2019 年度卒業論文

Docker Compose ファイルと UML 図の互換及び検証をサポートする astah* プラグインの実装

信州大学工学部電子情報工学科

16T2133H 野々川麗樺

令和 2 年 2 月

目次

1 はじめに

1.1 背景と目的

近年情報システム界で DevOps やアジャイル開発といった開発・運用の手法が盛んに取り入れられている．できるだけ素早く，軽量的という開発スタイルは従来の開発手法に比べ，パッケージの管理，インフラストラクチャの管理などシステムの運用が複雑である上，デプロイが頻繁に行われ，人為的なミスが多く発生する．

デプロイの手間と管理を短縮するため，Docker 社が開発した Docker はサービス基盤の構築で幅広く使用されている [1]．Docker を用いることで，ビルドや環境パッケージングの手間が省かれ，システム環境の違いで発生しうる問題も解決される．

しかし，Docker 自体は単体のホスト上で単体のコンテナを動かすというのを目的としているため，複数のコンテナや複数のホストが必要とされるサービス基盤での実用が困難である．複数のコンテナの管理を可能にするために Docker 社は Docker Compose をリリースしている．Docker Compose は Docker エンジンに基づいており，マルチコンテナの環境をローカルで管理するツールである．Docker Compose で用いるマルチコンテナ設計の効率向上及び正確性の向上をサポートするのが本研究の目的である．

1.2 研究の概要

Docker Compose は，実行時にユーザーから提供された YAML 形式の設定情報を読み込み，環境を構築し，複数のコンテナを一度に起動することができる．複数のコンテナを動かす以上，コンテナ間の依存関係やネットワークについての記述が一般的であるが，YAML テキストでは直観的な記述ができない．

そこで，YAML テキストと UML 図の互換機能を astah[2] に実装し，さらに YAML 編集用のユーザーインターフェースを設けることで，外部コンテンツに頼らないマルチコンテナ環境設計を可能にする．その上，リファレンス検査及び妥当性の検証を UML 図での設計時に動的に行い，エラーを早い段階で発見し，正確性を高める．

1.3 論文構成

第 2 章では Docker コンテナの構造と原理及び応用と拡張性について述べる．第 3 章では Docker Compose について紹介をし，Docker Compose を用いたシステム設計における課題について述べる．第 4 章では前章で取り上げた課題を解決するための手法を提案し，述べる．第 5 章では前章の提案手法を astah* プラグインによる実装と評価について述べる．第 6 章ではまとめと今後の課題について述べる．

2 Docker

2.1 概要

2.1.1 Docker とは

Docker とは，Docker 社が開発しているコンテナ型の仮想環境の作成，配布，実行を行うためのプラットフォームである．ミドルウェアのインストールや各種環境設定をコード化して管理し，それを共有することで，どんな環境でも同じコンテナを作成し，動かすことができる．

例えば，OS の違いや，ミドルウェアのバージョンの違いにより，デベロッパの開発環境では正常に作動するのが，デプロイ環境で動作しないというケースが度々発生する．Docker を活用することで，開発工程の中で使っていた環境をそのままデプロイ環境に持っていくことが可能なため，環境差分が少なく，環境による問題を減らすことができる．

2.1.2 Docker コンテナ実行例

Docker Hub は，Docker 社が運営しているイメージを自由に公開できるイメージのレポジトリである．Mysql や tomcat といった公式のイメージやユーザーがアップロードしたカスタマイズのイメージなど様々な Docker イメージが公開されている．

本節では，例として Nginx の Docker イメージを使用した Web サーバーの構築を行う．Docker がインストールされた環境で以下のコマンドを実行する．

```
1 $ docker run -d -p 8080:80 nginx
```

このコマンドより Docker は以下の操作を行う．

1. Docker hub から nginx がインストールされているコンテナイメージを取得する
2. 取得したコンテナイメージでコンテナを起動する
3. コンテナが起動されたら，コンテナの中で Nginx を起動する

その後，ブラウザで http://<IP アドレス >:8080/を開くと，図 1のように Nginx の Web サーバーにアクセスすることができる．

図 1: Docker コンテナで構築した Nginx の Web サーバー

2.2 Docker コンテナの構造と原理

2.2.1 従来の仮想化技術との違い

Docker は「コンテナ型仮想化技術」を実現したプロダクトである．従来の仮想化（ホスト型仮想化）は図 2のように，ホスト OS 上に仮想化ソフト/ハイパーバイザ（Virtal Box や VMware vSphere）を使用して，仮想マシン/ゲスト OS を立ち上げ，その仮想マシンの中でミドルウェアの環境構築をし，プログラムを実行してアプリケーションを動かすという構造である．

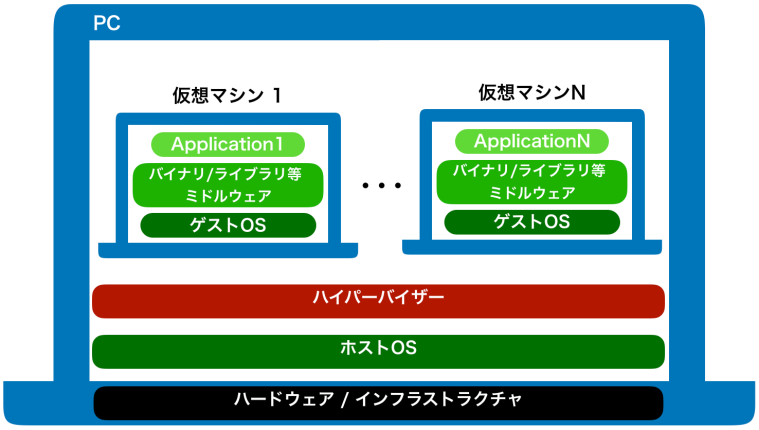

図 2: 従来の仮想化技術（文献 [1] から引用）

一方で，Docker が提供するコンテナ型仮想化は，図 3の構造である．従来のホスト型仮想化とは異なり，Docker はゲスト OS を起動しない．Docker エンジンは，コンテナと呼ばれるミドルウェアを構築できる実行環境を作成し，その中でアプリケーションを動作させる．

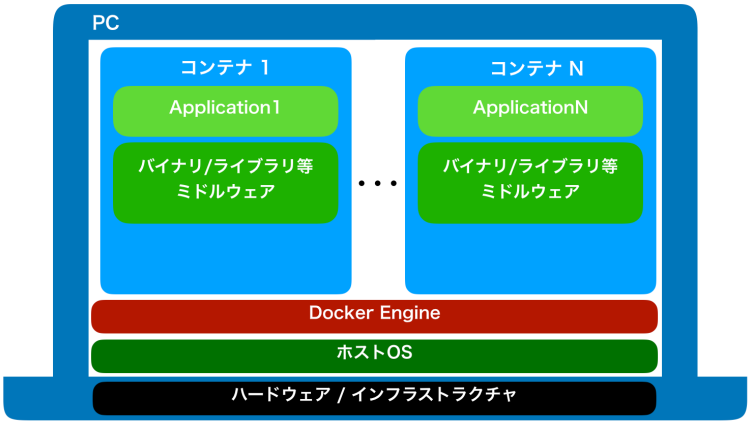

図 3: Docker コンテナによる仮想化技術（文献 [1] から引用）

これによって従来のホスト型仮想化よりも圧倒的に軽量に動作するのがコンテナ型仮想化の特徴である．

2.2.2 イメージからコンテナの構築

2.1.2節の Nginx の実行例では，コンテナを構築するために Docker イメージを使用した．イメージとは，コンテナ，つまりアプリケーションの実行環境を起動するために必要な設定ファイルをまとめたものである．Docker ではこのイメージを共有することで，様々なマシンで同じコンテナ（実行環境）を動作させることができる．イメージは Dockerfile に設定情報を記述し，Docker エンジンに通すことで作成することができ，Docker Hub でイメージを共有することができる．

2.3 Docker の拡張と応用

2.3.1 Docker エンジンを拡張したツール

Docker は単一ホストで単一コンテナの管理を目的としているため，実際のサービス基盤では実用てきでない．その代わり，Docker エンジンを拡張した実用的なツールがたくさんある．

例えば，Docker Compsoe は単一ホストで複数コンテナの管理する際に使用するツールである．それぞれのコンテナの設定情報を docker-compsoe．yml と呼ばれる YAML テキストに記述することで，複数のコンテナを同時に起動できる．コンテナの設定を一つずつ行う必要がなくなり，デプロイの手間を省くことができる．

また，Docker エンジンを搭載した仮想マシンの管理は Docker Machine で行うことができる．Docker Machine は，Docker エンジンを搭載した仮想マシンの管理（作成，起動，停止，再起動など）をコマンドラインから実行できるツールである．仮想化ソフト（Virtual Box）をドライバーに使用して，Docker エンジンを搭載した仮想マシンの管理を行うことができる．

複数台の Docker ホストマシンの間でコンテナ間通信の設定・管理を自動化するには，通常コンテナオーケストレーションエンジンを使用する．ツールとして Docker 社の Docker Swarm や Google 社の Kubernetes などがあり，開発の現場で多く使われるツールである．

2.3.2 Docker 応用事例

Docker を用いることで，環境構築やパッケージ管理を容易になり，さらにオーケストレーションツールを使用することで，コンテナの自動管理が可能になることから，多くの企業が Docker を導入するようになった．

例えば，株式会社リクルートテクノロジーズでは，Docker を開発環境の構築に使用している．Docker を活用したことで，環境構築のリードタイム 99%を削減し，特定の人間に依存していた作業を撤廃し属人性を排除した．また，株式会社ワークスアプリケーションズでは Jenkins を使った CI 環境（継続的インテグレーション）の再構築に Docker を利用している．複数バージョンのアプリケーション保守を行うのに，ソースコードに対応したバージョン別の環境を準備してテストを実行することで，バージョン別の実行環境の差異によるトラブルを解決している．株式会社インターネットイニシアティブ（IIJ）は Hadoop/Hive を用いたデータ分析の計算ノード部分で Docker を利用している．ユーザー毎に実行環境の隔離やノード数の増減を必要とすることから，隔離が可能でかつ起動が高速なコンテナ技術を採用した．自社開発のオーケストレーションの doma でコンテナの管理を行っている．

3 Docker Compose

3.1 概要

3.1.1 Docker Compose とは

Docker Compose は，コンテナの設定情報を事前に定義して実行することができ，ローカルで複数コンテナを管理するのに用いられるツールである．例えば，コンテナ一つを起動するにも，volume などのマウント設定や所属させる Docker ネットワークの設定などを明記する必要があり，複雑なオプション付きのコマンドを入力する必要があり，手間がかかりミスの原因ともとなる．

ソースコード 1: 複雑な Docker コマンドの例

```
1 $ docker run -itd ー name mount-test ー mount source=volume-test, destination=/etc/nginx,readonly
      nginx
```

しかし，Docker Compose では，複数のコンテナの定義を一つの docker-compose.yml ファイルに記述しておくことによって，コンテナをまとめて起動・設定することができる．

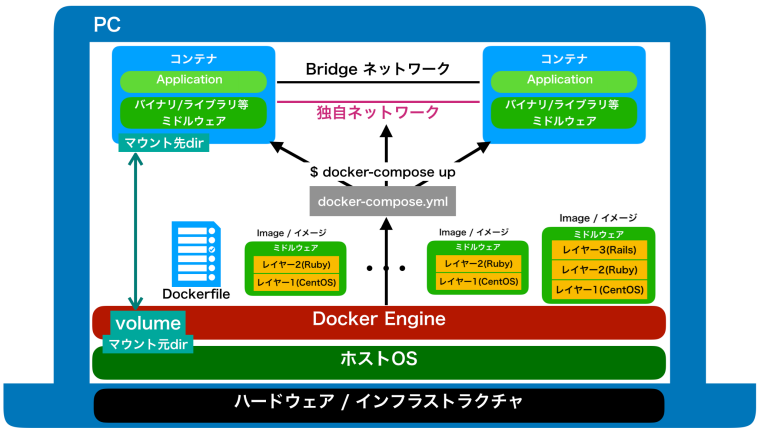

図 4: Docker Compsoe と Docker の関係（文献 [1] から引用）

Docker Compose を使用した構築の手順は以下の通りである．

1. Dockerfile を用意するか Docker Hub などに使用するイメージを用意する
2. docker-compose.yml を定義する
3. YAML ファイルがあるディレクトリで $ docker-compose up を実行する

次節はこの手順に従い Docker Compose の実行例について紹介する．

3.1.2 Docker Compose によるアプリ実行環境構築例

一例として，Django を使用した Python の Web サーバーと Postgres の DB サーバーからなる Web アプリケーションを構築する．

まず，作業ディレクトリを用意し，ソースコード 2のように Python サーバーを構築できる Dockerfile を作成する．

ソースコード 2: Python Web サーバーの Dockerfile

```
# image名にpython3実行環境のイメージを指定
FROM python:3
# pythonの標準出力をバッファにため込まないための環境変数設定
ENV PYTHONUNBUFFERED 1
# serviceディレクトリを作成
RUN mkdir /service
# serviceディレクトリに移動
WORKDIR /service
# requirements.txt(事前に作成)をserviceディレクトリに置く
COPY requirements.txt /service/
# pip  installでパッケージをインストール
RUN pip install -r requirements.txt
# buildコンテキストの内容を全て/service内に置く
COPY . /service/
```

次に，ソースコード 3のように docker-compose.yml ファイルを作成する．このファイルで Python の Web サーバーと Postgres の DB サーバー両方の設定情報について記述する．起動順序に影響するため，サービスの依存関係についての記述し，image か build を使ってイメージを指定することができる．

ソースコード 3: Web アプリケーションを構築するための docker-compose.yml

```
# docker-composeのversion指定
version: '3'
# 起動するサービスコンテナを書いていく
services:
  # dbサーバーコンテナを起動
  db:
   # イメージはdockerhub上のpostgres:latestを使用
   image: postgres
  # webサーバーコンテナを起動
  web:
   # カレントディレクトリにあるdockerfileからimage作成
   build: .
   #コンテナ起動時に実行されるコマンドを指定
   command: python3 manage.py runserver 0.0.0.0:8000
   # currentディレクトリを/appディレクトリにbind mountする
   volumes:
    - .:/app
   # コンテナの8000番を公開
   posts:
    - "8000:8000"
   # webサーバーコンテナが起動する前にdbサーバーコンテナが起動するようにする
   depends_on:
    - db
```

最後に docker-compose.yml のある作業ディレクトリで，以下のコマンドを入力することで，二つのサーバーからなる Web アプリケーションが起動される．

```
$ docker-compose up
```

逆に，次のコマンドを実行すると，作成されたコンテナやネットワークを削除することができる．

```
$ docker-compose down
```

また，次のコマンドを実行すると docker-compose.yml のリファレンス検査を行うことができる．

```
$ docker-compose config
```

3.1.3 Docker Compose version3 リファレンス

本節では，docker-compose.yml に用いるリファレンス [3] について，使用頻度の高いものを抜粋して紹介する．

build

指定したディレクトリにある dockerfile でイメージを作成する．

```
#シンプル記述
build: ./dir
#コンプレックス記述
build:
  context: ./dir
  dockerfile: Dockerfile-alternate
  args:
    buildno: 1
```

image

コンテナを実行時に元となるイメージを指定する．リポジトリ名・タグあるいはイメージ ID の一部を指定できる．

```
#デフォルトでリポジトリ=Docker Hub,tag=lastestとなる
image: redis
#ta(バージョン)が14.04のイメージを指定する.
image: ubuntu:14.04
#tutumというリポジトリを指定する
image: tutum/influxdb
#イメージIDを指定する
image: a4bc65fd
```

ローカルにイメージが存在しない場合，リポジトリから pull（取得）を試みる．

command

コンテナが構築された際に実行するコマンドを指定できる．

```
command: bundle exec thin -p 3000
#配列形式で引数を渡しでも良い
command: ["bundle", "exec", "thin", "-p", "3000"]
```

ports

公開用のポートである．これを定義することで，コンテナ外部（ホスト側）からもアクセスが可能になる．ホスト側とコンテナ側の両方のポートを指定するか，コンテナ側のポートのみを指定ができる（この場合，ホスト側はランダムなポートが選ばれる）．

```
ports:
 #コンテナ側の3000ポートを公開する
 - "3000"
 #コンテナ側の3000から3005ポートを公開する
 - "3000-3005"
 #コンテナ側の8000ポートを公開し,ホストの8000からアクセスできる
 - "8000:8000"
```

depends_on

コンテナの依存関係を定義する．次の例では，‘web’ は ‘db’ と ‘redis’ に依存する．依存関係によって起動順序が前後される．

```
version: '3'
services:
  web:
    build: .
    depends_on:
      - db
      - redis
  redis:
    image: redis
  db:
    image: postgres
```

links

他のサービスとのリンクを定義する．サービス名の指定に加え，リンク用エイリアスを指定することでが可能である．depends_on と同じく，記述によって起動順序が前後される．

```
links:
 - db # サービス名
 - db:database #サービス名:エイリアス
 - redis
```

networks

ネットワークを定義することができる．同じネットワークを持つ場合，サービス名やエイリアスをホスト名として，互いをアクセスすることができる．

下の例では，３つのサービス（‘web’，‘worker’，‘db’）と２つのネットワーク（‘new’ と ‘legacy’）が提供されている．‘db‘ サービスはホスト名 ‘db’ または ‘database’ として ‘new’ ネットワーク上で到達可能である．そして，‘legacy’ ネットワーク上では ‘db’ または ‘mysql’ として到達できる．

```yaml
version: "3.7"

services:
  web:
    image: "nginx:alpine"
    networks:
      - new

  worker:
    image: "my-worker-image:latest"
    networks:
      - legacy

  db:
    image: mysql
    networks:
      new:
        aliases:
          - database
      legacy:
        aliases:
          - mysql

networks:
  new:
  legacy:
```

3.2 Docker Compose における課題

3.2.1 耐保障性における課題

Docker Compose は restart リファレンスでコンテナが停止した際の再起動設定を行えるが，頻繁な再起動が行われた場合，ホスト全体をダウンさせる可能性がある．コンテナの監視やリソース状態に応じた自動修復機能がない．

3.2.2 システム設計と正確性における課題

Docker Compose は複数のコンテナの管理を目的とするため，コンテナ間に依存関係が存在する場合が多い．ユーザーは docker-compose.yml に事前に依存関係を含む設定情報を記述するが，YAML テキスト形式では，依存関係やネットワークについて直観的な記述ができない．このため，設計の際に間違いが生じたり，読み手がテキストからアプリケーションのアーキテクチャを推測しにくいと考えられる．また，ソースコード 4のように一つのサービスを削除した場合，これを引用した箇所すべてを編集する必要がある．

ソースコード 4: service-d 削除前

```
1  version: "3.7"
2  services:
3    service-a:
4      build: ./service-a
5      depends_on:
6      - service-c
7      - service-d
8    service-b:
9      build: ./service-b
10     depends_on:
11     - service-c
12     - service-d
13   service-c:
14     build: ./service-c
15     depends_on:
16     - service-d
17
18   service-d:
19     build: ./service-d
```

ソースコード 5: service-d 削除後

```
20 version: "3.7"
21 services:
22   service-a:
23     build: ./service-a
24     depends_on:
25     - service-c
26 #   - service-d 削除
27   service-b:
28     build: ./service-b
29     depends_on:
30     - service-c
31 #   - service-d 削除
32   service-c:
33     build: ./service-c
34     depends_on:
35 #   - service-d 削除
36
37 #  service-d: 削除したservice-d
38 #    build: ./service-d
```

Docker Compose では安全のため起動時に $ docker-compose config と同じ機能のプログラムで docker-compose.yml のリファレンス検査を行う．システム設計段階では，環境内に実際に使用する Dockerfile やボリュームファイルが用意できない場合があるが，検査プログラムではファイルが特定できない場合エラーが発生し，検査が中止されるため，そのほかの項目の正確性確認ができない．

検査プログラムは該当バージョンのリファレンスと一致するかの検査がメインであるため，ポートの重複やサービス名の重複の検査は行われず，これが原因で起動中に問題が発生する場合がある．サービスの孤立などの妥当性の検証もこの時は行われない．

Docker Compose は Docker と比べ，事前に定義した設定情報を元にコンテナ環境の構築を一斉に実行するため，コンテナ起動の手間が省かれる．その一方で，エラーも一斉に起こりうるため，デバッグは容易でないと考えられる．

4 提案手法

4.1 UML による設計方式の概要

YAML テキストでの設計は直観的でないことからミスが生じやすく，可読性が低いためコミュニケーションコストの高い記述形式である．本研究では，YAML を統一モデリング言語 UML（Undified Modeling Launguage）の合成構造図に互換して設計を行う手法を考案した．

4.1.1 統一モデリング言語 UML

統一モデリング言語 UML（Unified Modeling Language）は，オブジェクト指向分析や設計を行うために記法が統一されたモデリング言語である [4]．UML2.0 以降では 13 の図法があり，用途で図を使い分けることができる．プログラミングの知識がなくても図による感覚的な理解が可能であり，ソフトウェア開発に限らず広く使われている．

4.1.2 合成構造図

UML の図法である合成構造図はクラス図に似ているが，クラス図は属性や振る舞いを含めたクラス構造の静的なビューをモデリングする [5]．一方，合成構造図は実行時の構造や利用方法･要素間の関連など静的なダイアグラムでは表現できない部分を表現するのに利用される．

4.2 Docker Compose YAML と UML の互換

docker-compose.yml のリファレンスのほとんどは環境変数，ボリュームファイル設定といったコンテナ内のプログラムで使用する設定情報が多いが，これは設計者やアーキテクチャを把握したい開発者にとって重要情報ではない．コンテナ間の依存関係やホストとの関係を記述したリファレンスのみを抜粋し，UML 図との互換を行う．これによって，Docker や Docker Compose の知識を持たない人さえも UML 図を通じてシステムのアーキテクチャがわかるので，説明の手間が減り，情報交換が容易になる．

4.2.1 合成構造図モデルとリファレンスの対応付け

本研究では，UML ツール astah の合成構造図モデルに特定し，モデルとリファレンスの対応付けを表 3に定義する．

astah モデル	docker-compose リファレンス	モデル設定
構造化クラス	service	
パート	image	黄色
	build	グレー
依存	depends_on	
	links	
	networks	色付き

表 1: 合成構造図モデルとリファレンスの対応付け

リファレンスのうち，depends_on，links，networks の 3 つとも依存のモデルに対応付けされるが，実際に依存モデルとして生成されるのは depends_on，links のみであり，networks によって図に示す依存モデルの名前や色が決められる．例えば，“サービス A” が “サービス B” に依存しており，両者は “ネットワーク 1” を唯一の共通ネットワークとして持っている場合，“サービス A” は “ネットワーク 1” を通じて “サービス B” に依存していると言える．なのでこの時，“ネットワーク 1” を依存モデルの名前とし，両者間に示す．一方，両者間に複数の共通ネットワークが存在する場合，共通ネットワークのうち使用頻度の高いネットワークを依存モデルとして図に示す．

表 3に従ってソースコード 6を UML 図にすると図 5になる．イメージの由来，ポートと依存関係といった，docker-compose.yml を把握するうえで重要な部分のみが示されている．

ソースコード 6: 対応付けを示すための docker-compose.yml

```
version: "3.7"
services:
  service-a:
    build: "./path"
    volumes:
    - data:/var/data
    networks:
    - "network"
    depends_on:
    - "service-b"
    ports:
    - "80:80"
  service-b:
    image: "image-name"
    environment:
      WORDPRESS_DB_HOST: db:3306
    ports:
    - "90:90"
    networks:
    - "network"
networks:
  network: null
```

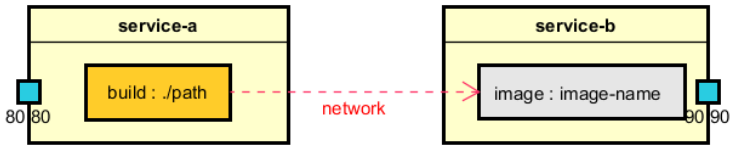

図 5: ソースコード 6を示した合成構造図

4.2.2 合成構造図におけるサービスの配置位置

変換した合成構造図をより理解しやすくするために，依存関係を考慮したうえで，配置位置を決める必要がある．ここでは，扇円を描く配置方式を提案する．例えば，Web アプリケーションの構築を行うには，通常ゲートウェイ，Web サーバー，データベースの順で上位のサーバーが下位に依存する．このように，下位の基盤となるコンテナほど多く依存される傾向がある．なので，他のコンテナに多く依存しているサービスほど，扇円の中央に近づき，多く依存されているものほど外側になるように配置することで，読み手は UML 図の上から順にアーキテクチャを理解できるようになる．

次にネットワークについてさらに考慮すると，同じネットワークを持つコンテナは何等かの関係を持つ場合が多い．そこで，同じネットワークを持つコンテナを縦のブロックに集中して配置することで，よりコンテナ関係の理解が容易になる．

言い換えるなら，コンテナは図 6のような扇円上に広がり，依存の数で “行” が決められ，ネットワークで “列” が決められる．

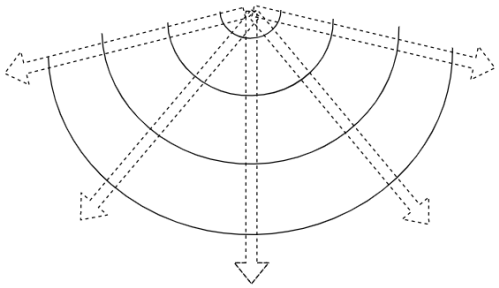

図 6: 扇円に広がるコンテナ配置

4.2.3 YAML 編集のためのユーザーインターフェース

UML 図で docker-compsoe.yml を表現することでアーキテクチャの理解が容易になったが，図に示されていない環境変数といったコンテナの詳細設定もデプロイや開発時に欠かせない情報である．なので，これらの情報の読み取りと編集ができる手段が必要である．

例えば，astah のモデルにはタグ付き値という機能があり，これを使用して情報をモデルに格納できる．しかし，タグ付き値はキーバリューのペアである必要があり，一方，対象の YAML はキーバリューでなく木構造に似た形式となっている．図 7では，リファレンスをタグ付き値の名前とし，記述の YAML 情報を JSON にしたうえで値に格納できるが，これでは可読性が低く，YAML の再利用ができないため推奨できない．

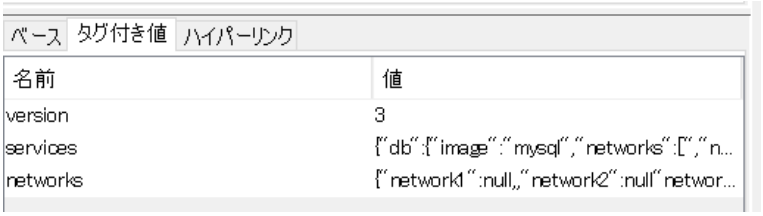

図 7: タグ付き値による詳細情報の格納

そこで，自由度の高い YAML エディターの機能を持つ UI について提案する．この YAML エディターは UML 図と連動し，UML 図上でタップしたコンテナに関する詳細情報を元の YAML のままで部分的に取り出しエディターで示す．この時，エディターからコンテナについて編集，追加，削除を行うと，元の YAML が上書きされ，UML 図がリロードされる．また，編集に関係する箇所を自動的に反映するので，一度の編集が全体に反映され，人為的なミスを減らすことが可能である．

4.3 コンテナの正確性を高める検査と検証の手法

docker-compose.yml のチェックは Docker Compose で起動した時に初めて行われるため，エラーの発見が遅れる場合がある．ここでは，新たに動的なリファレンス検査と妥当性検証のシステムを提案する．早い段階でエラー発見ができるように，3 つのタイミングでチェックを行う．

- YAML を UML に変換する時
- UML を YAML に変換する時
- 実装の YAML エディターで YAML に変更が加わった時

以上の段階で $ docker-compose config と同じ機能のプログラムを実行し，docker-compose.yml のリファレンス検査を行う．ただし，$ docker-compose config は指定のボリュームファイルが既存するかをチェックするが，設計段階の環境に考慮し，ここではファイルのチェックを行わない．

さらに，リファレンスの検査だけでなく，Docker Compose にない妥当性の検証も追加することで，コンテナ起動時及び起動後でエラーによる中断のリスクを減らす．妥当性検証は次のような要項があげられる．

(1) ワーニングを示す箇所

- サービスが孤立している
- サービス間の依存がループになっている

(2) エラーを示す箇所

- 同じ名前のサービスが存在する
- 他のサービスとポートが被っている
- 既存でないネットワークに依存している

ソースコード 7が妥当性に問題ある docker-compose.yml の例である.

ソースコード 7: 妥当性に問題ある docker-compose.yml の例

```
version: "3"
services:
  # a,b,cの依存関係がループとなっている.
  a:
    build: "./a"
    depends_on:
    - "b"
  b:
    build: "./a"
    depends_on:
    - "c"
  c:
    build: "./a"
    depends_on:
    - "a"
  #dが孤立したサービスである.
  d:
    build: "./d"
```

5 astah* プラグインによる実装と評価

5.1 Docker Compose YAML と UML の互換機能の実装

前章で提案した手法について，UML ツールの astah にプラグインとして実装を行った．実装で使用したプログラミング言語は Java である．使用したライブラリーは，YAML テキストを解析できる Jackson と Python コードランナーの Jython[6] があり，Maven でプロジェクト管理を行っている．

プラグインのソースコードのクラスのうち，YAML のリバース，コンポーズ，編集については ComposeDiagramManager，リファレンス検査は ComposeReferenceChecker，妥当性検証は ComposeValidationChecker で実装している．

5.1.1 Docker Compose YAML から UML を生成するリバーサー

リバーサーでは，Docker Compose YAML を読み込み，UML 図を生成する．この時，UML 図を新規で生成するか，既存の docker-compose.yml をインポートし生成するかの二つの方式がある．新規で生成する場合，ソースコード 8と同じ YAML テキストがテンプレートとして，UML 図にリバースされる．

ソースコード 8: 新規作成のためのテンプレート

```
1  version: "3.7"
2  services:
3    sample-service1:
4      build: ./dir1
5      networks:
6        - sample-network
7      depends_on:
8        - sample-service2
9    sample-service2:
10     build: ./dir2
11     networks:
12       - sample-network
13 networks:
14   sample-network: null
```

いずれにしても，プラグインは YAML テキストを読み込み，リファレンスなどのチェックをしたのち，Java のオブジェクトに変換する．その後，astah の API を用いて合成構造図を作成し，オブジェクトに変換されたサービスや依存を astah モデルとして図に示し，ネットワークと依存の数に応じて配置位置の調節を行う．既存の docker-compose.yml からリバースする際のフローを図 9に示す．

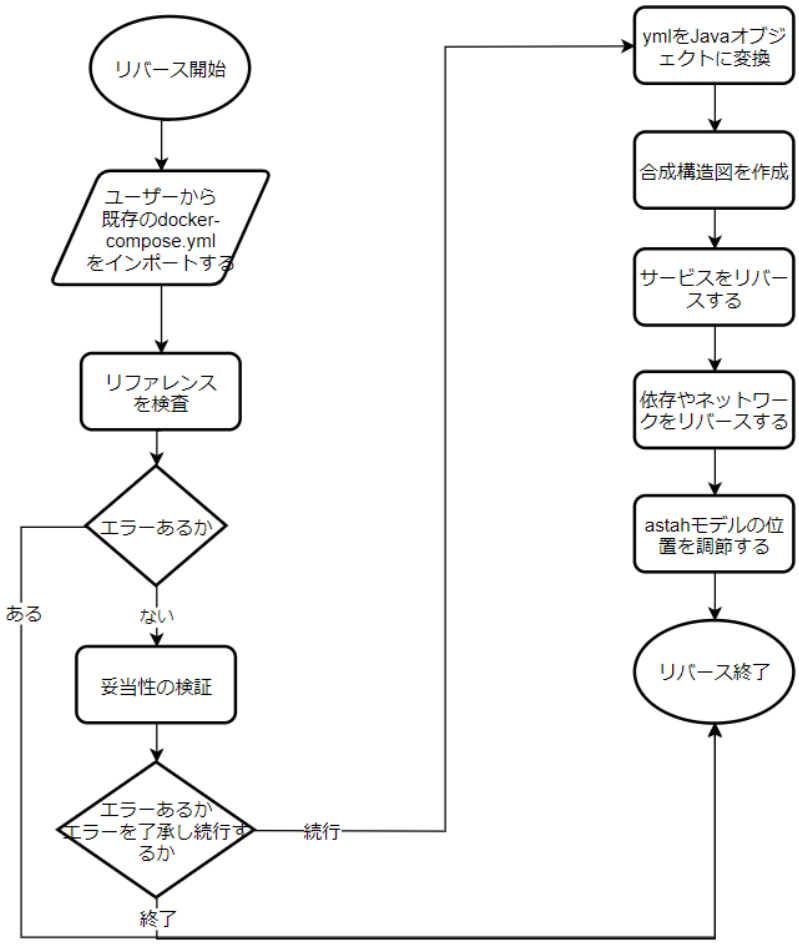
リバース開始
ユーザーから
既存のdocker-
compose.yml
をインポートする
リファレンス
を検査
エラーあるか
ある
ない
妥当性の検証
エラーあるか
エラーを了承し続行す
るか
続行
終了
ymlをJavaオブジ
ェクトに変換
合成構造図を作成
サービスをリバー
スする
依存やネットワー
クをリバースする
astahモデルの位
置を調節する
リバース終了

図 8: リバースのフロー

リバーサーを実装したプラグインでソースコード 9をリバースした結果が図 9である.

ソースコード 9: docker-compose.yml の例

```
version: "3"
services:
  db:
    image: "mysql"
    networks:
    - network2
  mq:
    image: "rabbitmq"
    networks:
    - network1
  nosql:
    image: "redis"
    networks:
    - network1
    - network2
  gateway:
    build: "./gateway"
    ports:
    - "8080:8080"
    networks:
    - network1
    - network2
    depends_on:
    - "service-a"
    - "service-b"
  service-a:
    build: "./service-a"
    networks:
    - network1
    depends_on:
    - "mq"
    - "nosql"
  service-b:
    build: "./service-b"
    networks:
    - network2
    depends_on:
    - "db"
    - "nosql"
networks:
  network1:
  network2:
```

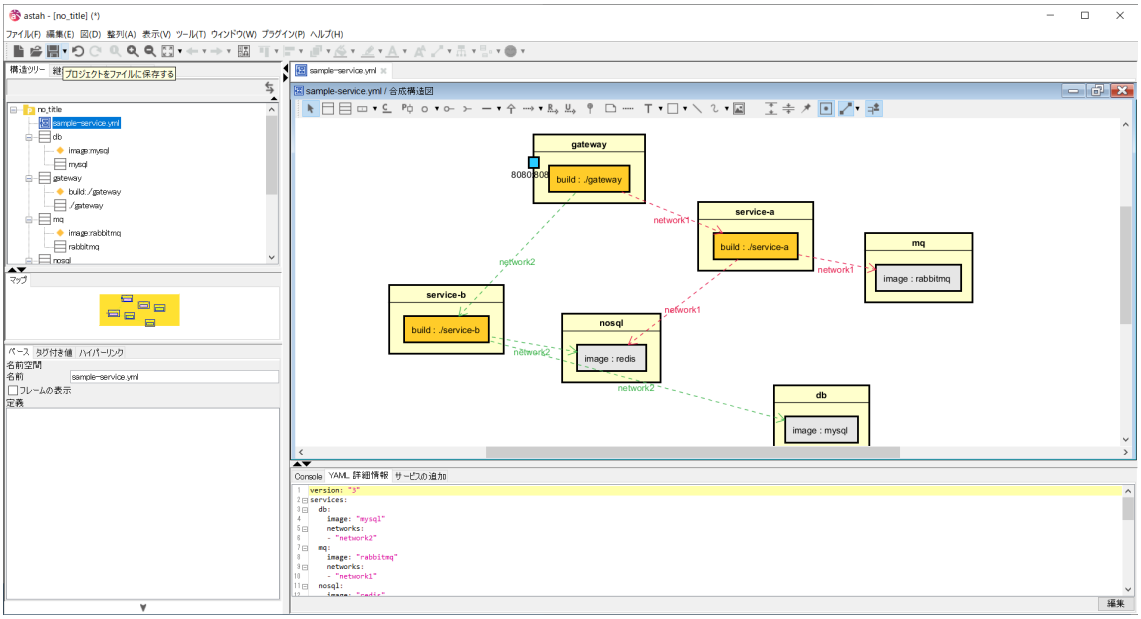

図 9: ソースコード 9をリバースした結果

上位のコンテナ（gateway）から扇円が広がり，下位の基盤となるコンテナ（db，mq，nosql）が扇円の外周に配置されている．また，network1 は左側，network2 は右側の縦ブロックに集中した配置となっている．

5.1.2 UML から Docker Compose YAML を生成するコンポーザー

コンポーザーでは，現在 astah に表示されている UML 図を YAML に変換し，ユーザー指定のファイルに保存する．リバースの時に，YAML テキストを Java オブジェクトに変換したが，元のロー YAML は String の形式でオブジェクトの属性に格納されている．また，編集時もこの属性が更新されるので，コンポーズする際はこの属性にあるテキストがそのままエックスポートされる．検査などを含めるとコンポーズのフローを図 10に示す．

コンポーザーを実装したプラグインで図 11の UML をコンポーズした結果がソースコード 10である．

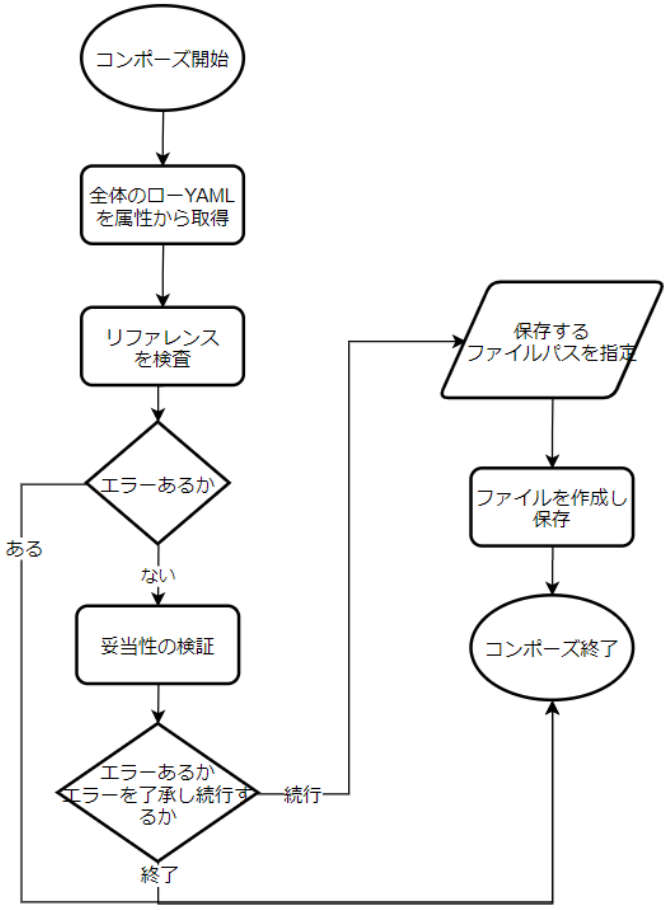

図 10: コンポーズのフロー

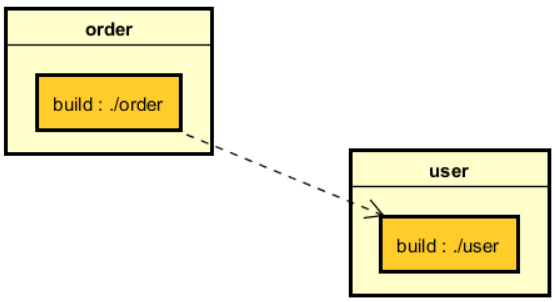

図 11: astah で作成したマルチコンテナ環境を示す UML 図

ソースコード 10: 図 11をコンポーズした結果

```
version: "3.7"
services:
  vote:
    build: "./vote"
    depends_on:
    - "user"
  user:
    build: "./user"
```

5.2 設定情報の編集機能と UI の実装

5.2.1 UML 編集機能の実装

astah の拡張ビューにある YAML エディターでサービスまたは全体についての編集が可能である．拡張ビューで編集を適応した時，現在 Java オブジェクトに格納されているローな YAML テキストを Jackson（JSON や YAML を扱う Java ライブラリ）で直接編集を加え，変更箇所を引用している部分の編集も行う．更新後の YAML について検査検証をしたのち，再リバースが行われ，図にリペイントされる．拡張ビューからサービスまたは全体の編集をした際のフローを図 12に示す．

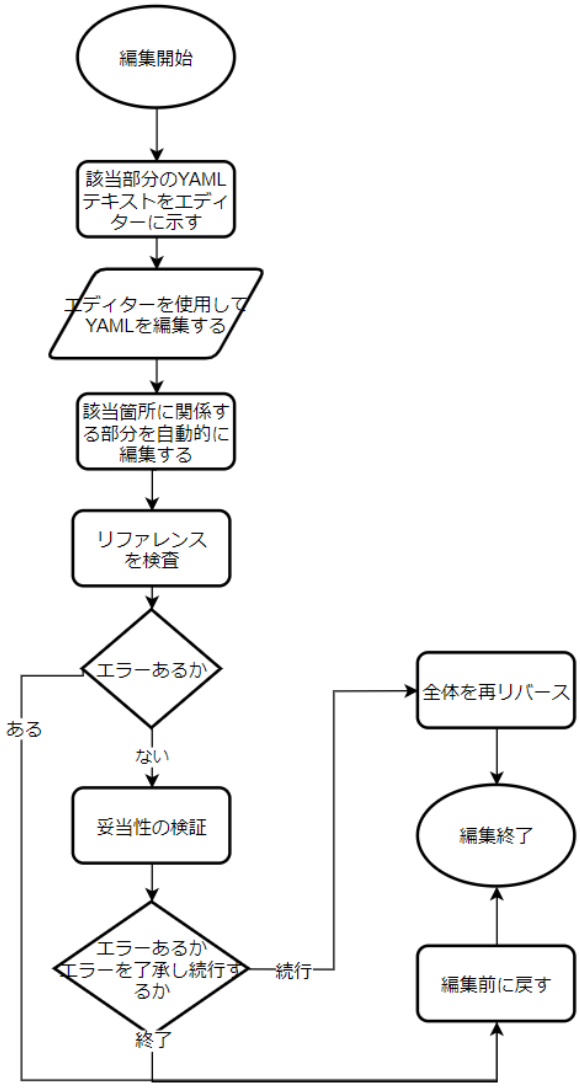
編集開始
該当部分のYAML
テキストをエディ
ターに示す
エディターを使用して
YAMLを編集する
該当箇所に関係す
る部分を自動的に
編集する
リファレンス
を検査
エラーあるか
ある
ない
妥当性の検証
エラーあるか
エラーを了承し続行す
るか
続行
終了
全体を再リバース
編集終了
編集前に戻す

図 12: 編集のフロー

5.2.2 YAML 編集で使用する UI の実装

UI の持つ機能は YAML の編集，サービスの削除，サービスの新規の 3 つあり，Java の Swing で開発し，astah の拡張ビューに実装した．図 13の “YAML の詳細情報” のタブでは該当部分の YAML の編集とサービスの削除ができる．サービスのリネームや削除を行った場合，このサービスに依存している他のコンテナの該当箇所も自動的に編集される．

図 13: 拡張ビューで表示される YAML の詳細情報タブ

“サービスの追加” のタブでは，図 14のようにデフォルトでプラグインに内蔵されているテンプレートが表示される．ここに新規サービスのリファレンスについて記述するし，作成することで，全体に追加することができる．

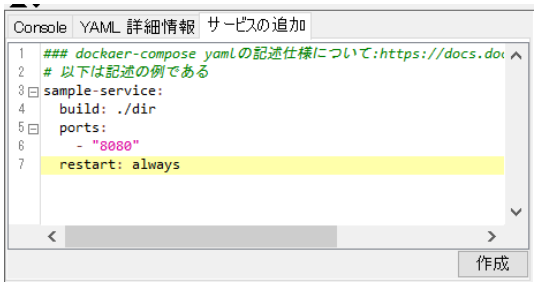

図 14: 拡張ビューで表示されるサービスの追加タブ

一般的には，エディターを使わない，UML 図上で直接の編集方式が考えられるが，astah の API においてリスナーの機能が不十分であるため，編集を適応するタイミングを定義することができない．また，プラグインで規定した UML 図の書き方に従う必要があるため，学習コストも生じる．なので，実装では UML で直接の編集を禁止し，拡張ビューに設置したエディターからの編集に限定している．UML で編集しようとした場合には，編集操作が取り消される仕組みとなっている．

5.3 成果物の正確性を高めるための検査及び検証システム

5.3.1 リファレンス検査の実装

リファレンス検査プログラムは Docker Compose ソースコードの一部，特に $docker-compose config でファイル位置を特定する部分を改造し，実装している．Docker Compose のソースコードは Python 言語で書かれているため，Jython（Python コードを実行できる Java ライブラリー）で実装を行った．実装では，Docker Compose と関係するライブラリーの Python コードをプラグインに内蔵し，Jython で呼び出すことで検査を行う．

初回の検査を行う際は，Jython の起動と該当ライブラリーのインポートが必要である．セットアップの過程をソースコード 11に示す．

ソースコード 11: リファレンス検査機能セットアップ

```
1  private void setupPyInterp() throws IOException {
2      String jythonJarPath = getJythonJarPath();
3      pyInterp = buildInterpEnv(jythonJarPath);
4
5      pyInterp.exec("import sys");
6      pyInterp.exec("sys.path.append('__pyclasspath__/Lib')");
7      pyInterp.exec("sys.path.append('__pyclasspath__/Lib/site-packages')");
8    //該当ライブラリーをインポートする
9      pyInterp.exec("from compose.cli.main import main");
10 }
```

リファレンス検査の実行は，$docker-compose config と同じにコマンドオプションを設定して行う．実行時のログは Java で取得し検査の結果として astah に表す．検査を実行する過程をソースコード 12に示す．

ソースコード 12: リファレンス検査を実行

```
1  public void check(File file) throws ComposeCheckException {
2      String filePath = file.getPath();
3      pyInterp.exec(
4          "sys.argv = ['docker-compose','-f','"
5          + filePath + "','config','-q']\n" +
6          "main()");
7      String checkResult = new String(errOut.toByteArray(), StandardCharsets.UTF_8);
8      //以下省略
9  }
```

5.3.2 妥当性検証機能の実装

4.3節で提案した検証項目すべてを Java で実装を行った．孤立しているサービスの検証を行う部分をソースコード 13に示す．

ソースコード 13: 孤立しているサービスの検証

```
1  public void checkServiceIsolation(Compose compose) throws ComposeCheckException {
2          List<Service> services = compose.getServices();
3          Set<Service> isolated =
4              compose.getServices().stream().collect(Collectors.toSet());//copy
5      //isolatedに孤立のサービスを格納
6          for (Service service : services) {
7              Set<Service> depends = service.getDependsOrLinksService().keySet();
8              if (depends.size() > 0) {
9                  isolated.remove(service);
10             }
11             depends.forEach(d -> isolated.remove(d));
12         }
13         if (isolated.size() > 0) {
14             //一部省略
15             //孤立したサービスがある場合ワーニングする
16             throw new ComposeCheckException(Type.VALIDATION, Level.WARNING,
17                     "サービス(" + sb.toString() + ") が依存関係を持たない孤立したサービスです.");
18         }
19     }
```

リバースやコンポーズ時にリファレンス検査をしたのち妥当性の検証を行うが，妥当性検証ではエラーやワーニングが発生している場合でも，ユーザーの了承を得たうえでの操作続行は可能である．だが，リファレンス検査でエラーが発生する場合，続行は不可能であるので，現在の操作は取り消される実装となっている．

クラス ComposeValidationChecher のうち，検証項目それぞれに対応する実装メソッド名を表 2に示す．

メソッド名	検証項目	レベル
checkServiceIsolation()	サービスが孤立している	ワーニング
checkServiceDependsLoop()	サービス間の依存がループになっている	
checkDuplicatedServiceName()	同じ名前のサービスが存在する	エラー
checkDuplicatedServicePorts()	他のサービスとポートが被っている	
checkUndifinedNetwork()	既存でないネットワークに依存している	

表 2: 検証項目を実装した ComposeValidationChecker のメソッド

5.4 評価

5.4.1 Docker Compose YAML と UML の互換機能の評価

互換機能を評価するために，Github などから Docker Compsoe を使用しているレポジトリを探し，実際に用いられる 20 個以上の docker-compose.yml の例を用いて互換を試みた．使用した実例を表 3に示す．

Github レポジトリまたは URL	リファレンスバージョン	結果
ory/examples	2	失敗
vovimayhem/docker-compose-rails-dev-example	2.1	
chriszarate/docker-compose-wordpress	3	
cypress-io/cypress-example-docker-compose	3	成功
DataDog/docker-compose-example	3	
https://docs.docker.com/compose/django/	3	
samrocketman/docker-compose-ha-consul-vault-ui	2.2	
tarunlalwani/docker-compose-mysql-master-slave	2	
dockersamples/example-voting-app に含む 3 つの実例	3	
ewolff/microservic	3	
neumayer/mysql-docker-compose-examples	3	
juggernaut/nginx-flask-postgres-docker-compose-example	3	
PagerTree/prometheus-grafana-alertmanager-example	3.1	
https://docs.docker.com/compose/rails/	3	
dmitrym0/simple-lets-encrypt-docker-compose-sample/	2	
JetBrains/teamcity-docker-samples	3	
wmde/wikibase-docker	3	
https://docs.docker.com/compose/wordpress/	3.3	

表 3: 合成構造図モデルとリファレンスの対応付け

プラグインの機能を確かめるために古いバージョンのものから最新の 3.7 バージョンのものまでファイルについて互換を行ったが，一部失敗する実例も見られた．

原因の一つは，エイリアスに対応する機能の不足のためである．links や networks などのリファレンスでサービスやネットワークの別名が指定できるようになっており，エイリアスでコンテナにアクセスすることが可能である．しかし，本プラグインでエイリアスに関する機能の実装は行われていないので，互換をする際指定のサービスを特定できず，エラーが発生する．

5.4.2 検査検証機能の評価

リファレンスの検査について，本研究では新たに検査システムを開発するのでなく，既存のソースコードを動かすことで実装を行った．このため，新たなリファレンスバージョンがリリースされたとき開発コストがほぼなく，Docker Compose で動作できる YAML であることを保証できる．だが，$ docker-compsoe config の部分のみでなく，Docker Compsoe のソースコード全体をプラグインに内蔵しているため，プラグインのビルドや初めて検査を行うセットアップは時間がかかる．CPU Intel Core i5-8350U，RAM 8GB のマシンで実験したところ，セットアップは 10 秒ほど要しが，実際の検査実行は 20ms 300ms のみであった．

5.4.3 ユーザービリティーに対する評価

UML を取り入れたことで，YAML を扱うより断然に可読性が向上した．しかし，UML 図からの編集削除を禁止としたため，UML で直接行える操作は配置位置を変えるためのドラッグアンドドロップだけに限定してしまった．その代わり，編集操作が安全になったが，普段から UML 図を使い慣れている設計者からすると，ユーザービリティーは下がったと言える．

6　まとめと今後の課題

6.1　まとめ

本研究で Docker Compsoe の設計効率及び正確性を向上させるため，docker-compose.yml と UML 図の互換機能を実装し，UML 図で YAML を扱うことで，コンテナ間の依存関係が理解しやすくなった．また，YAML エディターの UI を実装したことで，astah のみで外部に頼らない設計手法が可能になった．さらに，起動時に行われるリファレンスの検査を UML の設計時に動的に行うことで，エラーを早い段階で発見することができ，妥当性検証の機能を付け加えたことで，マルチコンテナの設計の正確性を高めた．

6.2　今後の課題

6.2.1　リファレンス検査の軽量化

実装でリファレンス検査のため Docker Compsoe に関係する Python コードすべてををプラグインに組み入れているため，依存ライブリーをふくめるとトータルで 30 以上のライブラリーを内蔵していることとなる．このため，ビルトで生成されるプラグインの jar ファイルは６０ MB にもなり，ビルド時間が 1 分以上かかる（CPU Intel Core i5-8350U，RAM 8GB）．しかし，リファレンス検査に使用する $ docker-compsoe config はテキストの確認のみを行う部分であるので，該当の Python コードは複雑ではない．なので，ユーザービリティとプラグイン運用のため，該当部分のソースコードを Docker Compose 全体から隔離し，軽量化する必要がある．

6.2.2　妥当性検証の強化

本研究では，4.3節の 7 つの検証項目について実装した．しかし，ほかにも多くの検証項目が存在する．例えば，image リファレンスで指定しているイメージ名は実際にレポジトリで存在するイメージか，オフィシャルのイメージであるか，認証されているイメージであるかを確かめることで，イメージの安全性を確かめ，リスクを減らすことができることができる．より正確性を向上させるためにはさらなる検証項目の考察と実装が必要である．

参考文献

[1] kotaro-dr:【図 解】Docker の 全 体 像 を 理 解 す る （https://qiita.com/kotaro-dr/items/b1024c7d200a75b992fc）

[2] 株 式 会 社 チ ェ ン ジ ビ ジ ョ ン: astah*professional（http://astah.change-vision.com/ja/product/astah-professional.html）

[3] Docker:（https://docs.docker.com/compose/compose-file/）

[4] 畑瀬尚之，和﨑克己（2019）: UML 上位設計からの自動コード生成を対象とした整合性検査; 2019 年度電子情報通信学会信州大学 Student Branch 論文発表会講演論文集，（A-1）

[5] IBM: コンポジット構造図(https://www.ibm.com/support/knowledgecenter/ja/SS5JSH_9.1.2/com.ibm.xtools.modeler.doc/topics/ccompstruc.html）

[6] Jython: （https://www.jython.org/）

謝辞

本論文を執筆するにあたって，熱心にご指導いただいた信州大学工学部和崎克己教授には，終始熱心な御指導，御教示を賜った．心より感謝申し上げる．

付録

5.4.1節で実例評価のために行った Docker Compose ファイルと UML の互換結果の一部を以下に示す.

DataDog/docker-compose-example

Github レポジトリ DataDog/docker-compose-example にある docker-compose.yml（ソースコード 14）をプラグインで互換した結果が図 15である.

ソースコード 14: Github レポジトリ DataDog/docker-compose-example の実例

```
version: "3"
services:
  web:
    build: web
    command: python app.py
    ports:
     - "5000:5000"
    volumes:
     - ./web:/code # modified here to take into account the new app path
    links:
     - redis
    environment:
     - DATADOG_HOST=datadog # used by the web app to initialize the Datadog library
  redis:
    image: redis
  # agent section
  datadog:
    build: datadog
    links:
     - redis # ensures that redis is a host that the container can find
     - web # ensures that the web app can send metrics
    environment:
     - DD_API_KEY=__your_datadog_api_key_here__
     - DD_DOGSTATSD_NON_LOCAL_TRAFFIC=true
    volumes:
     - /var/run/docker.sock:/var/run/docker.sock
     - /proc/:/host/proc/:ro
     - /sys/fs/cgroup:/host/sys/fs/cgroup:ro
```

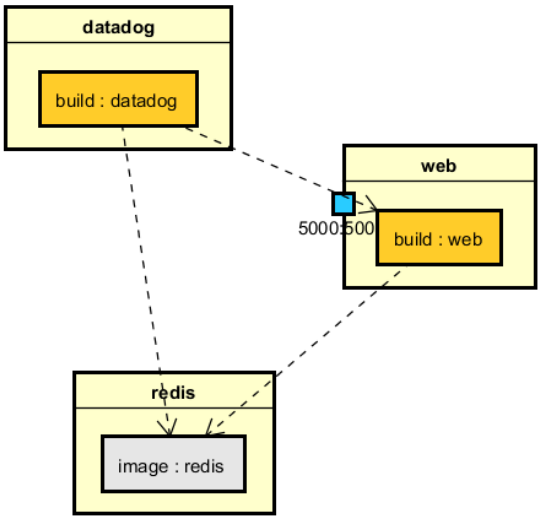

図 15: ソースコード 14をプラグインでリバースした結果

dockersamples/example-voting-app

Github レポジトリ dockersamples/example-voting-app にある docker-compose.yml（ソースコード 15）をプラグインで互換した結果が図 16である.

ソースコード 15: Github レポジトリ dockersamples/example-voting-app の実例

```
#example-voting-app
version: "3"

services:
  vote:
    build: ./vote
    command: python app.py
    volumes:
     - ./vote:/app
    ports:
      - "5000:80"
    networks:
      - front-tier
      - back-tier

  result:
    build: ./result
    command: nodemon server.js
    volumes:
      - ./result:/app
    ports:
      - "5001:80"
      - "5858:5858"
    networks:
      - front-tier
      - back-tier

  worker:
    build:
      context: ./worker
    depends_on:
      - "redis"
      - "db"
    networks:
      - back-tier

  redis:
    image: redis:alpine
    container_name: redis
    ports: ["6379"]
    networks:
      - back-tier

  db:
    image: postgres:9.4
    container_name: db
    volumes:
      - "db-data:/var/lib/postgresql/data"
    networks:
      - back-tier

```

```
52 volumes:
53   db-data:
54
55 networks:
56   front-tier:
57   back-tier:
```

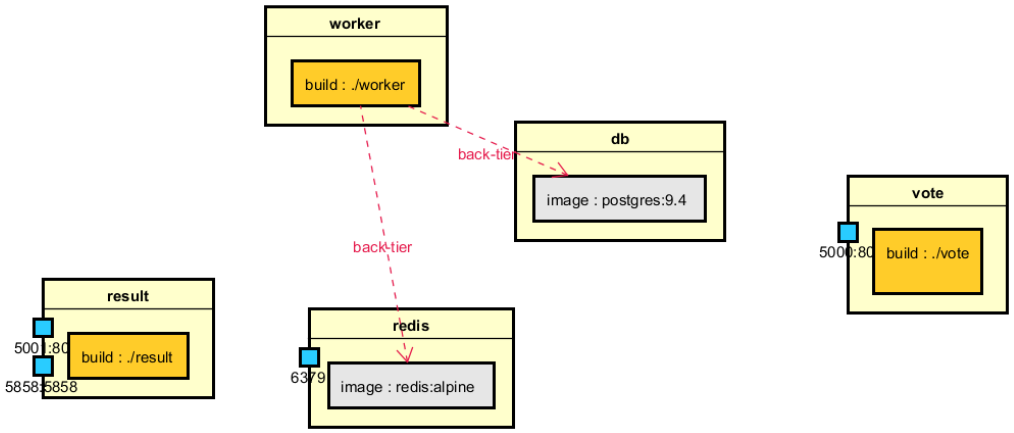

図 16: ソースコード 15をプラグインでリバースした結果

ewolff/microservice

Github レポジトリ ewolff/microservice にある docker-compose.yml（ソースコード 16）をプラグインで互換した結果が図 17である．

ソースコード 16: Github レポジトリ ewolff/microservice の実例

```
#https://github.com/ewolff/microservice/blob/master/docker/docker-compose.yml
version: '3'
services:
  eureka:
    build: ../microservice-demo/microservice-demo-eureka-server
    ports:
      - "8761:8761"
  customer:
    build: ../microservice-demo/microservice-demo-customer
    links:
     - eureka
  catalog:
    build: ../microservice-demo/microservice-demo-catalog
    links:
     - eureka
  order:
    build: ../microservice-demo/microservice-demo-order
    links:
     - eureka
  zuul:
    build: ../microservice-demo/microservice-demo-zuul-server
    links:
     - eureka
    ports:
      - "8080:8080"
  turbine:
    build: ../microservice-demo/microservice-demo-turbine-server
    links:
     - eureka
    ports:
      - "8989:8989"
```

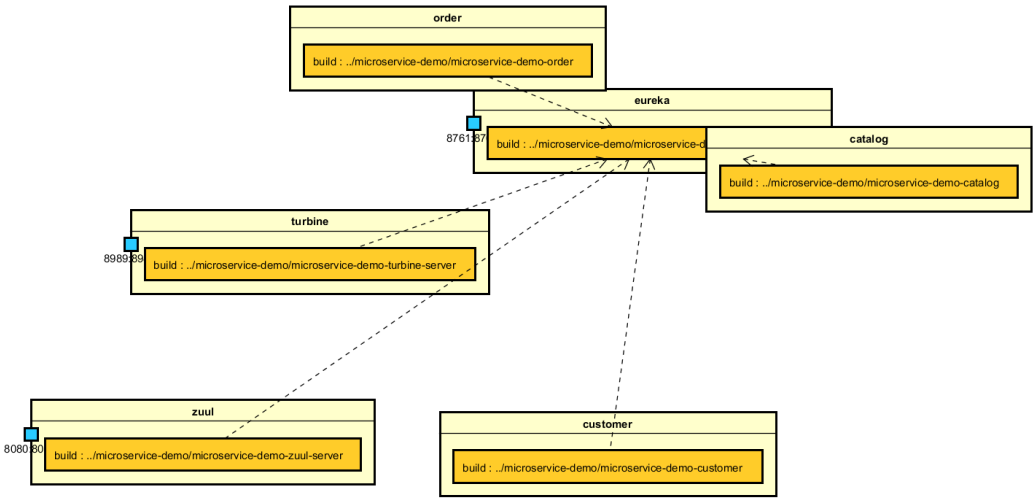

図 17: ソースコード 16をプラグインでリバースした結果

neumayer/mysql-docker-compose-examples

Github レポジトリ neumayer/mysql-docker-compose-examples にある docker-compose.yml（ソースコード 17）をプラグインで互換した結果が図 18である．

ソースコード 17: Github レポジトリ neumayer/mysql-docker-compose-examples の実例

```
#https://github.com/neumayer/mysql-docker-compose-examples/blob/master/mysql-server/docker-compose.yml
version: '3'
services:
  mysql-server-56:
    image: mysql/mysql-server:5.6
    volumes:
        - ./docker-entrypoint-initdb.d/:/docker-entrypoint-initdb.d/
    ports:
      - "3306:3306"
  mysql-server-57:
    image: mysql/mysql-server:5.7
    volumes:
        - ./docker-entrypoint-initdb.d/:/docker-entrypoint-initdb.d/
    ports:
      - "3307:3306"
  mysql-server-80:
    image: mysql/mysql-server:8.0
    volumes:
        - ./docker-entrypoint-initdb.d/:/docker-entrypoint-initdb.d/
    ports:
      - "3308:3306"
    command: ["mysqld", "--default-authentication-plugin=mysql_native_password"]
  dbwebapp-56:
    env_file:
      - dbwebapp.env
    environment:
      - DBHOST=mysql-server-56
    image: neumayer/dbwebapp
    ports:
      - "8056:8080"
    depends_on:
      - mysql-server-56
  dbwebapp-57:
    env_file:
      - dbwebapp.env
    environment:
      - DBHOST=mysql-server-57
    image: neumayer/dbwebapp
    ports:
      - "8057:8080"
    depends_on:
      - mysql-server-57
  dbwebapp-80:
    env_file:
      - dbwebapp.env
    environment:
      - DBHOST=mysql-server-80
    image: neumayer/dbwebapp
    ports:
      - "8080:8080"
```

```
51      depends_on:
52        - mysql-server-80
```

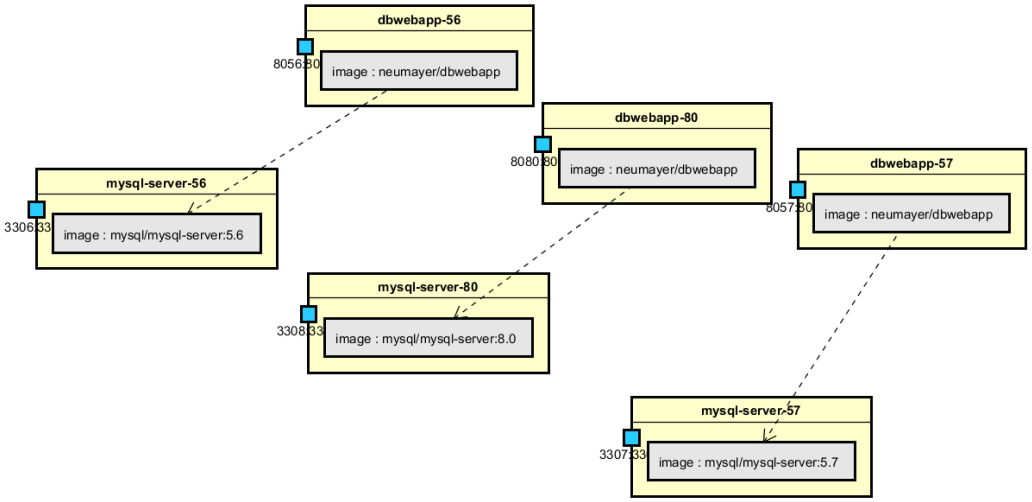

図 18: ソースコード 17をプラグインでリバースした結果