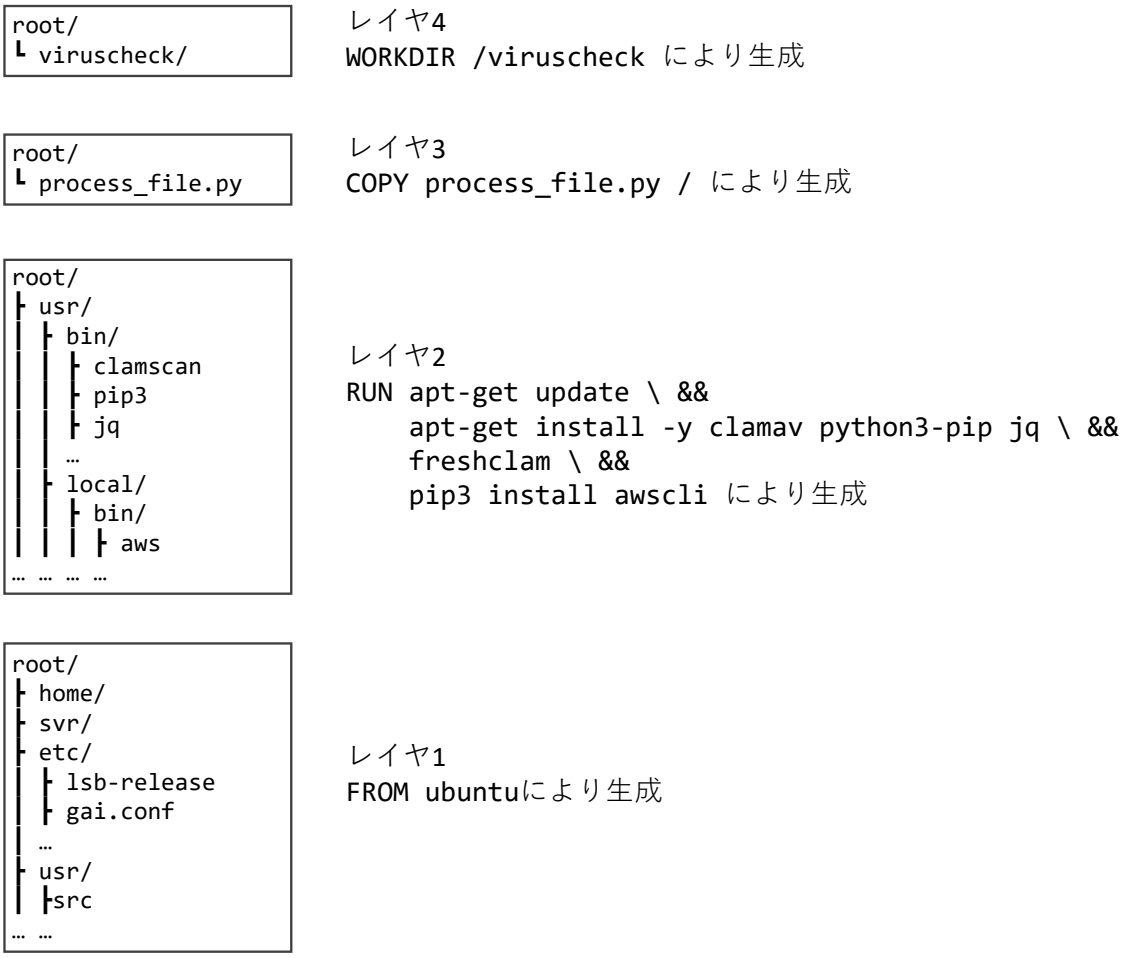

図 2: 図 1 の Dockerfile により構築される Docker イメージ

- commandTests：コンテナ内で Shell コマンドを実行しその出力結果をテストする．
- fileExistenceTests：特定のファイルやフォルダが存在するかをテストする．
- fileContentTests：ファイルの中身をテストする．
- metadataTests：コンテナのメタ情報（ポート番号や WORKDIR）をテストする．
- licenseTests：ライセンスファイルをテストする．
- globalEnvVars：環境変数をテストする．

図 1 の Dockerfile に対して，開発者が用意したテストケースを図 3 に示す．図 3 は 2 つのテストで構成される．1 つ目のテストは clamscan というバイナリがインストールされているかどうかを調べるテストである．テストの内容は which コマンドを clamscan を引数につけて実行し，その実行結果がインストールされている場所である/usr/bin/clamscan を指しているかどうかをテストしている．2 つ目

```
commandTests:
  - name: "clamav installation"
    command: "which"
    args: ["clamscan"]
    expectedOutput: ["/usr/bin/clamscan\n"]
  - name: "aws cli installation"
    command: "aws"
    args: ["--version"]
    exitCode: 0
```

図 3: Container Structure Test のテスト例

のテストは aws cli がインストールされているかどうかを調べるテストである．テストの内容は aws コマンドを–version を引数につけて実行し適切に終了しているかを調べることでインストールされているかをテストしている．これらのテストは図 2 のレイヤの中でレイヤ 2 の内容に対してのみテストが実施されている．つまり RUN 命令の実行による結果しかテストすることができていない．COPY 命令の実行の結果追加されたファイルや WORKDIR 命令の実行の結果作成されたディレクトリが存在するかどうか，本来テストするべきである項目がテストできていない．これは，C 言語や java のテストを作成する際にプログラムの動作を網羅するテストを用意するのが困難と同様に，各 Docker 命令の実行結果がどうなるのかを考え，各命令毎に適切なテストを作成するのは困難なためである．そのため，本研究では Dockerfile から Container Structure Test のテストケースを自動で生成することを考える．

3 関連研究

C 言語や Java 等の手続き型言語を対象としたテスト生成手法が数多く研究されている．本節は既存の研究について述べる．

Csallner ら [34] は Java で作成されたソースコードに対して頑健性を調べるためのツール，JCrasher を開発した．このツールではテスト対象のプログラムのメソッドにランダムなデータを与えテストをする．様々なデータをランダムに与え，実行時の例外をスローさせることでバグを検出しようとする．このように，テスト対象に対しランダムに値を与えテストする手法はランダムテストと呼ばれ，ランダムテストを自動で生成する研究は他にも Pacheco ら [35] による Randoop やそれを改良した Liu ら [36] による Randoop-TSR が存在する．これら 2 つのツールはフィードバック指向型のランダムテスト生成アプローチを採用している．既に生成されたインスタンスやプリミティブ型といった単純なものを再利用することで冗長な入力や不正な入力を避けている．

これらの手法より形式的な手法として，テスト対象のプログラムを静的や動的に解析することでテストの網羅率を上げようとする試みも存在する．Artzi ら [37] はテスト対象のプログラムに対し静的，動的解析を実行しその結果を利用してテストケースを生成する Palus を開発した．Palus ではまずプログラムを動的解析し，オブジェクトの状態とメソッド呼び出しの関係を取得する．次に静的解析により，メソッドと変数の書き込みや読み込み関係を抽出する．これらの情報をランダムテスト生成時に活用し，テストを生成する．

さらに探索ベースのアルゴリズムを活用したテスト生成手法も存在する．Fraser ら [38] は Evosuite という Java プログラムのテストを自動で生成するツールを開発した．Evosuite では生成されたテストに対し，遺伝的アルゴリズムを用いて変異，交叉の繰り返しにより網羅率を最大化するようにテストの選別を繰り返す．また，Sakti ら [39] はテスト生成の際にランダムサーチを取り入れたテスト生成ツール JTeXpert を開発した．

上記のように C 言語や Java を対象にしたテスト自動生成は存在するが，Docker には運用できない．既存手法は，引数に与える入力値に応じた実行経路に注目している．テスト対象の入力値をランダムに決定し，テスト対象を可能なかぎり網羅する．そのため，探索問題と見なすことが可能となり，探索アルゴリズムが適用できる．既存手法は複数の経路を取り得る手続き型言語を対象に，その経路を充足させる問題と置き換えている．しかし，Dockerfile では経路の着目に意味がない．分岐が存在せず，順番に各命令が実行される．実行経路は 1 つだけで直線的であるため，探索にならない．

また，テストは実行時の作用が期待通りかを確認する手法である．そのため，テストを考えるうえでは対象実行時の作用を考える必要がある．手続き型言語の場合，実行時の作用はテスト対象のメソッドや関数を呼び出した時の返り値が該当する．手続き型言語の多くは，メソッドや関数等のプログラムの

一部の作用を返り値として表現している．その結果，既存手法は返り値を作用対象と考える．しかし，Docker は返り値を持たない．図 1 の RUN 命令の作用は必要なバイナリのインストールであり，その作用はイメージレイヤに保存される．また，CMD 命令はコンテナ起動時の実行コマンドの設定であり，コンテナのメタ情報として設定される．

上記のように，C 言語や Java 等の手続き型言語と Docker とはそのソースコードの性質が大きく異なる．よって，既存のテスト生成のアプローチは適用不可であり，Dockerfile 専用の自動テスト生成手法が必要である．

4 提案手法

4.1 手法の概要

提案手法の目的は与えられた Dockerfile から Container Structure Test のテストを生成することである．提案手法では Docker が採用している OverlayFS に着目した．Docker イメージの各レイヤに含まれる情報とそのレイヤを生成した Docker 命令を利用してテストを生成する．

Dockerfile では開発者がベースイメージを指定し，そのうえで必要な環境変数の設定やファイルの追加，バイナリのインストールを Docker 命令により実行する．ベースイメージ指定以降の Docker 命令による作用が開発者の欲しい環境を構築するために必要な要素であり，テストするべき対象と考える．ここで，Docker 命令の作用は以下の 2 つに分けられる．

1. レイヤに変更差分として記録される作用
2. コンテナ自体の情報を設定する作用

前者は RUN 命令によるインストールや COPY 命令によるファイル追加等，後者は ENV 命令による環境変数の設定や PORT 命令によるポート番号の設定が挙げられる．後者の作用に関しては単なる設定であり，Docker 命令の作用の規模としては小さく，前者の作用の方が優先度が高いと考える．よって，提案手法では生成対象を前者のみとする．このような作用では，1 つ 1 つの Docker 命令は複数のファイルを生成，バイナリをインストールすることがほとんどであるため，提案手法では 1 つの Docker 命令から 1 つ以上のテストケースを生成する．

図 4 に提案手法の概要図を示す．提案手法は以下の 3 ステップから構成される．

1. Dockerfile に対する前処理
2. 各命令に対応するレイヤの取得
3. テスト生成

Dockerfile に対する前処理のステップでは，RUN 命令内の “&&” で結合された shell コマンドの分割と全ての Docker 命令のコメントアウト化を行う．各 Docker 命令に対応するレイヤを取得するステップでは，前処理が施された Dockerfile のコメントアウトを上から順に 1 つずつ外しながら Dockerfile をビルドする．これにより，Docker イメージの各レイヤとそれに対応する Docker 命令の取得が可能となり，各 Docker 命令に対応する変更差分の取得が可能となる．最後に各イメージレイヤの変更差分からテストを生成し 1 つのテストファイルにまとめる．以下，各ステップの詳細について述べる．

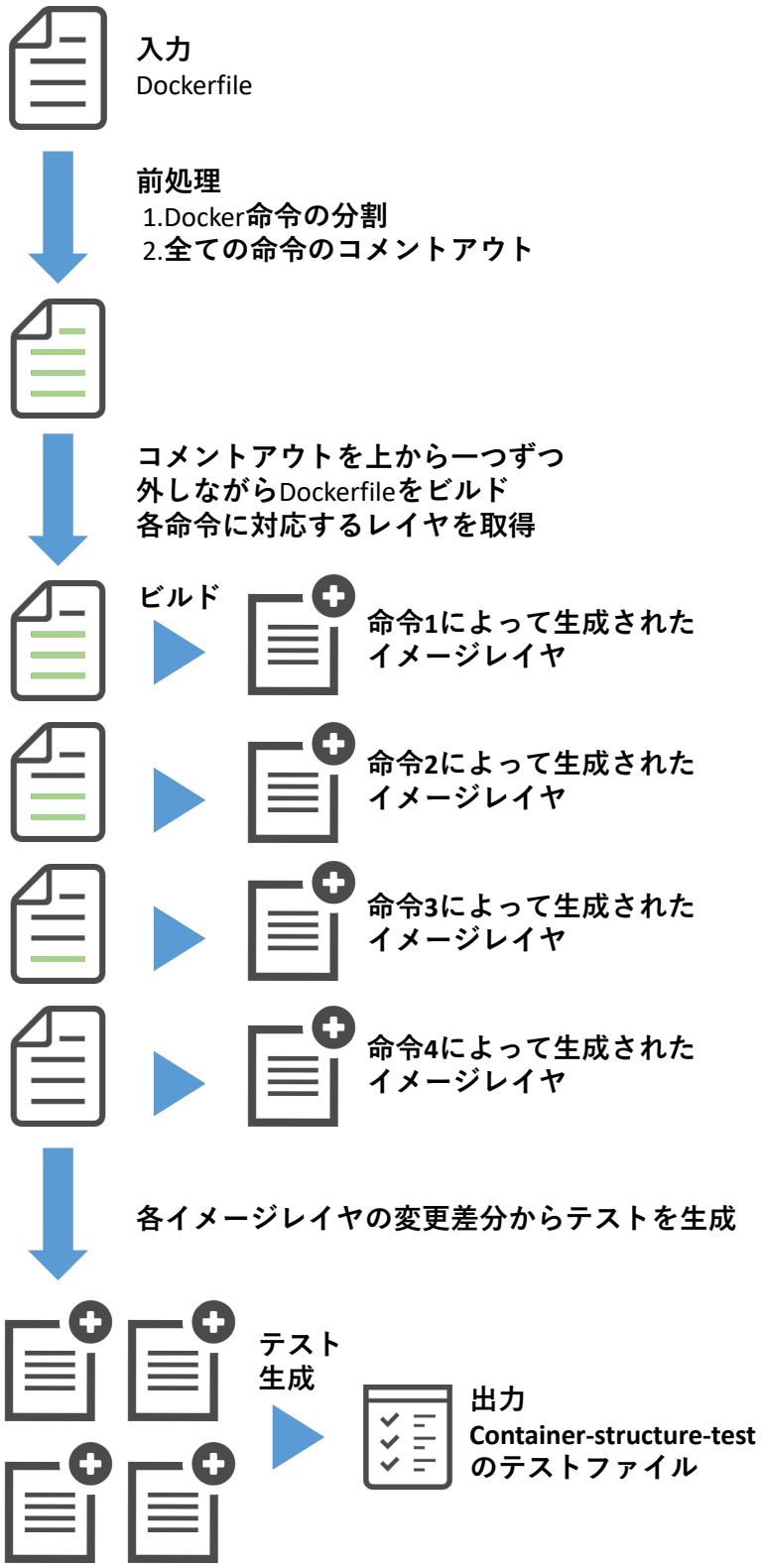
入力
Dockerfile
前処理
1.Docker命令の分割
2.全ての命令のコメントアウト
コメントアウトを上から一つずつ
外しながらDockerfileをビルド
各命令に対応するレイヤを取得
ビルド
命令1によって生成された
イメージレイヤ
命令2によって生成された
イメージレイヤ
命令3によって生成された
イメージレイヤ
命令4によって生成された
イメージレイヤ
各イメージレイヤの変更差分からテストを生成
テスト
生成
出力
Container-structure-test
のテストファイル

図 4: 提案手法の概要

```
1 # FROM ubuntu
2 # RUN apt-get update
3 # RUN apt-get install -y clamav python3-pip jq
4 # RUN freshclam
5 # RUN pip3 install awscli
6 # COPY process_file.py /
7 # WORKDIR /viruscheck
8 # CMD aws s3 sync s3://tdr-upload-files-dev/$CONSIGNMENT_ID . && clamscan > out
      .log || : && python3 /process_file.py
```

図 5: 前処理が施された Dockerfile

4.2 前処理

前処理のステップでは入力の Dockerfile に変更を加える．このステップの目的は 2 つある．Docker イメージの各レイヤに含まれる変更差分を Docker 命令による作用 1 つだけにすることと，各レイヤがどの命令から生成されるかの対応を取れるようにすることである．後者については 4.3 節で詳細を述べるため，本節では前者について説明する．図 5 は図 1 の Dockerfile に前処理を施した結果である．図 1 では “&&” で結合されていた 2 行目から 5 行目が図 5 ではそれぞれ独立の RUN 命令に分割されている．この結果，ビルドされる Docker イメージも変更される．図 6 に図 5 の Dockerfile からビルドされる Docker イメージの一部を示す．これらのレイヤ以外にも COPY 命令によるレイヤ等が Docker イメージには存在するが，図では割愛している．レイヤ 2 からレイヤ 5 までの内容は図 2 では 1 つのレイヤにまとめて保存されていたが，前処理を施すことで各 Docker 命令の作用が各レイヤに保存される．

4.3 Docker 命令とレイヤの対応の取得

2 節で述べたとおり，レイヤの中身は `/var/lib/docker/overlay2/‘‘レイヤ ID’’/diff/` に保存される．これらのディレクトリに find コマンドを実行することで各レイヤの中身，つまり変更差分の取り出しが可能となる．Docker では作成した Docker イメージに対して，各種情報を取得するための docker inspect コマンドが用意されており，各レイヤの ID を取得できる．しかし，docker inspect では各レイヤがどの Docker 命令によって生成されたかまでは分からない．そのため，本ステップでは各レイヤがどの命令から生成されているのかの対応の取得をするために以下の処理を行う．

1. 最も若い行のコメントアウトを外す
2. Dockerfile をビルドし Docker イメージを生成
3. そのイメージに対し docker inspect コマンドを実行

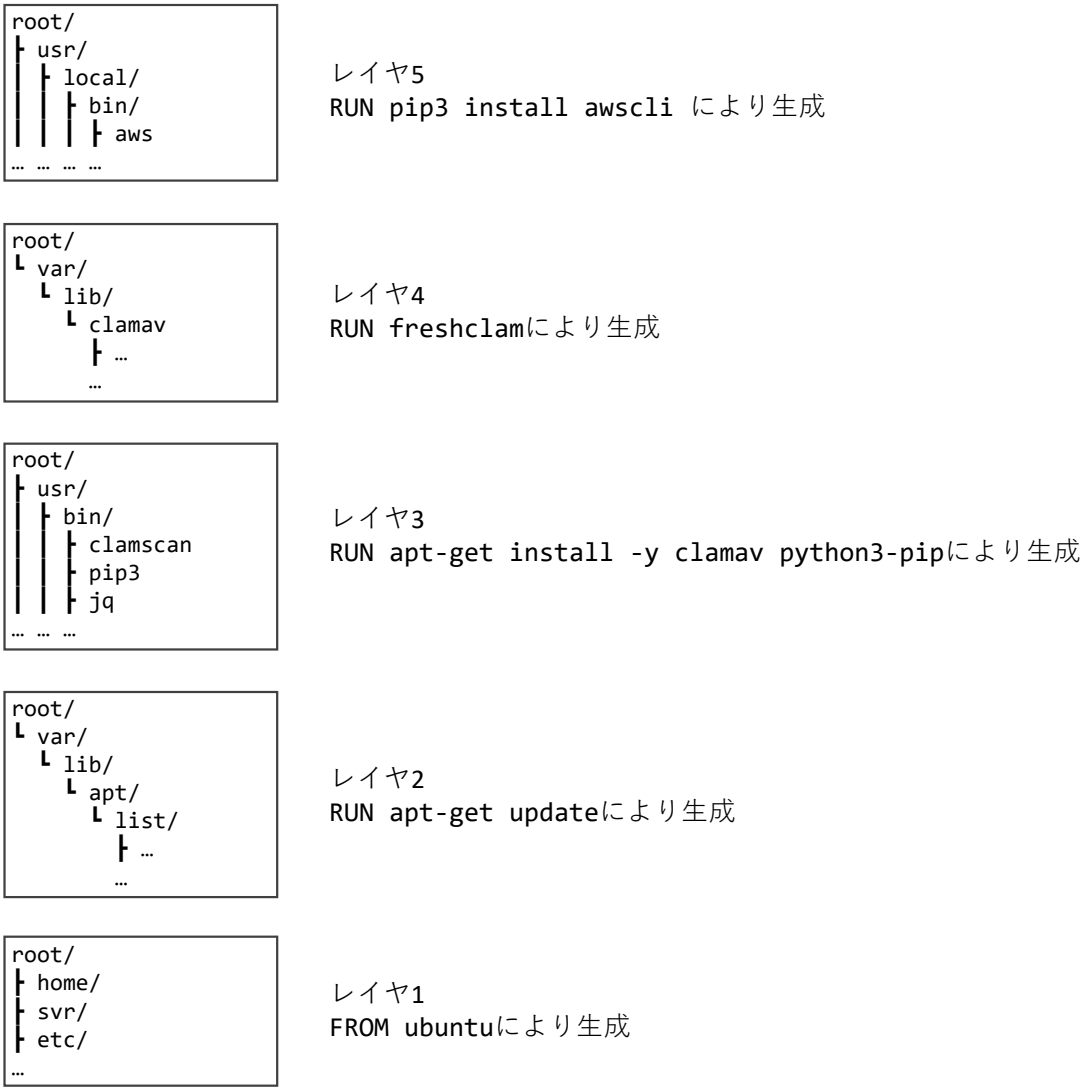

図 6: 図 5 の Docker イメージの一部

4. 新しいレイヤが生成されている場合，コメントアウトが外された Docker 命令がそのレイヤを生成したので対応を取得
5. 1 から 4 をコメントアウトが無くなるまで繰り返す．

図 5 に対しこの処理を実行すると，まず 1 行目のコメントアウトを外し，Docker イメージを生成する．この時，図 6 のレイヤ 1 のみが Docker イメージに存在する．docker inspect コマンドの実行によりこのレイヤ ID を取得し，その ID と FROM ubuntu 命令を対応付ける．次に 2 行目のコメントアウト

を外し，Docker イメージを生成すると，図 6 のレイヤ 1，2 が Docker イメージに存在する．docker inspect コマンドにより新たなレイヤ ID を取得し，RUN apt-get update 命令と対応づける．以下同様に繰り返すことで，各 Docker 命令と各レイヤの対応が取得でき，各 Docker 命令による変更差分の取り出しが可能となる．

4.4 テストの生成

本ステップでは各レイヤの変更差分一覧からテストを生成する．まず，変更差分一覧からテストの対象を選択し，それらのテスト対象全てにテストを生成する．最後に各テストを 1 つのファイルにまとめて出力する．

4.4.1 テスト対象の選別

提案手法では各レイヤの変更差分一覧から 3 つのリストを作成する．それぞれのリストの中身は以下の通りである．

リスト 1　バイナリがインストールされているかつそのファイル名が Docker 命令の中に出現している差分

リスト 2　バイナリがインストールされている差分

リスト 3　Docker 命令の中に出現している単語を含む差分

例えば，`RUN apt-get install -y clamav python3-pip jq` 命令による変更差分に含まれる `usr/bin/pip` や `usr/bin/jq` の差分はバイナリのインストール結果であり，ファイル名の pip や jq は Docker 命令に含まれているため，リスト 1 に入る．リストには優先度があり，リスト 1 が最も高く，リスト 3 が最も低い．要素を持つリストの内，最も優先度の高いリストに含まれる要素全てがテスト対象となる．全てのリストが空の場合，そのレイヤからはテストは生成されない．

4.4.2 テスト生成

既存の Github 上にある Container Structure Test ではバイナリがインストールされているかどうかテストする方法として，ほとんどが commandTests を利用し，which コマンドでテストしている．また，バイナリ以外のファイルが存在するかどうかは fileExistenceTests でテストしている．そのため，提案手法でも，テスト対象がバイナリをインストールしている差分の場合は which コマンドによるテスト，ファイル差分の場合は fileExistenceTests を生成する．図 7 に提案手法が生成する各テストを示す．commandTests ではテストの名前，使用する shell コマンド，コマンドの引数，期待される出力を生成する．提案手法ではテストの名前にどの Docker 命令から生成されたか，使用するコマンドに which，

```
commandTests:
  - name:"generated from RUN apt-get install -y clamav python3-pip jq"
    command:"which"
    args: ["jq"]
    expectedOutput: ["usr/bin/jq"]
fileExistenceTests:
  - name: "generated from COPY process_file.py "
    path: "process_file.py"
    shouldExist: true
```

図 7: 提案手法が生成するテスト

コマンドの引数にインストールされたバイナリの名前，期待される出力にインストールされたコマンドのパスを割り振る．fileExistenceTests ではテストの名前，テストファイルのパス，ファイルが存在するかを生成する．提案手法ではテストの名前にどの Docker 命令から生成されたか，テストファイルのパスにテスト対象のパス，ファイルが存在するかに True を割り振る．上記を各レイヤそれぞれに実施する．各レイヤから生成されたテストを最後に 1 つにまとめ，出力する．

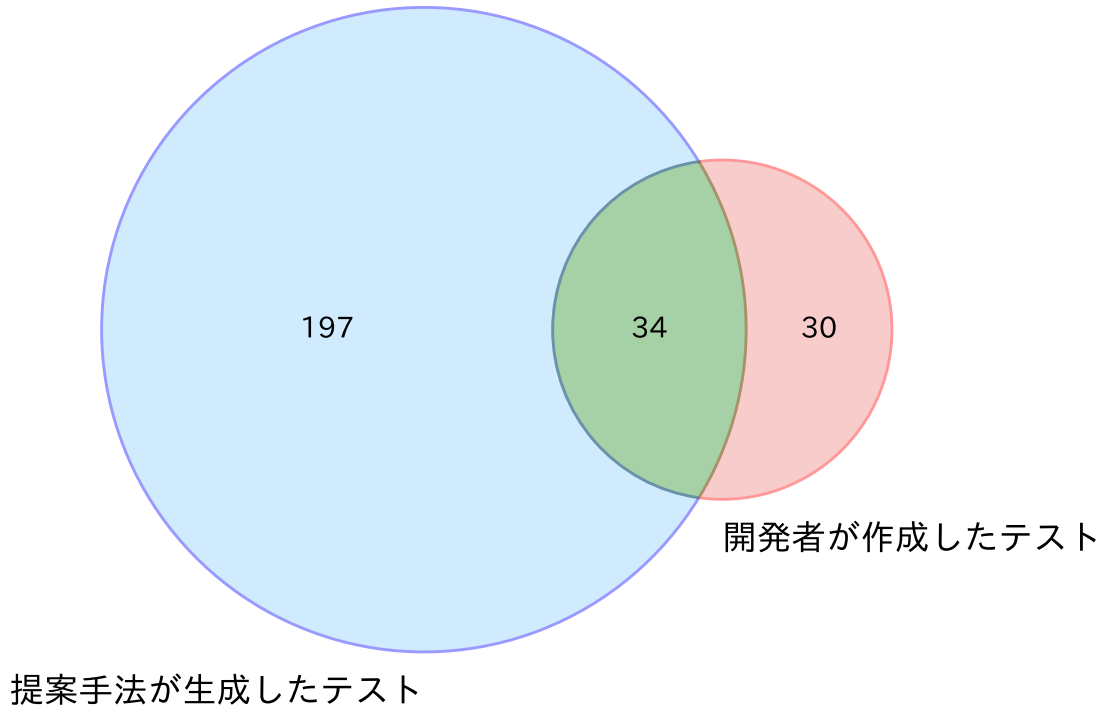

図 8: 提案手法が生成したテストと開発者が作成したテストのベン図

5 実験

5.1 実験概要

提案手法が適切にテスト生成できるか確認するために評価実験を行う．Container Structure Test を持つ Dockerfile に対し，提案手法によるテストの生成を実行する．各 Dockerfile の開発者が作成したテストと同じ内容のテストをどれだけ提案手法が生成できるのかを確認する．開発者が作成したテストと同じ内容のテストを生成できれば成功，できなければ失敗とする．今回用意した Container Structure Test を持つ Dockerfile は 20 個で，各 Dockerfile の持つテストの総数は 64 個である．これら 20 個の Dockerfile に対し提案手法によるテスト自動生成を実施する．

5.2 実験結果

提案手法が生成したテストと開発者が作成したテストとの関係を図 8 のベン図に示す．左の円が提案手法が生成したテスト，右の円が開発者が作成したテスト，共通部分が開発者が作成したテストの内，

```
commandTests:
- name: "cfn-lint installation"
  command: "which"
  args: ["cfn-lint"]
  expectedOutput: ["/usr/local/bin/cfn-lint"]
```

(a) 開発者が作成したテスト

```
commandTests:
- name: "generated from RUN pip install --no-cache-dir cfn-lint==${lint_version
    }"
  command: "which"
  args: ["cfn-lint"]
  expectedOutput: ["usr/local/bin/cfn-lint"]
```

(b) 提案手法が生成したテスト

図 9: 提案手法がテスト生成に成功したテスト

提案手法が生成できたテストである．各領域内の数字はそれぞれのテスト数を表す．提案手法は 20 個の Dockerfile から 231 個のテストを生成した．そのうち 34 個は開発者と同じ内容のテストを生成に成功した．しかし，30 個の開発者が作成したテストに関しては提案手法では生成に失敗した．以下，提案手法がテスト生成に成功したケース，テスト生成に失敗したケース，提案手法だけが生成したテストそれぞれについて詳細を述べる．

5.3 テスト生成に成功したケース

図 9 に提案手法がテスト生成に成功したテストを示す．開発者が作成したテスト（図 9a）と提案手法が生成したテスト（図 9b）を比較すると，テスト名以外が完全に一致している．また，完全一致はしていないがテスト生成に成功している場合も存在する．図 10 に該当するテストを示す．開発者が作成したテスト（図 10a）はテスト名にあるように aws がインストールされているかのテストである．提案手法が生成したテスト（図 10b）も aws がインストールされているか確認するテストである．よってこの 2 つのテストは実行しているコマンドは異なるがテスト内容は同一のため，テスト生成に成功したと考える．

5.4 テスト生成に失敗したケース

提案手法が生成できなかったテスト 30 個に対し分析したところ，以下の 3 つに分類される．

```yaml
commandTests:
- name: "aws cli installation"
  command: "aws"
  args: ["--version"]
  exitCode: 0
```

(a) 開発者が作成したテスト

```yaml
commandTests:
 - name: "generated from RUN pip install awscli"
  command: "which"
  args: ["aws"]
  expectedOutput: ["usr/bin/aws"]
```

(b) 提案手法が生成したテスト

図 10: 完全一致ではないが提案手法がテスト生成に成功したテスト

```yaml
commandTests:
- name: "Check Nginx Version"
  command: "nginx"
  args: ["-v"]
  expectedError: ["1.24."]
```

図 11: ベースイメージに対するテスト

1. ベースイメージに対するテスト：12 個
2. レイヤーに存在しない内容に対するテスト：7 個
3. Docker 命令の作用に対するテスト：11 個

FROM 命令により生成されたベースイメージの内容を確認するテストがいくつか見られた．図 11 に例を示す．このテストを持つ Dockerfile は FROM 命令で nginx:1.24.0-alpine というベースイメージを指定している．そのため，このテストはベースイメージ内の nginx のバージョンを確認するテストである．提案手法ではベースイメージの内容に対するテストは作成しないため，このようなテストの生成に失敗した．

次に，レイヤーに存在しない内容のテストの 1 つを図 12 に示す．このテストはコンテナを実行するホストのアーキテクチャが正しいかを調べるテストである．テストの内容は Docker 命令の作用ではないため，レイヤの変更差分にも表れない．よって提案手法では生成できない．その他のテストも同様に

```
commandTests:
# check that image builds for the right architecture
- name: "architecture check"
  command: "uname"
  args: ["-m"]
  expectedOutput: ["x86_64"]
```

図 12: レイヤーに存在しない内容のテスト

```
commandTests:
- name: "curl version"
  command: "curl"
  args: ["--version"]
  expectedOutput: ["curl 7\\.\\d+.*"]
```

図 13: Docker 命令の作用に対するテスト

```
fileExistenceTests:
- name: 'generated from RUN apk update'
  path: 'var/cache/apk/APKINDEX.49104001.tar.gz'
  shouldExist: true
```

図 14: 提案手法が生成した不要なテスト

Docker 命令の作用ではない内容をテストしていたことを確認した．

最後に，Docker 命令による作用にも関わらずテスト生成に失敗したテストを図 13 に示す．このテストはインストールされた curl の version を確認するテストである．提案手法では生成するテストは which による commandTests，fileExistenceTests の 2 つのみなので，それ以外の作用に対するテストの生成に失敗した．

5.5 提案手法だけが生成したテスト

提案手法だけが生成したテストが 197 個存在した．全てのテストを確認したところ，197 個中 56 個が不要と見なせるテストであった．不要なテストの 1 つを図 14 に示す．このテストは RUN 命令による apk update コマンドの実行により追加されたファイルであるが，キャッシュ内のため，このファイルに対するテストは不要である．他にも，tmp ファイルに対するテスト等が不要であると見なした．

一方，残りの 141 個のテストに関しては不要ではないテストであった．例えば，RUN 命令で複数のバイナリをインストールしている場合に，開発者が作成したテストはどれか 1 つがインストールされているのかテストしている．それに対し，提案手法はインストールしようとした全てのバイナリに対しテストする．このような，開発者が作成したテストでは確認しきれていない内容に対するテストを提案手法は生成していた．

6 考察

本節では実験結果を踏まえた考察を述べる．まず，それぞれの Dockerfile の開発者が作成したテスト 64 個に対して，提案手法は半分以上である 34 個のテストの生成に成功している．このことから，提案手法は適切なテストの生成が可能であると考えられる．次に，テストの生成に失敗した 30 個について述べる．開発者はベースイメージを指定し，更に必要なファイルの追加や，バイナリのインストール等をする．このベースイメージ以降の変更差分が重要であると考え，提案手法のテスト対象もこれらの変更差分である．そのため，テスト生成に失敗したテストの内，ベースイメージに対するテストとレイヤーに存在しない内容のテストに関しては提案手法のスコープ外である．しかし，提案手法の生成対象である作用の確認テストの生成に失敗したテストも存在する．生成できない理由はテストの生成ルールの不足によると考えられる．提案手法では which コマンドのテストと，fileExistenceTests のテストしか生成できない．そのためテスト生成対象の拡大する必要があると考える．最後に，提案手法だけが生成したテストは 197 個と既存のテスト数と比べると多い．中には不要なテストや，同じ命令に対し，過剰に生成しているテストが存在した．不要なテストに関して，一時ファイルやキャッシュ内のファイル等は生成しないように提案手法を改善する必要があると考える．また，自動テスト生成はあくまで支援ツールであり，自動で生成したテストについて開発者が必要なテストを確認し選択するという利用を考えているため，過剰に生成する場合については問題はないと考える．

7　妥当性の脅威

提案手法の効果を確認するために実施した評価実験では，Github 上の Container Structure Test を持つ Dockerfile 20 個に対し提案手法でテストを生成した．そのため，これらの Dockerfile 以外を利用して実験を実施した場合に同様の結果が得られるとは限らない．また，実験対象の Dockerfile はランダムに選択し，プロジェクトのスター数やコミット数，Dockerfile の行数といった要素を考慮していないことも妥当性の脅威といえる．大規模なプロジェクトや実際の現場で使用されている Dockerfile に対し提案手法を実行した場合，異なる結果が得られる可能性がある．

8　おわりに

本研究では Dockerfile のテストを自動生成する手法を提案した．Dockerfile は Java や C 言語等の手続き型言語とソースコードの性質が大きく異なる．そのため既存のアプローチが適用できない．そこで，本研究では OverlayFS に着目した．Docker イメージは OverlayFS を使用しており，複数のイメージレイヤから構成される．そして，各レイヤには Docker 命令の実行による変更差分が保存されている．提案手法のアイデアはこの Docker イメージ内の各レイヤの解析によるテストの生成である．提案手法では各レイヤの変更差分と Docker 命令の対応を取得し，それぞれの命令による変更差分を取得する．その情報を基に Container Structure Test のテストを生成する．

提案手法が適切にテストを生成できるかを確認するために評価実験を実施した．Container Structure Test のテストを持つ Dockerfile 20 個に対し，提案手法によるテスト自動生成を実行した．各 Dockerfile の開発者が作成したテストと同じ内容のテストを生成できるかどうか確認した．実験結果として，各 Dockerfile の開発者が作成したテスト 64 個のうち，半数以上の 34 個のテストを生成できたことを確認した．また，開発者が用意できていなかったテストも提案手法は生成した．一方，提案手法が生成できなかったテストに対しては今後の研究課題の 1 つである．

今後の研究課題として，生成に失敗したテストの生成と不要なテストの削除の 2 つが挙げられる．提案手法ではテスト生成ルールが少なく，一部のテストの生成に失敗した．より多くのテスト生成ルールをヒューリスティックに追加することで解決できると考える．不要なテストの削除に関しては，一時ファイルといった不要なファイルに対するテストを生成しないように改善する必要があると考える．

謝辞

本研究を遂行するにあたり，多くの方にご指導，ご支援を賜りました．

研究に対する的確な助言，貴重なご意見を楠本真二教授にいただきました．深謝の意を表します．研究指導はもちろんですが，他にもみかんやお菓子の差し入れは研究で疲労した時に精神的にも非常にお世話になりました．

非常に長い期間，本研究に対しご指導をいただきました柗本真佑助教に深謝いたします．学部 3 年生の時に初めて授業で教えていただいた時にこの人の元で学びたいと思い，ご指導頂いたこの 3 年間は私にとって学問に関してはもちろんのこと，人としても非常に多くの学びを得ることができました．改めて感謝の意を表します．

ソフトウェア工学講座の肥後芳樹教授には研究に関するご指摘や，私では気づけないような視点でのご意見をいただき，心より感謝いたします．研究以外でも研究室での雑談では面白い話を数多くしていただき，学生生活が非常に楽しいものとなりました．

事務補佐員の橋本美砂子氏には研究活動を行うにあたり，円滑に進められるよう多くのご支援をいただきました．また，研究室内での生活も快適に過ごせるように様々なご支援もいただきました．心より感謝申し上げます．

楠本研究室の同期，後輩，卒業した先輩方の皆様に関しては研究内容の議論や，疲労した時の何気ない会話など様々な面で支えていただきました．非常に楽しい学生生活を皆様と過ごせたこと，心より感謝します

最後に，ここまで私を支えてくれた家族，励ましや応援の言葉をかけていただいた友人へ感謝の意を表します．

参考文献

[1] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the international middleware conference*, pp. 1–13, 2016.

[2] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. A dataset of dockerfiles. In *Proceedings of the International Conference on Mining Software Repositories*, pp. 528–532, 2020.

[3] Michele Guerriero, Martin Garriga, Damian A Tamburri, and Fabio Palomba. Adoption, support, and challenges of infrastructure-as-code: Insights from industry. In *Proceedings of the international conference on software maintenance and evolution*, pp. 580–589, 2019.

[4] Docker. Docker overview. https://docs.docker.com/get-started/overview/(accessed 2024-01-25.

[5] K Hanayama, S Matsumoto, and S Kusumoto. Humpback: Code completion system for dockerfiles based on language models. In *Proceedings of the. Workshop on Natural Language Processing Advancements for Software Engineering*, pp. 1–4, 2020.

[6] Jordan Henkel, Christian Bird, Shuvendu K Lahiri, and Thomas Reps. Learning from, understanding, and supporting devops artifacts for docker. In *Proceedings of the international conference on software engineering*, pp. 38–49, 2020.

[7] Yu Zhou, Weilin Zhan, Zi Li, Tingting Han, Taolue Chen, and Harald Gall. Drive: Dockerfile rule mining and violation detection. *ACM Transactions on Software Engineering and Methodology*, Vol. 33, No. 2, pp. 1–23, 2023.

[8] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[9] Yujuan Jiang and Bram Adams. Co-evolution of infrastructure and source code-an empirical study. In *Proceedings of the Working Conference on Mining Software Repositories*, pp. 45–55, 2015.

[10] Yang Zhang, Bogdan Vasilescu, Huaimin Wang, and Vladimir Filkov. One size does not fit all: an empirical study of containerized continuous deployment workflows. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 295–306, 2018.

[11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing

based on java predicates. *Journal on ACM SIGSOFT Software Engineering Notes*, Vol. 27, No. 4, pp. 123–133, 2002.

[12] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan De Halleux, and Wolfram Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the European software engineering conference and the symposium on The foundations of software engineering*, pp. 193–202, 2009.

[13] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0.* IEEE Computer Society Press, 2014.

[14] Paul Ammann and Jeff Offutt. *Introduction to software testing.* Cambridge University Press, 2016.

[15] Glenford J. Myers, Corey Sandler, and Tom Badgett. *The Art of Software Testing.* Wiley Publishing, 2011.

[16] Gregory Tassey. The economic impacts of inadequate infrastructure for software testing. *Journal on National Institute of Standards and Technology*, 05 2002.

[17] P. Runeson. A survey of unit testing practices. *Journal on IEEE Software*, Vol. 23, No. 4, pp. 22–29, 2006.

[18] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 213–223, 2005.

[19] Carlos Pacheco, Shuvendu K Lahiri, Michael D Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the International Conference on Software Engineering*, pp. 75–84. IEEE, 2007.

[20] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the symposium on Principles of programming languages*, pp. 47–54, 2007.

[21] Bogdan Korel. Automated software test data generation. *IEEE Transactions on software engineering*, Vol. 16, No. 8, pp. 870–879, 1990.

[22] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: adaptive random testing for object-oriented software. In *Proceedings of the international conference on Software engineering*, pp. 71–80, 2008.

[23] Tsong Yueh Chen, Hing Leung, and Ieng Kei Mak. Adaptive random testing. In *Proceedings of the Higher-Level Decision Making: Asian Computing Science Conference.*, pp. 320–329,

2005.

[24] Carlos Pacheco and Michael D Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the European Conference Object-Oriented Programming*, pp. 504–527, 2005.

[25] James C King. Symbolic execution and program testing. *Journal on Communications of the ACM*, Vol. 19, No. 7, pp. 385–394, 1976.

[26] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *Journal on ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 5, pp. 263–272, 2005.

[27] Jürgen Cito, Gerald Schermann, John Erik Wittern, Philipp Leitner, Sali Zumberi, and Harald C. Gall. An empirical analysis of the docker container ecosystem on github. In *Proceedings of the International Conference on Mining Software Repositories*, pp. 323–333, 2017.

[28] Portworx. Annual container adoption report, 2019. `https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf` (accessed 2024-01-25).

[29] Charles P Wright and Erez Zadok. Kernel korner: unionfs: bringing filesystems together. *Journal on Linux Journal*, Vol. 2004, No. 128, p. 8, 2004.

[30] Emna Ksontini, Marouane Kessentini, Thiago do N Ferreira, and Foyzul Hassan. Refactorings and technical debt in docker projects: An empirical study. In *Proceedings of the International Conference on Automated Software Engineering*, pp. 781–791. IEEE, 2021.

[31] Changyuan Lin, Sarah Nadi, and Hamzeh Khazaei. A large-scale data set and an empirical study of docker images hosted on docker hub. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pp. 371–381, 2020.

[32] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of Conference on Data and Application Security and Privacy*, pp. 269–280, 2017.

[33] Peiyu Liu, Shouling Ji, Lirong Fu, Kangjie Lu, Xuhong Zhang, Wei-Han Lee, Tao Lu, Wenzhi Chen, and Raheem Beyah. Understanding the security risks of docker hub. In *Proceedings of the European Symposium on Research in Computer Security*, pp. 257–276, 2020.

[34] Christoph Csallner and Yannis Smaragdakis. Jcrasher: an automatic robustness tester for java. *Journal on Software: Practice and Experience*, Vol. 34, No. 11, pp. 1025–1050, 2004.

[35] Carlos Pacheco and Michael D Ernst. Randoop: feedback-directed random testing for java. In *Proceedings of the conference on Object-oriented programming systems and applications companion*, pp. 815–816, 2007.

[36] Xiangjun Liu and Ping Yu. Randoop-tsr: Random-based test generator with test suite reduc-

tion. In *Proceedings of the Asia-Pacific Symposium on Internetware*, pp. 221–230, 2022.

[37] Shay Artzi, Adam Kiezun, David Glasser, and Michael D Ernst. Combined static and dynamic mutability analysis. In *Proceedings of the International Conference on Automated Software Engineering*, pp. 104–113, 2007.

[38] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the conference on Foundations of software engineering*, pp. 416–419, 2011.

[39] Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering*, Vol. 41, No. 3, pp. 294–313, 2015.