Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Automated Detection and Extraction of Deployment Components of Docker Services

Marcel Zeller

Course of Study: Softwaretechnik

Examiner: Prof. Dr. Dr. h. c. Frank Leymann

Supervisor: Lukas Harzenetter, M.Sc.

Commenced: November 9, 2018

Completed: May 9, 2019

Abstract

Validation of deployability of application topologies at design-time is a difficult task. Often application topologies can not be checked until the complete system is deployed. A reliable and up-to-date data source that could be used for validation is a big problem. A possible solution can be the use of the build-scripts of Docker images: *Dockerfiles*. They provide application component relationships in a well-defined syntax.

To be able to extract reliable software component relationship data from dockerfiles a big source set is necessary. Online software repository services provide many public available dockerfiles. Crawlers can be used to collect them. Afterwards the dockerfiles need to be analyzed. Ambiguous, misleading or incorrect content of dockerfiles must be taken into account as far as possible to ensure reliable results.

This thesis outlines a concept for collection of dockerfiles, discovery and extraction of deployment components and understandable depiction of the results. The collection is done with crawlers for specific online services. For the discovery and extraction analyzer for component installation commands are provided. A metamodel for models of component structures ensures the comprehensibility of the results. The concept is implemented prototypically with the support of one online software repository service and several wide-spread installation command-line tools.

Contents

1 Introduction **17**
1.1 Problem Statement 17
1.2 Main Goal 17
1.3 Content Outline 18

2 Background **19**
2.1 Docker 19
2.2 Package-Manager 20

3 Related Work **21**
3.1 Topology Models 21
3.2 Automated Discovery of Topologies 24
3.3 Crawler 32
3.4 Model-Driven Docker Management 37

4 Considerations for an Approach for Automated Discovery Based on Dockerfiles **41**
4.1 Requirements 41
4.2 Overall Algorithm 41
4.3 Crawler 42
4.4 Analyzer 48
4.5 Model 54

5 An Approach for Automated Discovery Based on Dockerfiles **57**
5.1 Crawler 57
5.2 Analyzer 61
5.3 Model 66
5.4 Algorithm 66

6 Prototype and Evaluation **71**
6.1 Extension to Eclipse Winery 71
6.2 Evaluation 71

7 Conclusion and Future Work **77**

Bibliography **79**

A Appendix **83**

List of Figures

3.1 Application template class by Machiraju et al. [MDW+00] 22
3.2 Aeolus example based on [CZZ12] 23
3.3 Metamodel of the ETG-approach by Binz et al. [BFL+12] 24
3.4 Generic auto-discovery with variant parameters which can be filled by different variants (“what to discover” and “how to discover”) 26
3.5 Specific auto-discovery with predefined variants 27
3.6 Example application overview model by [MDW+00] 27
3.7 Step 1 of Orion approach [CZMB08]: Two packets are combined to a flow in each case 30
3.8 Step 2 and 3 of Orion approach [CZMB08] 31
3.9 Meta-model of DLT-models by [MDJV08] 32
3.10 Architecture of DockerFinder based on [BNS17] 35
3.11 Crowd crawling architecture by Ding et al. [DCF13] 37
3.12 Architecture of the MDE-approach by Paraiso et al. [PCAM16] 38
3.13 Metamodel of the MDE-approach by Paraiso et al. [PCAM16] 40

4.1 Steps of the component discovery approach 42
4.2 Activity diagram of a rate limit counter solution 44
4.3 Activity diagram of a rate limit discontinuation time solution 45
4.4 Activity diagram of a rate limit open window solution 46
4.5 Metamodel of this approach’s model 54
4.6 An example model of this approach 56

5.1 Split crawler functionality to comply with rate limits 58
5.2 Sequence diagram of the GitHub crawler implementation 60
5.3 Activity diagram of the analyzer implementation 62
5.4 Activity diagram of the Pre-Analyzer part 63
5.5 Docker image layer associated to the corresponding dockerfile instructions 64
5.6 Activity diagram of the algorithm as depicted in Algorithm 5.1 69

6.1 Number of searched repositories of all seven test executions 73
6.2 Results of the evaluation of installation commands 75

List of Tables

4.1 Overview over docker instructions 49

6.1 Numbers of a crawler execution test 73
6.2 Calculations for a complete crawl of GitHub 74

A.1 Test execution results 83

List of Listings

2.1 Simple dockerfile 19
2.2 Different docker RUN instructions 20
4.1 Dockerfile and SQL FROM instructions 47
4.2 Syntax overview for *npm install* [npm19] 50
4.3 Example of incorrect command syntax 50
4.4 Potential false-positives for a heuristic 51
4.5 Potential false-negatives for a heuristic 51
4.6 Multi-Stage Build example 53
4.7 Comment examples in a dockerfile 53
4.8 Examples of URLs in *npm install* commands 54
5.1 Two examples of multi-line commands 61
5.2 Syntax variants of FROM instruction [Doc19b] 63
5.3 ARG variable definition with and without default value 65

List of Algorithms

3.1 Basic steps of a crawler [HN99] 33
4.1 Basic algorithm steps of this approach 42
5.1 Algorithm of this approach 68

List of Abbreviations

API Application Programming Interface. 27

ARIS Architektur Integrierter Informationssysteme. 22

ATM Automatic Teller Machine. 25

DLT Data Locations Template. 30

DNS Domain Name System. 27

EA Enterprise Architecture. 21

ETG Enterprise Topology Graph. 24

ID Identifier. 74

IP Internet Protocol. 25

MDE Model-Driven Engineering. 38

OSI Open Systems Interconnection Model. 29

PID Process Identifier. 27

RCM Result Collection Module. 36

REST Representational State Transfer. 58

SQL Structured Query Language. 19

TAM Task Assignment Module. 36

TCP Transport Control Protocol. 29

TOSCA Topology Orchestration Specification for Cloud Applications. 55

UDP User Datagram Protocol. 29

UID Unique Identifier. 36

UML Unified Modeling Language. 21

URL Uniform Resource Locator. 33

1 Introduction

Modern application systems and IT infrastructure consist of compositions of several software components. Many applications rely on other applications and form an application stack. If one part of the stack is missing or has the wrong version, all other parts building on this part fail to operate correctly. Validation of deployability of an application stack while design-time is difficult, because reliable data of deployable application relationships is missing. A manually created overview over deplyoable relationships would be outdated very quickly. Often the only possibility to check deployability of an application stack is to try it and check if it works.

The build-scripts for images of containers of the container technology *Docker* provide a standardized syntax to build an application stack. Images of containers build on each other and every image can contain several software components. This defines relationships between the software components of different container images. These relationships can be automatically analyzed, because the syntax of the build-scripts of Docker images (*dockerfiles*) have a well-defined and known syntax.

In Section 1.1 the problem tackled by this work is outlined. Section 1.2 depicts the main goal of this work. Section 1.3 gives an overview over the complete thesis.

1.1 Problem Statement

The arrangement of several components is known as deployment model. While the development process of a deployment model it would be helpful to know, which components run together or run based on each other. It is necessary for that to building up knowledge. One possibility is to try doubtful combinations of components out. This would be very costly and time consuming. Another possibility is to evaluate exising dockerfiles and extract their component relationships. If the number of evaluated dockerfiles is big enough, it is very likely that repeatedly found relationships of components are actually runnable.

1.2 Main Goal

To get an overview over deployable components a methodology is necessary. There are several requirements the methodology have to fulfill. First, the complete evaluation process and extraction process have to be automatic. Due to fast changes and trends in the IT sector an overview gets outdated quite fast. Therefore, it has to be possible to update it often. Manual processes and high effort prevent regular updates. Second, the results need to be depicted in an understandable and documented model to use it for further tasks. Third, the data source of dockerfiles needs to be big. Open source software platforms maintain large number of repositories with dockerfiles, they have to be used.

1.3 Content Outline

This thesis is structured as follows. Chapter 2 gives a short introduction into important technologies. Chapter 3 introduces into the state of the literature. Automated topology discovery approaches, topology models and crawler approaches are depicted. Chapter 4 outlines considerations for the concept of the approach for discovery and extraction of software components from dockerfiles. It establishes requirements, gives a short overview over the necessary steps, outlines challenges of a crawler and an analyzer and depicts the metamodel for the approach. Chapter 5 depicts the concept details, solutions for the crawler and analyzer and outlines the algorithm of the approach. Chapter 6 shows implementation details and evaluates the implementation. The work is summed up in Chapter 7 and an outlook is given.

2 Background

This chapter introduces technologies used in this work. They are necessary to understand the concept and implementation of this approach. First the container technology *Docker* is introduced (Section 2.1). Then a short introduction to *Package-managers* is given (Section 2.2).

2.1 Docker

Docker [Doc19a] is a container technology used for virtualization. It is a more lightweight virtual machine technology. It splits up application stacks into small layers which allows not only the reuse of layers in a running environment, but also the reuse of these small so-called images in the building process of a new application [Tur15]. Every software component installation command to a docker container forms a new layer. But not every layer forms a new image. Several layers are joined together to an image. An image depends on another image, for example, a Java-image depends on a operating system image like an Ubuntu-image.

Dockerfiles are text documents that contain different build instructions each followed by parameters [Doc19b]. The syntax slightly reminds of Structured Query Language (SQL). Main parts are the "FROM" part which determines the base image, the "RUN" part which executes new layer on top of the base image and the "CMD" part which provides defaults for the executing container[1]. A dockerfile is the build-script for a docker image. As every image is based on another image, a base image needs to be defined in the dockerfile as shown in line one of Listing 2.1. In this example *ubuntu 16.04* is used as base image for the new docker image. To end the chain of images, docker provides base images, for example, images of operating systems like Ubuntu.

Listing 2.2 shows some examples of docker layers. All lines are examples of dockerfile instructions. Line one and three are each one layer, each consisting out of one software component. Line five is one layer, too. It consists out of four software components. Line seven is composed of several commands. Nevertheless it is one docker layer, because all commands are assembled by '&&'. Lines nine to eleven form three docker layers, which build on each other, because every command is an own RUN instruction.

Listing 2.1 Simple dockerfile

```
1 FROM ubtunu:16.04 AS ubuntu
2 CMD abcService.sh
3
```

[1]A complete overview over the syntax of a docker-file can be found at the docker reference docs [Doc19b].

Listing 2.2 Different docker RUN instructions

```
RUN apt-get install libsm6

RUN apt-get update -qq -y

RUN apt-get install libsm6 libxrender1 libxext-dev python3-tk

RUN apt-get update -qq -y && apt-get install -y libsm6 libxrender1 && apt-get clean

RUN apt-get update -qq -y
RUN apt-get install -y libsm6 libxrender1
RUN apt-get clean

```

2.2 Package-Manager

A package-manager or package management software is a software which allows installing, upgrading and removing of other software packets, libraries, command-line tools and other software. A package-manager has sub-commands for different tasks like installation. Examples of package-managers are *apt-get*, *yum* and *apk*. To install a software with apt-get the following example command can be used: *apt-get install softwareName*. A software packet is also called software component in this work.

3 Related Work

This chapter gives an overview over the literature about topology models (Section 3.1), automated topology discovery (Section 3.2) and crawler (Section 3.3). These topics are related with the concept and implementation of the software component discovery approach of this work. In the end of this chapter an approach for model-driven analyzes of Docker is depicted (Section 3.4).

3.1 Topology Models

Modeling of systems provides many opportunities [Béz05]. Models can be used to depict complex systems in an understandable way. Topology models depict all components of an application environment as well as the relations between the applications. As base for a topology model standardized modeling languages like Unified Modeling Language (UML) [Obj19] can be used. Section 3.1.2 outlines several approaches for topology models. Topology models display a certain view on the depicted application environment. Section 3.1.1 gives a small overview over model views and concludes with the necessary view for topology models.

3.1.1 Model View

A main challenge in the field of modeling is the selection of the correct view of a model [Sch11]. In the field of enterprise topologies and architecture different views are present [The19]. An early approach for Enterprise Architecture (EA) management is published by Zachman [Zac99]. The approach creates a framework to model IT systems. The approach intends five views for an enterprise system: scope, business, logical systems, technical systems and detailed representation. Each view provides different information for different use cases of different stakeholders of the system:

- **Scope view**: Strategists can plan strategic decisions behind the system with a model of this view
- **Business concept view**: Provides an overview over the business logic for executive leaders
- **Logical system view**: System architects can use models of this view to design a system
- **Technical system view**: Engineers can build the infrastructure with this view's model
- **Detailed component view**: Technicians and developers can implement the system with models of this view

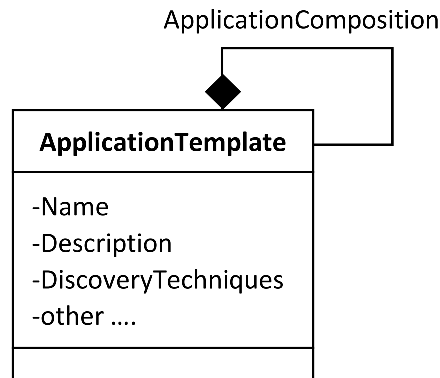

Figure 3.1: Application template class by Machiraju et al. [MDW+00]

Scheer [Sch13], Klar [Kla99] and Keller et al. [KNS92] focus on data flows and the structure of the system instead on stakeholders, as their common approach *Architektur Integrierter Informationssysteme (ARIS) (architecture of integrated information systems)* is a process-oriented approach. It consists of five views:

- **Organization view**: describes the structure of the system
- **Data view**: provides an overview over the business data objects
- **Function view**: describes the business functions
- **Output view**: describes the results of the business functions
- **Control view**: assemble the other views as processes

Both approaches suffer from limited usability as they have predefined target groups [Zac99] [Sch13]. To provide models with an overall view for undefined users The Open Group define four architecture domains as subsets of the overall Enterprise Architecture [The19]: business architecture, data architecture, application architecture and technology architecture. Dependent on the topic which an approach wants to work on and improve another domain and therefore model view is interesting. For application topology analysis the application architecture domain is the important domain.

3.1.2 Application Topology Models

An approach for application topology models was published by Machiraju et al. [MDW+00]. Their approach includes a simple application template class with basic properties which is shown in Figure 3.1. This class is the base to build more complex *application template models* with several applications in accordance with the structure of the analyzed system.

An application template model from the approach of Machiraju et al. [MDW+00] has to depict the expected applications of the analyzed system. It represents the limits for a generic auto-discovery as described in Section 3.2. After the discovery process finishes the discovered application instances are mapped to the template model applications. This presupposes an accurately created application template model, which is difficult if not much is known about the system which has to be analyzed.

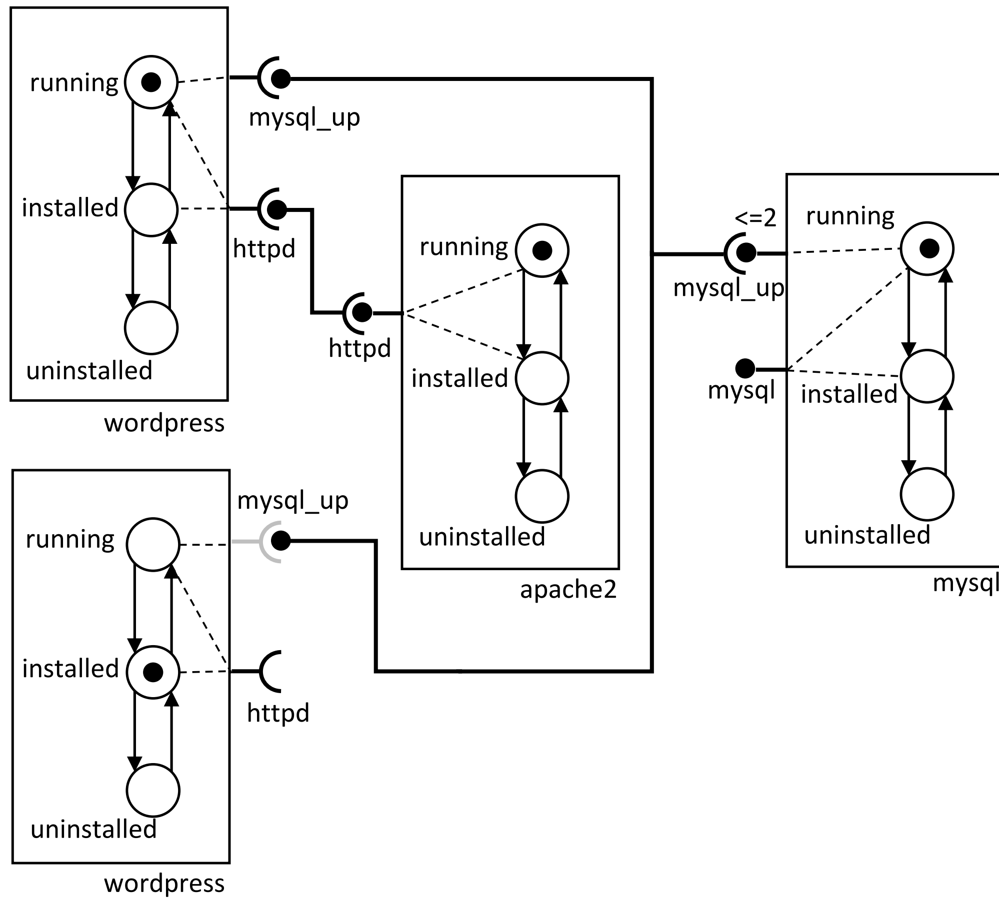

Figure 3.2: Aeolus example based on [CZZ12]

Other approaches do not presuppose any knowledge about the system to be analyzed. Instead metamodels are used which define only the syntactical structure of the resulting models: The approach *Aeolus* from Di Cosmo et al. [CZZ12] includes a component model defined by a metamodel and can be used to model cloud components and their relationships. Each present component is represented by a block, for example, *wordpress* in Figure 3.2. A component can have several states which are represented by circles inside the blocks. States can be, for example, installed, uninstalled and running [CZZ12]. The states have directed transitions. The transitions can be cyclic. A component can have several interfaces which provide or require specific other components. In Figure 3.2 *wordpress* has a requirement interface *httpd*, which requires a suitable interface, in this case provided by the *apache2* component. An interface is connected to one or several states inside the component and can be active or inactive, dependent on the current state of the component. Each interface can have a cardinality together with an operator to indicate how many other components this interface can serve for, for example, up to 2 other components as the *mysql_up* interface of the *mysql* component in Figure 3.2.

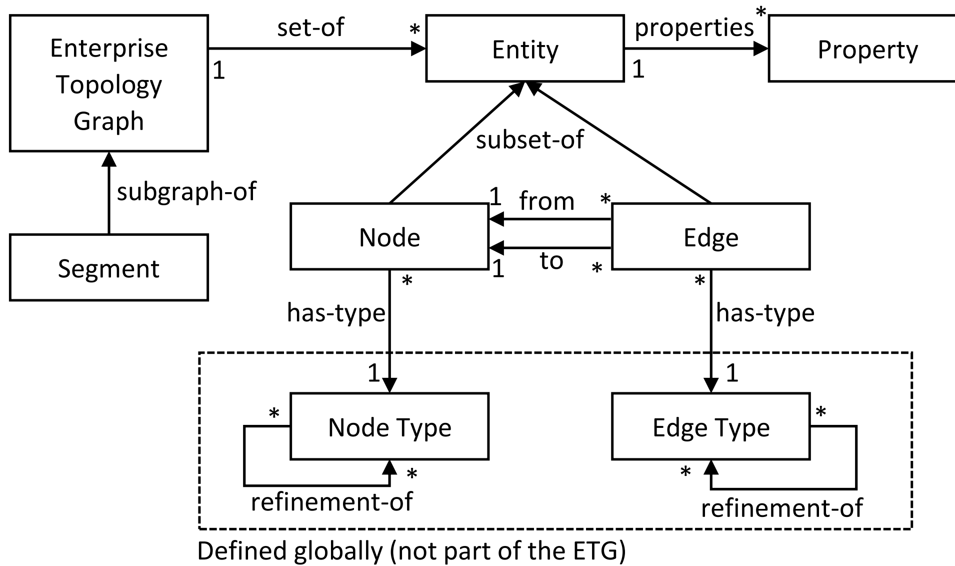

Figure 3.3: Metamodel of the ETG-approach by Binz et al. [BFL+12]

Di Cosmo et al. [CZZ12] build Aeolus for cloud systems. However, it can be used for all component-based systems. In particular it gives an overview for package installation, activation of components, redundancy, capacity and conflict management and for creating and destroying resources [CZZ12].

Binz et al. [BFL+12] published a similar model approach for Enterprise Topology Graphs (ETGs). They focus on the relationships between applications. These relationships can be logical, functional and physical [BFL+12]. Therefore, different abstraction layer can be modeled. The model is defined with a metamodel, shown in Figure 3.3, and can be built without any foreknowledge only based on data discovery by auto-discovery approaches. *Nodes* represent applications, *Edges* represent relationships between them. Both have hierarchical ordered *Types* to ensure taxonomy between different abstraction layer [BFL+12]. For example, all “Windows” and “Linux” nodes in a graph can be generalized to “operating system”, if the type hierarchy provides these types. Nodes as well as edges can have properties. The properties are key-value pairs and show additional domain-specific information. For example, data gathered at runtime like workload or runtime errors can be added.

3.2 Automated Discovery of Topologies

This work is targeting at automated detection of deployable topologies and structures for deployment of applications and services. In the literature different approaches exist which try to discover topologies semi-automatic or fully automatic. They use different information as source for their

discoveries, but mainly they are based on information extracted out of an analyzed running system [CZMB08] [KGE06], because configuration files, deployment scripts and documentations are not available [CZMB08] or need manual analyzes as they are not machine-processable.

The motivation for automated dependency discovery approaches is often the need to improve and optimize a complex system or IT infrastructure, because knowledge of dependencies of human experts does not scale with large enterprise systems [CZMB08]. Manual discovery processes have different disadvantages as Machiraju et al. [MDW+00] note:

- **Time consuming**: Manually discovering all applications and dependencies takes a lot of time.
- **Lack of reuse**: The manual activity needs to be repeated similarly many times.
- **Distributed intelligence**: The expertise is distributed over several people.
- **Inconsistency**: Several people working on the same activity often leads to inconsistent results.

Ensel [Ens99] state that manual processes are very time consuming and regular manual updates on the discovered topology models are not possible. To circumvent the need of regular updates, manually created models tend to be on a high abstraction level to not have to pay attention to small technical changes [BLNS12]. Brown et al. [BKK01] deduce, that only simple and small systems can be discovered by human experts. With growing size and complexity of systems manual processes are no longer useful and should be replaced by automated discovery processes. Breitenbücher et al. [BBKL13], Joukov et al. [JPRD10] and Magoutis et al. [MDM07] further explicate these problems with manual discoveries in the field of enterprise topologies which are examples of system with a large size and high complexity.

3.2.1 From Specific to Generic Automated Discovery Processes

There are different automated discovery approaches available using different data as base for their discovery. In the following the automated discovery approach from Lin et al. [LLC98] [LYL99] is presented which uses specific, predefined data as source for their discovery. Machiraju et al. [MDW+00] outline problems with specific data sources and depict possible solutions.

More than twenty years ago first automatic topology discovery approaches and tools were published [LLC98]. With increasing size of networks and enterprise infrastructures the need for an overview over these systems came up. To serve this need, models based on automatic gathered data were build. One early simple example is specific to a particular system: Lin et al. [LYL99] introduce an automated topology discovery for connected Automatic Teller Machine (ATM) systems. Serving only ATMs as a specific system enables the discovery approach to use specific data stored in these systems. In this case routing tables in interface-configurations are used to find all neighbors of an ATM. The routing tables contain entries with Internet Protocol (IP) addresses of all neighbors connected to the network interface of this ATM. With a recursion algorithm the complete topology of the ATM environment can be automatically discovered [LYL99]. However, this example demonstrates the problem which comes up as soon as the automated specific discovery process should be applied to other environments: Another environment does not have this specific routing table. The automated discovery fails.

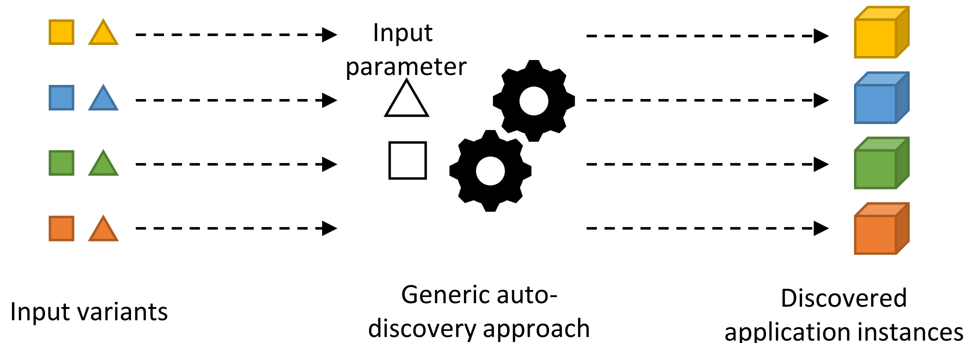

Figure 3.4: Generic auto-discovery with variant parameters which can be filled by different variants ("what to discover" and "how to discover")

Machiraju et al. [MDW+00] outline two major problems with specific automated discovery, like the ATM discovery tool:

- **Multi-application management tools**: The tool needs to be created for every single type of application again.
- **Reuse across applications**: Methods from a tool mostly cannot be reused for another tool.

To build more generic tools other approaches including Machiraju et al. [MDW+00] renounce the use of application specific data and propose a generic approach for application auto-discovery. Generic means that the data used as source for the approach is not specific to an application, but can be found in many applications. Machiraju et al. have made basic considerations on generic application auto-discovery and outlined the importance to understand a system for proper administration and development of improvements. They argue that it is essential to know about "what to discover" and "how to discover" [MDW+00] before applying an automated discovery to an application system. In a specific discovery approach both is preconfigured in the algorithm at design time of the approach. In the ATM example "what to discover" is predefined with *all neighbor ATMs* and "how to discover" is predefined with *use the routing tables*. Both cannot be changed at execution time. In contrast, for generic discovery approaches both need to be input variables which are set at execution time, because at design time both is not defined. Machiraju et al. call those variables *variants of applications* which enable the discovery process to discover instances of any application as shown in Figure 3.4. Dependent on the input variants (different colors in the figure) on the left side the generic auto-discovery approach can discover different applications instances on the right side. For specific auto-discovery approaches (Figure 3.5) the variants are predefined (other colors do not match).

Machiraju et al. focus on the existence of application instances in an IT infrastructure. Dependencies between them are not discovered. The results are presented in an UML-Model. Figure 3.6 shows an example, consisting out of three applications. The applications are connected, that means they belong all to one IT infrastructure. An administrator knows with this model, that these three applications are running in his infrastructure. The applications contain several properties like a name and description. The *DiscoveryTechniques*-property indicates the information which was used to discover this application. Machiraju et al. use five different information sources [MDW+00]:

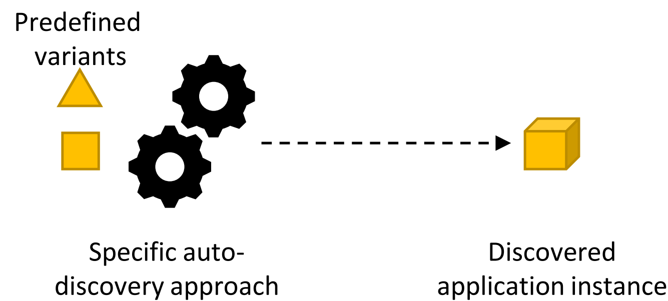

Figure 3.5: Specific auto-discovery with predefined variants

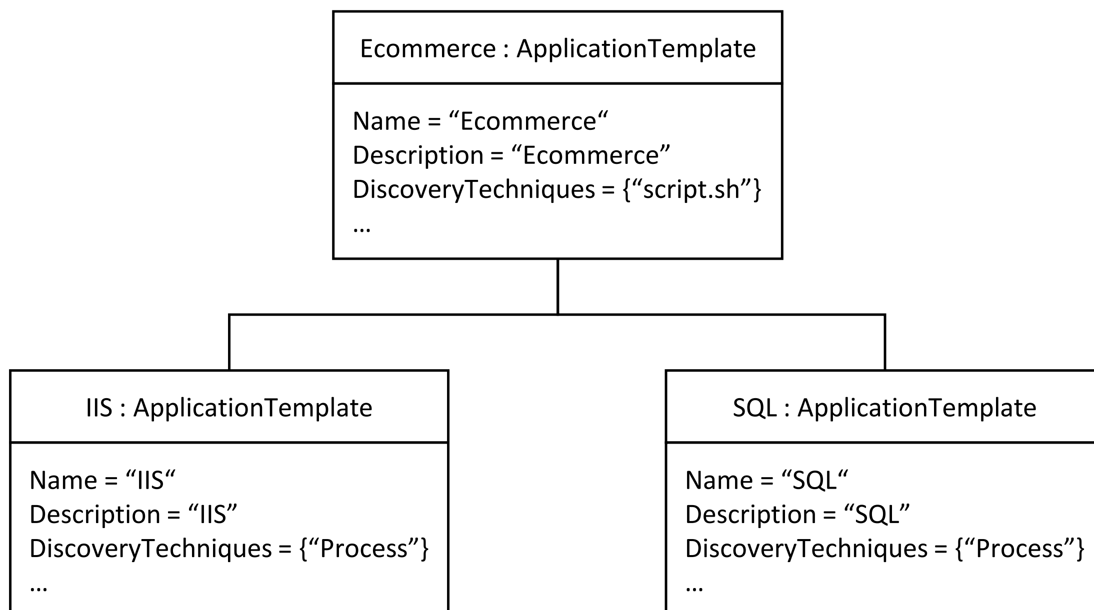

Figure 3.6: Example application overview model by [MDW+00]

- **Processes**: By analyzing all Process Identifiers (PIDs) and process names the existence of processes and therefore applications on that operating system can be discovered. The same is possible for services.
- **Files**: A file's name and it's content can indicate the existence of applications.
- **Registry**: The registry of the operating system refers to installed applications.
- **Network services**: Analyzing pre-collected data like Domain Name System (DNS) lists can indicate the existence of applications.
- **Application Programming Interfaces (APIs)**: APIs of applications can be used to gather information about the existence of applications which access the API. This source is not generic and needs application specific knowledge.

All sources are gathered from the operating system and analyzed to discover the existence of applications. For example, if *apacheHttpServer* exist in the list of processes the application *Apache HTTP Server* can be added to te list of discovered application instances.

The approach of Machiraju et al. can be used to discover the existence of applications. However, to be able to create a model of the complete topology of an enterprise IT infrastructure the knowledge about pure existence of an application is not enough. The relationships between applications need to be discovered as well. Other generic approaches try to solve this. The following section depicts examples.

3.2.2 Generic Automated Discovery Processes

The knowledge of relationships and dependencies between applications is necessary to perform changes on the IT infrastructure. Examples of changes are:

- **Removal of an application**: To avoid breaking applications that depend on an application, all parts of the system, that depend on this application need to be known [CZMB08].
- **Relocation of an application**: Hard references like explicitly defined addresses to and from an application may break by a relocation. Beside traffic may be blocked due to firewalls and other network segmentations between the old and the new location of the application [KGE06].
- **Update of an application**: The new version of an application may not work together with other parts of the system. The affected applications may also need an update to avoid problems.

A dependency discovery approach has to provide the expected discovery results. Besides this functional requirement Breitenbücher et al. [BBKL13], Chen et al. [CZMB08] and Binz [Bin15] state summarized seven non-functional requirements for dependency discovery approaches:

- **Quality**: the result should be complete, accurate, up to date and show the necessary granularity.
- **Expandability**: the resulting models need to be expandable for current and future components. The approach should be applicable to all topologies.
- **Integration**: It should be possible to use existing data sources.
- **Effort**: the human effort needed should be low.
- **Ease of use**: it should be easy to use the approach.
- **Influence**: the effects on a productive system used as source should be very small. The operation results of the productive system should not be influenced by the discovery.
- **Scale**: the approach should scale with the size of the network.

To achieve these requirements Chen et al. [CZMB08] are using network traffic as source data for their approach "Orion". Only common Transport Control Protocol (TCP) and User Datagram Protocol (UDP) headers are used to analyze the traffic. This promises an generic approach, because no adaption for a specific system is necessary. Application-specific headers provide more information, but an adaption of the generic approach to a specific system is necessary. The headers of Open Systems Interconnection Model (OSI) layer 4 (transport layer), which includes TCP and UDP, do not carry any information about dependencies between applications, but they carry the source and destination of the packet. To discover dependencies the approach calculates delays between packets and creates a distribution out of them to notice recurring behavior. If two applications frequently exchange packets it is likely, that they depend on each other. The approach is executed for a single application to discover all dependencies of this application to other applications. Hence, the discovery process can run independent for different applications and scales with the size of the network. The complete dependency topology can be created by combining the single results.

Orion is divided into three steps: (i) *flow generation* which aggregates packets into a flow, (ii) *delay distribution calculator* which calculates delays between the flows and (iii) *dependency extractor* which orders the flows. In the first step TCP respectively UDP packets, which match in local IP and Port, remote IP and Port and protocol and follow each other with a maximum delay of a threshold, are aggregated into a flow. In the example in Figure 3.7 each flow consists out of 2 packets. In the second step the delays between flows are calculated and added to a distribution. Two heuristics are used to decrease the number of compared flows and therefore reduce the calculation effort. First only flows within a maximum delay threshold are compared. Second only flows of persistent services are observed. A service is called persistent if it has a minimum number of flows. Otherwise it is treated as ephemeral and therefore not important for durable dependencies between applications. For the example in Figure 3.8 the delays between flows A-C, A-D, B-C and B-D are relevant and calculated. Between A-B and C-D no delays can be calculated, because A and B respectively C and D occurred between the same services. In the third step the calculated delays between flows are ordered into areas with a defined time window. Recurring requests between two applications, which indicates durable dependencies between these two applications, have likely stable execution times and therefore nearly the same delays in the network. Time window areas with a high number of classified delays between flows indicate such recurring request. In the example in Figure 3.8 A-C and B-D have the same or at least nearly the same delay, that's why they are classified into the same groups. Therefore, a dependency between A-C and B-D is discovered. It is suspected, that the packets of the flows A and C belong to the same packet-chains and therefore the same events as the packets of the flows B and D.

A similar approach is published by Kind et al. [KGE06]. They are also using network traffic to discover dependencies between applications. In their case the profiling protocol NetFlow [Cis14] is used for discovery of dependencies. In contrast to the Orion-approach, they try to evaluate the use of active discovery of relationships between applications by adding special packets. However, they discard this idea, because it has an influence on the productive system and actively added additional traffic may be blocked by firewalls located between parts of a big network.

NetFlow is a widespread protocol for traffic profiling. The approach of Kind et al. [KGE06] is using the traffic flow data provided by NetFlow rather than generating it by themselves, which means that the flow generation step from Orion is replaced with NetFlow. Kind et al. [KGE06] are using time differences (delays) between flows to identify flow chains, too. In addition, they calculate a correlation confidence value which indicates the probability that the identified correlation between

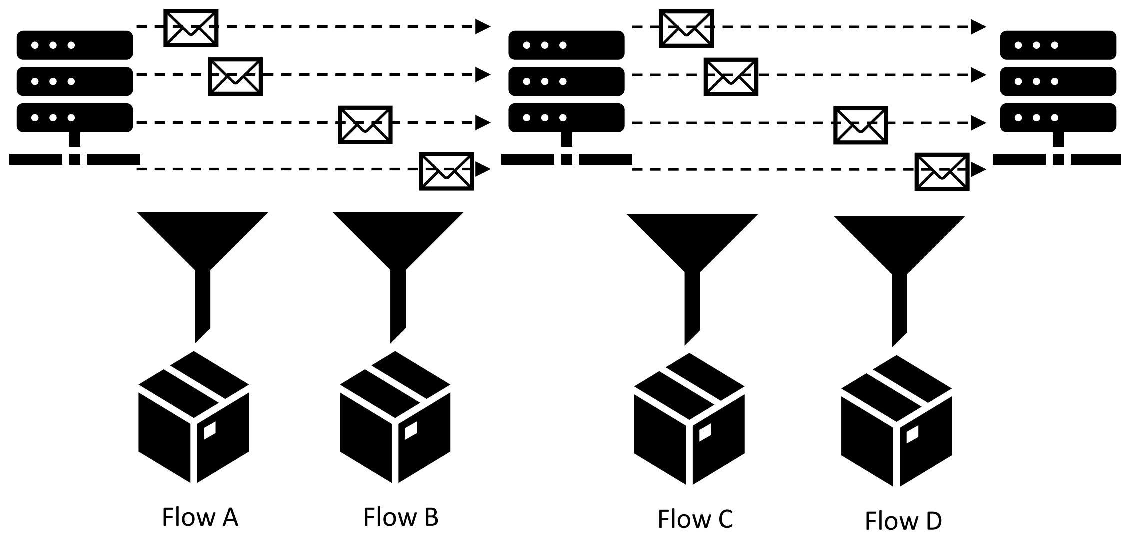

Figure 3.7: Step 1 of Orion approach [CZMB08]: Two packets are combined to a flow in each case

two services is correct. It is calculated based on the number of inspected flow-events as well as this number relative to the number of all flow-events. They claim that the confidence value together with the calculation results of the flow chains is a useful metric to discover indirect dependencies between applications. Indirect dependencies are relations over several steps.

3.2.3 Discovery of Data Dependencies

The previous approaches of Chen et al. [CZMB08] and Kind et al. [KGE06] focus on the pure existence of relationships between applications. Other approaches, for example the approach of Magoutis et al. [MDJV08], try to discover the data, which is transferred over these relationships between applications as well as towards storage systems. With this information more precise questions regarding improvements and optimizations can be answered [MDM07].

Magoutis et al. [MDJV08] published an approach called *Galapagos*. As source data they use topology graphs created by other discovery tools, for example, the Orion-approach. Experts of the analyzed system use the topology graphs to create models of the data, called *Data Locations Template (DLT)-models*. Every component of these models is either a data provider or a consumer. A consumer consumes data sets which are provided by a data provider component. The meta-model of the approach in Figure 3.9 depict a consumable data set on the top right corner. A consumable data set is assigned to an exported data type of a data provider on the top left corner. This assignment allows to build chains by string several DLT-components together, because every component can appear as data provider and data consumer simultaneously. This step is done automated and creates end-to-end data paths with explicit models of the data. Though Magoutis et al. extend the approach of [MDM07] in [MDJV08] to automate one step of the model creation process, still manual effort is needed to create the DLT-models. The approach is not fully automated.

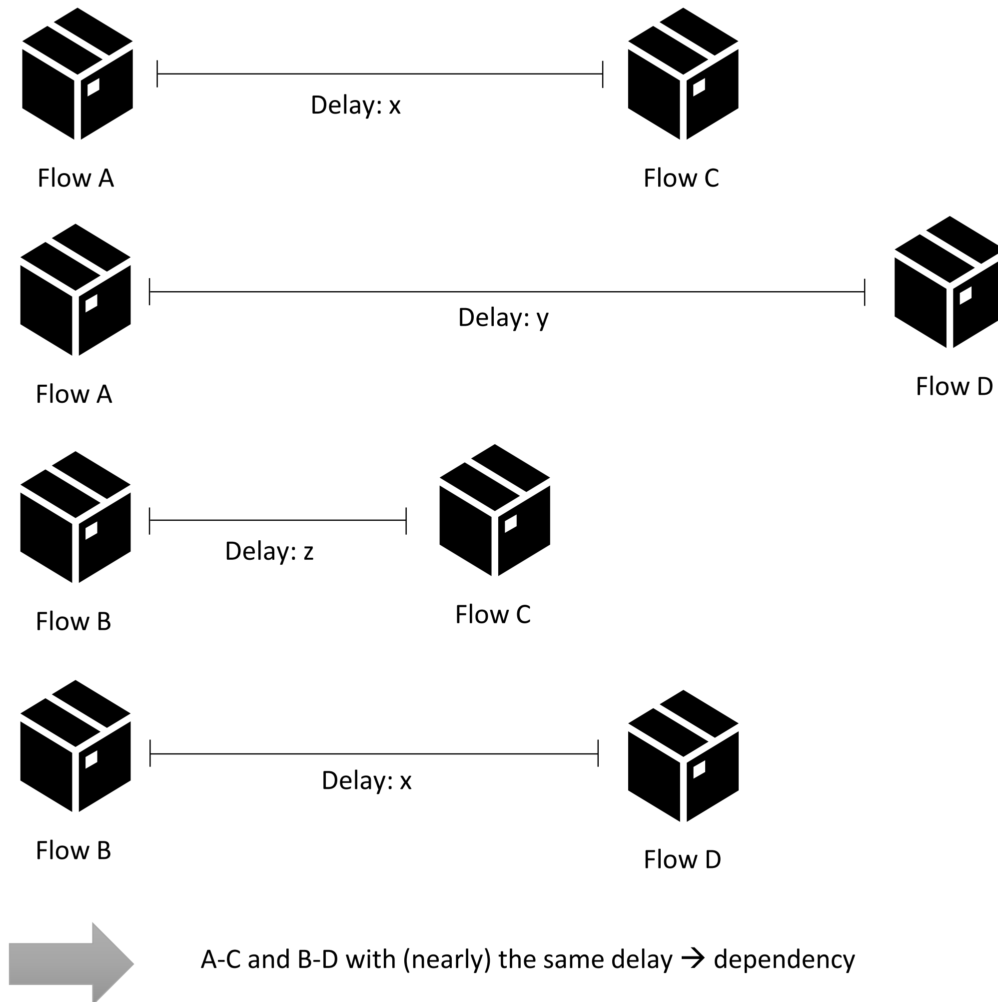

Figure 3.8: Step 2 and 3 of Orion approach [CZMB08]

The topology discovery approach of Joukov et al. [JPRD10] suffers from similar problems: Either manual expert knowledge must be applied to model all incoming and outgoing relations of a component or specific plugins for applications must be manually created. These relations are combined to chains by this approach to discover the application-data dependencies. Likewise, this approach is not fully automated. An efficient and fast discovery with modest human effort is not possible with the approaches of Magoutis et al. [MDJV08] and Joukov et al. [JPRD10].

3.2.4 Summary

There are several automated discovery tools and approaches available, which provide accurate and useful topology models of multi-component systems. One drawback of these tools is that they are based on running systems. Often these systems are productive systems and any kind of

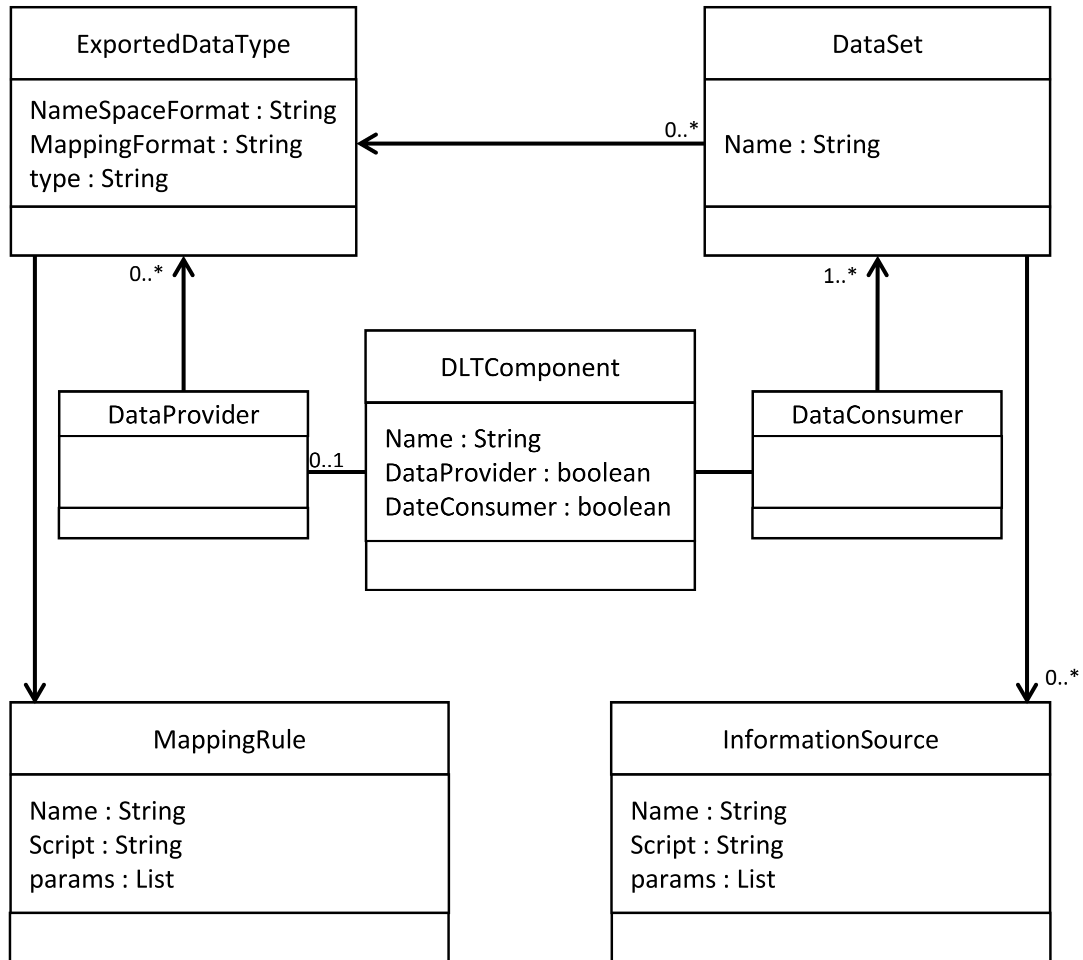

Figure 3.9: Meta-model of DLT-models by [MDJV08]

negative influence on these is not tolerated. But even if the normal process of such an approach does not influence the analyzed system, side-effects like an overutilized network switch due to additional traffic may occur. Furthermore, it is very expensive to carry out regular analyzes of a broad distribution of systems to gather data for models, which indicate runnable deployment structures [CZMB08], because on the one hand the effort would be huge, and on the other hand companies often do not allow to analyze their systems, if the results are published. However, for use cases as introduced in Chapter 1 a wide spread base and big number of source structures is necessary to guarantee correct results.

3.3 Crawler

Analyzes performed on large data sets can reveal results which are not found by analyzes on smaller data sets [SS13]. However, large data sets are often not available and must be gathered first. In the web an enormous amount of data is public available, provided by many different online services.

Algorithm 3.1 Basic steps of a crawler [HN99]

```
procedure CRAWLER(urls)                                    // Input: Seed URLs
    while urlList.notEmpty do
        nextUrl ← UrlList.next
        newWebside ← DOWNLOADWEBSIDE(nextUrl)
        alreadyVisitedUrls.add(nextUrl)
        urlList.addIfNotAlreadyVisited(extractUrls(newWebside))
        SAVEWEBSIDE(newWebside)
    end while
end procedure
```

Mostly the data of online services is not provided as complete data set, but is split into logic parts, for example, single repositories in case of software project data. Crawlers can be used to collect distributed data according to the requirements of an analysis.

In the beginning of the internet crawlers were invented to collect all available web sites [HN99]. Section 3.3.1 depict an example. If an analysis needs very specific data which is only provided by a small number of online services it is more suitable to crawl specific web pages instead of the whole web. In Section 3.3.2 *page-specific crawlers* are depicted. To accelerate crawling processes the crawler can be distributed over several machines. This is described in Section 3.3.3.

3.3.1 Web Crawler

Heydon et al. [HN99] published an approach for web crawler. They note scalability as important requirement, because the web is very large. To fulfill this requirement Heydon et al. [HN99] pay attention to bound the in-memory data to a maximum size and write data beyond the maximum size to persistent storage. This results in lower main memory demand and improves the scalability. In addition, they use different components to divide tasks to be able to scale them independent:

- **Uniform Resource Locator (URL) list ("URL frontier")**: A list of all URLs, that still need to be visited.
- **DNS resolver**
- **Already-visited list**: Holds all URLs, which were already visited.
- **Download component**: Takes URLs from the list, downloads the page and passes it to the next component. Removes the URL from the list and adds it to the Already-visited list.
- **Extract component**: Searches in a downloaded page for URLs and adds them to the URL list, if it was not visited before.

Heydon et al. [HN99] implemented a loop of several steps, which are reused and adjusted in many other crawler [BNS18] [FC13] [MLJC17]. These steps are shown in Algorithm 3.1. In a loop the next URL is taken and visited. The web page is downloaded and the URL added to the Already-visited list. All URLs that are found on the downloaded web page and were not previously visited are added to the main list of URLs. As last step the web page is saved. As long as more URLs are in the main list the loop continues.

3.3.2 Page-Specific Crawlers

Page-specific crawlers are used to collect data from predefined online services. They can be adjusted to the specific data which is provided by the crawled web page. In the following two examples are presented.

DockerFinder [BNS17] is a tool, which provide a comprehensive search function for Docker image repositories like Docker Hub[1]. First all images provided by the image repository service are crawled. To adjust a crawler to a specific online service like Docker Hub an additional difficulty need to be considered: The container images are much bigger than simple web pages. The download and processing components may need more time for their tasks. To avoid blocking other parts of the system an asynchronous crawler is necessary as shown in Figure 3.10. A *Message Broker* is added and the *Crawler*-component crawls only the name of an image and send it to the broker [BNS17]. The broker takes care of forwarding the names to download and processing components called *Scanner*. They can be scaled independent of the other components according to the current requirements. The example in the figure contains three Scanner. They download and analyze the docker images. The discovery results are provided over an API and are stored in a storage. Users can access the results over the API. The *Checker*-component is responsible to ensure eventual consistency, as the source data is dynamic and can change during the processing time.

Another main difference to web crawler is the discovery of docker images. Between web pages are URL links that can be followed to discover more pages. In contrast, between the docker images on Docker Hub and similar services are no links, that can be followed. Instead the search function of the image repository service must be used, which provides the functionality to list all available images [BNS17]. The authors extended their approach [BNS18] and added functionality to configure the crawl method.

Farah et al. [FC13] published a tool, which depends on a page-specific crawler. They give an overview over the popular software repository service GitHub[2]. The tool can be used to create different statistics for GitHub. Therefore, different public data needs to be crawled. They outlined several problems:

- Likewise DockerFinder, no or only few links between software repositories are present. Therefore, the search function need to be used.
- The result set of the search function is limited to 99 repositories.
- The information of a repository is spread over several html pages.
- For a repository a maximum of 1000 commits are shown.
- The RSS feed of commits show a maximum of 20 commits.
- GitHub is running a rate-limiter, which blocks too many requests from one IP in a short period of time.

[1] https://hub.docker.com
[2] https://github.com

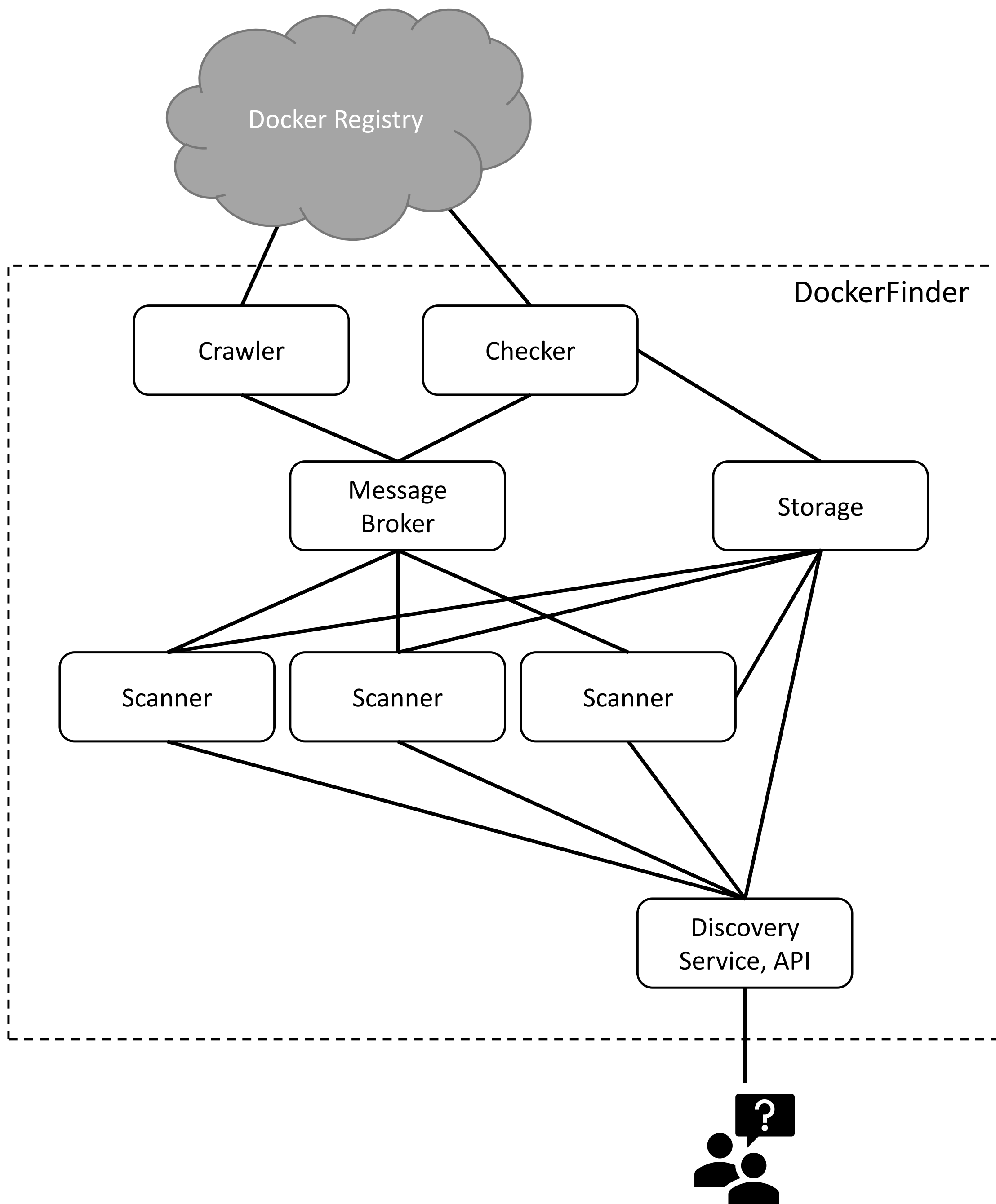

Figure 3.10: Architecture of DockerFinder based on [BNS17]

To solve these problems requests are split and the approach is executed on several machines independent of each other, but with different IP addresses, which perform request inside the search function rate limits.

3.3.3 Distributed Crawling

Distributed networks perform tasks spread over different machines. Often not every machine performs the same tasks, but is responsible for specific parts of the work of the system. The machines need to synchronize. This can be done for crawlers.

Ma et al. [MLJC17] published an approach which analyzes the developer accounts on GitHub. They use a distributed crawler to collect the account data from GitHub, because they identify the same problem as Farah et al. [FC13]: The rate limiter of GitHub limits their crawling speed. A difference to the approach from Farah et al. [FC13] is that they can use links between accounts ("follower") similar to html-links between web pages.

Three modules are necessary for a distributed crawler: *Crawler*, *Task Assignment Module (TAM)* and *Result Collection Module (RCM)* [DCF13]. The TAM holds a list of user accounts represented by their Unique Identifier (UID) similar to the URL list of a web crawler (Section 3.3.1). Many social network services identify their user by such a UID. The Crawler module is, in contrast to the TAM and RCM, instantiated in a crawler pool. An important aspect is, that each instance needs to have a unique IP inside the pool of Crawlers. The architecture is shown in Figure 3.11. The TAM specifies and distributes tasks for each Crawler instance (orange arrows). A task contains a set of the list of UIDs. The Crawler module instance crawls the corresponding data (green arrows) and forwards it to the RCM (blue arrows). The RCM analyzes the data and forwards linked UIDs ("follower/friends") to the TAM (yellow arrow). The TAM adds them to the list of users. Every UID is added to more than one task by the TAM and therefore the same data is crawled redundant and can be compared by the RCM to find malicious instances. Besides, the RCM holds a timeout for every Crawler task to detect crashed instances. The TAM has to notify the RCM about new tasks (black arrow) to enable the RCM to create the timeout-timer. This timeout is dynamically adjusted based on previous time measurements of tasks of the specific Crawler.

Each Crawler-component operates on 80% of the permitted request rate ("rate limit") of the service. This might be very slow for a single instance, but the complete system carries out the crawling task significantly faster compared to a single instance, if there are enough instances of Crawler-components available.

3.3.4 Summary

Web crawler technology is well known for quite long time. The fundamentals of web crawlers can be used for page-specific crawlers. Page-specific crawlers are simpler, because the expected data is well formatted due to the limitation to one source. But on the other hand, additional restrictions like rate-limiting are present when accessing one service heavily. Distributed Crawling can be a good solution to solve this issue, but it increases the complexity of the crawler significantly.

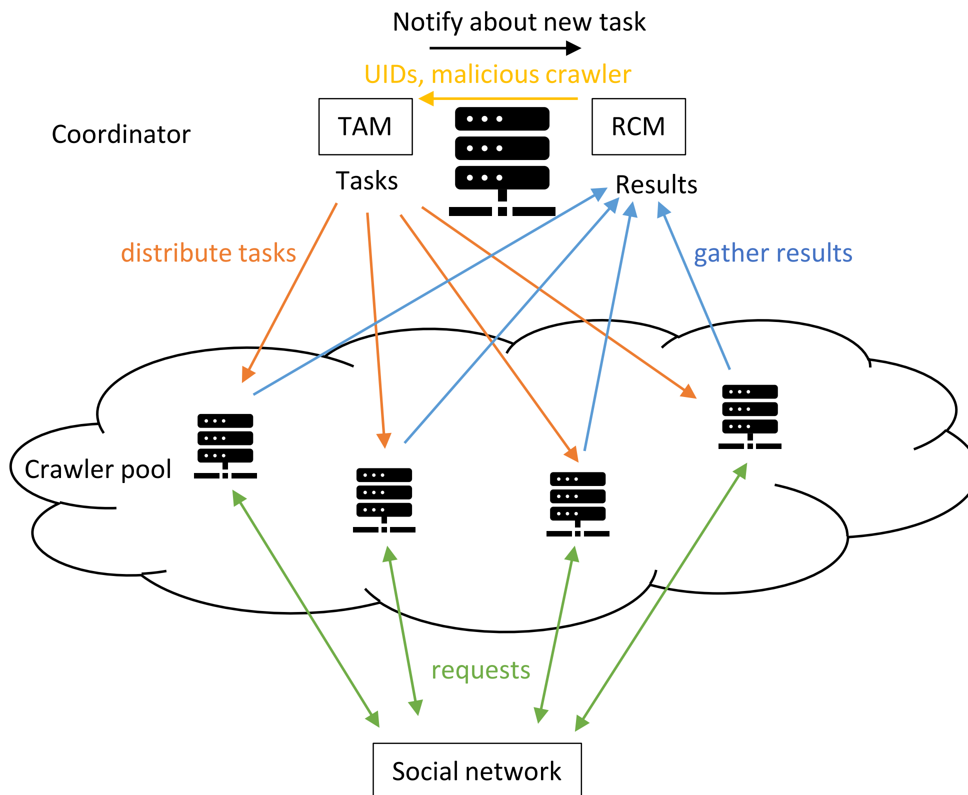

Figure 3.11: Crowd crawling architecture by Ding et al. [DCF13]

3.4 Model-Driven Docker Management

Paraiso et al. published an approach for model-driven management and analyzes of Docker in 2016 [PCAM16]. They identify several problems of Docker which partly coincide with problems tackled by this work. They provide an approach for modeling and deployability verification of Docker containers.

3.4.1 Model-Driven Approach

Paraiso et al. [PCAM16] identified four problems with docker containers:

- **Lack of verification**: There are design tools for docker container available. But the deployment of the containers is lacking any verification tool. The only way to check, if a container can be deployed is to actually deploy it. This can be error prone due to human errors. This problem corresponds to the missing deployability verification as introduced in Section 1.1.

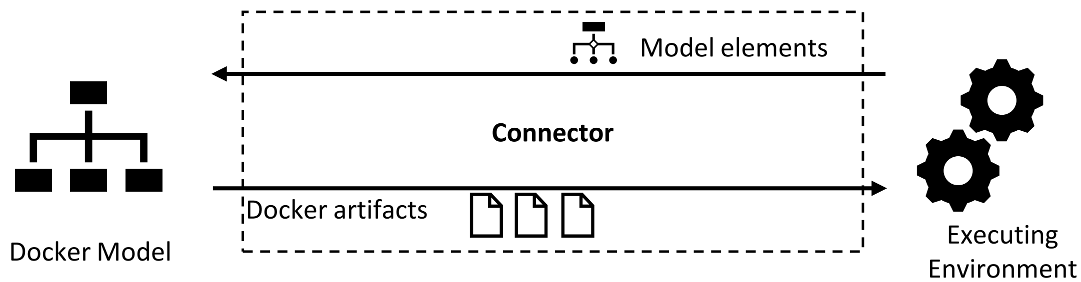

Figure 3.12: Architecture of the MDE-approach by Paraiso et al. [PCAM16]

- **Resource management at runtime**: The resources of a docker container can be defined in the creation process. But Docker does not provide any possibility to change and adjust the resources at runtime. Paraiso et al. [PCAM16] argue, that adjustments to meet the current workload are helpful.
- **Synchronization between design and execution environment**: A docker execution environment is created based on the predefined architecture. After the creation process is finished updates on the architecture are not applied to the execution environment and vice versa. Paraiso et al. [PCAM16] criticize this.
- **Inconsistent use of containers across organization**: As last major problem Paraiso et al. identified the heterogeneity of different container technologies, which makes it difficult to maintain them.

To solve these problems they use Model-Driven Engineering (MDE) [Béz05]. Updates to the model are automatically propagated as *Docker artifacts* over a connector to the executing environment and changes to the executing environment are automatically applied to the model as model elements (Figure 3.12). Especially problem one (Lack of verification) and three (Synchronization between design and execution environment) are addressed by this attempt. They argue, that it is easier to check the deployability with models at a higher abstraction level compared to the low-level container representations docker provides. Thus, they introduce a model divided in three levels as shown in Figure 3.13. On the bottom level the *Docker Model* provides a simplified view on a docker container. It consists out of several parts:

- **Machine**: This entity represents a physical or cloud VM, on which the Container is running and holds properties and attributes of it. It is extended by different possible machine types.
- **Container**: This entity represents the container itself and holds properties like name and image of it.
- **Volumesfrom**: This entity represents a storage for persistent data, which is attached to one or more container instances.
- **Contains**: This entity represents the relationship between machine and container instances.

On the middle level the *Infrastructure Model* abstracts the cloud infrastructure resources. The Docker Model extends this Infrastructure Model. On the top level *OCCIware* [OCC17] is used to model relationships between container instances. OCCIware is a metamodel to define interfaces between cloud resources. Several container instances can be linked using the *Link*-entity. This entity references source and target container instance.

The authors evaluated their Docker Model approach by comparing the time needed for creation, start and stop of docker containers. They identified a overhead time of 1.11% to create docker containers. To start a container the time increased by 2.12%. The time to stop a docker container increased by 2.25%. Paraiso et al. [PCAM16] conclude that the introduced overhead is negligible compared to the advantages of their approach.

3.4.2 Summary

Paraiso et al. [PCAM16] tackle several problems of Docker. However, they partly ignore microservice architectures of cloud system. Containers are kept small in these systems and operate as function blocks, which are scaled by the cloud platform. Changes to the resources of a single container are not necessary. In addition, business logic changes to running containers are not necessary, because it is faster to redeploy these small containers. Runtime updates between model and executing environment are not necessary.

On the other hand the Model-Driven approach can help to analyze and understand containers and relationships between them.

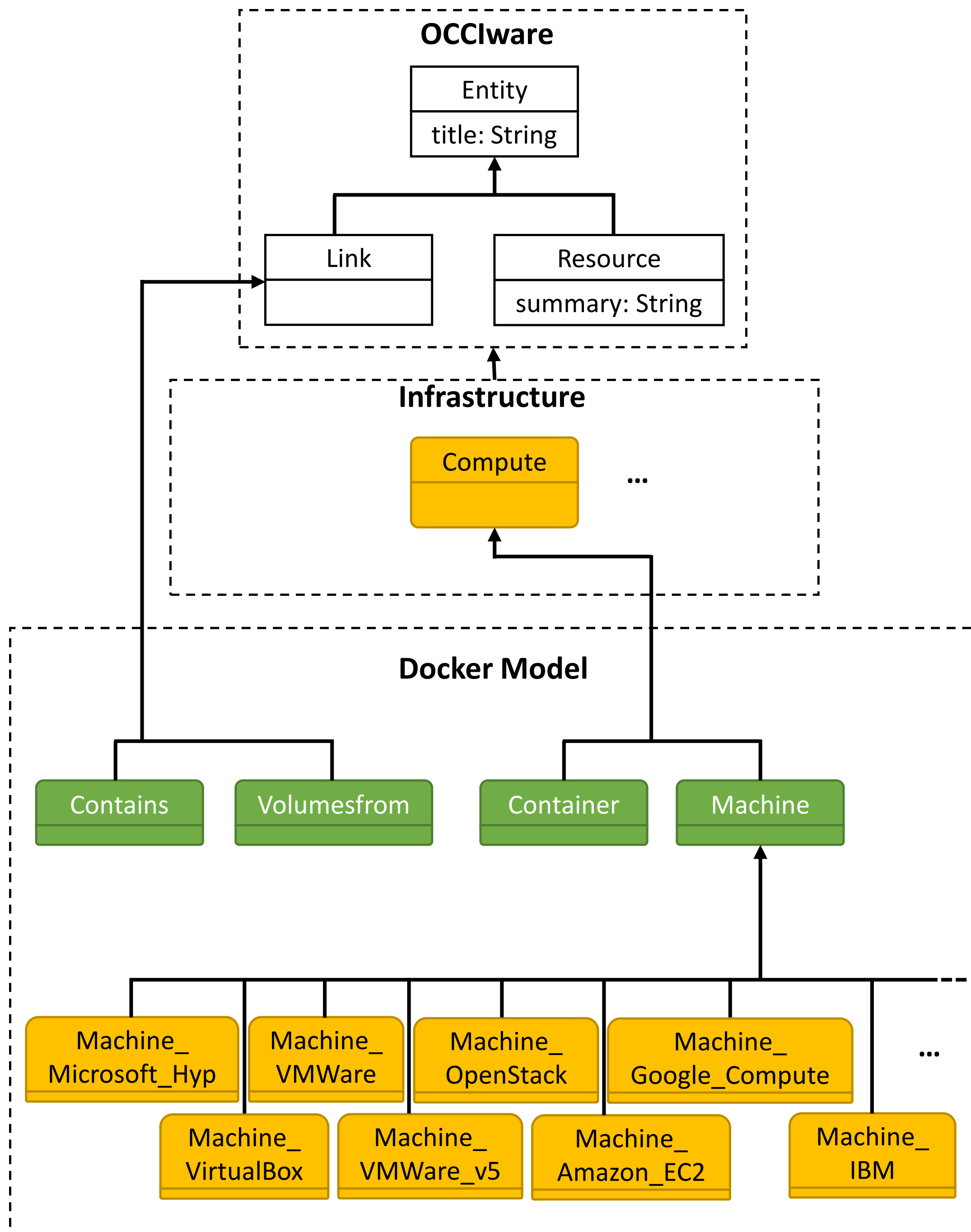

Figure 3.13: Metamodel of the MDE-approach by Paraiso et al. [PCAM16]

4 Considerations for an Approach for Automated Discovery Based on Dockerfiles

This chapter depicts the requirements and overall challenges for a concept and implementation of a deployment component discovery system for docker services. It outlines possible solutions. The selected solution and their details are presented in Chapter 5.

4.1 Requirements

This section gives an overview over all requirements for the dockerfile discovery approach.

Requirement 1 (RQ1): Automated process. The system must be able to analyze and evalute the data automatic without any manual help. Once started the system responds eventually with the result set.

Requirement 2 (RQ2): Scalability. The system must be able to operate on a huge data set of dockerfiles.

Requirement 3 (RQ3): Collection of dockerfiles. The system must be able to collect the dockerfiles, which it analyzes by its own. The system can use public available internet services. No specific dockerfiles are needed.

Requirement 4 (RQ4): Expandability. The system must be expandable. New data source services can be added. New instructions in dockerfiles, which indicates installation of applications can be added as recognized instructions.

Requirement 5 (RQ5): Robustness. The system must be able to deal with misleading content in dockerfiles, for example, comments or scripts with application names.

Requirement 6 (RQ6): Preparation of results. The results must be depicted in an understandable way.

4.2 Overall Algorithm

This approach has to fulfill several tasks to deliver the expected results. This has to be done automatically to fulfill RQ1 “Automated process”. First the crawler needs ot gather dockerfiles. Second, the analyzer has to extract the components and their relationships. Third, the results have to be processed and depicted into a model (Figure 4.1).

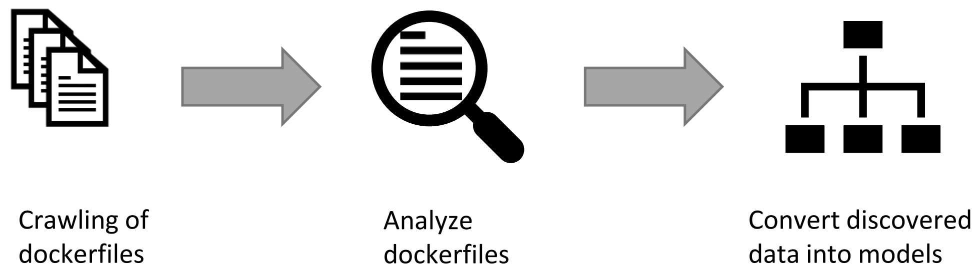

Figure 4.1: Steps of the component discovery approach

Algorithm 4.1 Basic algorithm steps of this approach

```
procedure GetModelOfKnownDeployableComponent
    dockerfiles ← crawler.crawl()
    for dockerfile in dockerfiles do
        newComponents ← AnalyzeDockerfile(dockerfile)
        listOfKnownComponentRelatoins.add(newComponents)
    end for
    return listOfKnownComponentRelations.buildModel()
end procedure
```

Algorithm 4.1 shows the pseudocode of the basic structure of this approach. As first step the crawler crawls dockerfiles in line two. This is the first step of Figure 4.1. In an iteration over all crawled dockerfiles ()line three to line six) they are analyzed one by one. This is the second step of the algorithm. All discovered components are added to a list. When the loops finish and no more dockerfiles are available, as third step the list of known component relations is converted into a model and returned.

The following chapters outline the challenges of these three parts of the approach and describe possible solutions.

4.3 Crawler

The crawler is the first part of the approach. It is responsible for gathering the data source, namely dockerfiles. Before conceptualizing a crawler, the source of the crawled data has to be defined. As stated in Section 3.3 the source of a crawler can be the whole internet as well as designated online services. The content which should be crawled by the crawler of this approach is a very specific content. Therefore, it is useful to specify the services the crawler uses as source and use a page-specific crawler. Challenges for page-specific crawlers are outlined in Section 4.3.1. A possibility for data sources is described in Section 4.3.2.

4.3.1 Challenges of Page-Specific Crawler

Four challenges need to be considered for page-specific crawlers (see Section 3.3.2):

- Rate Limit: Many online services protect their system by rate limits. Page-specific crawler access one service heavily and have to consider these rate limits. Otherwise the accessing system may get temporary or permanently blocked. Considering a rate limit might extend the execution time.
- Download extensive content: Specific content is often bigger compared to metadata crawled by other crawlers like web crawler. This has to be considered.
- Links between content: Web crawler use links found in previously crawled content to search for new data. Dockerfiles might not have any links between each other. Other discovery methods need to be used.
- Limited result sets: Many online services restrict responses which include possibly a huge amount of data to a maximum size. The approach needs to consider this by splitting requests into smaller parts. This extends the execution time.

There are different possible solutions for the rate limit challenge present. The suitable possibility highly depends on the specific rate limit implementation of the accessed service and is hard to determine. Three possibilities are:

- The local system holds a counter for the amount of send requests which is zero at the start (Figure 4.2). After every response this counter is incremented. If the next request has to be sent (step 5 of the figure) and the counter exceeds a threshold (“Check counter”) the system waits and do not send any further requests until the counter is reset. The counter is reset after a specific time interval or at specific points in time, for example, every full hour, depending on the method of the accessed service.
- Alternatively the online service holds a discontinuation time for every single request. To comply with this method, the local system has to save every point in time of a request added up by the rate limit blocking time of the service into a list (steps 3+4 in Figure 4.3), for example 12 o’clock plus 10 minutes. After a point of time in the list is reached, it has to be removed out of the list (step 5 in the figure). Before sending new requests the list size has to be compared with the rate limit. If the list size is higher than the rate limit, the local system has to wait.
- The online service can add a field in every response which contains the open rate limit. This lowers the effort for the local system, because is does not have to calculate anything. It only has to check the value of this field for the following request (“Check rate limit window field” in Figure 4.4). If it is zero, the local system waits before sending new requests. Either the online service adds a time when the window will open next as shown in the figure or the local system has to poll the service regularly for the current rate limit window before it continues with sending requests.

The second challenge, download of extensive content, is unproblematic for this approach, because it crawls for dockerfiles which have only a few kilobyte in size. As result, the time needed to download the dockerfiles is no problem.

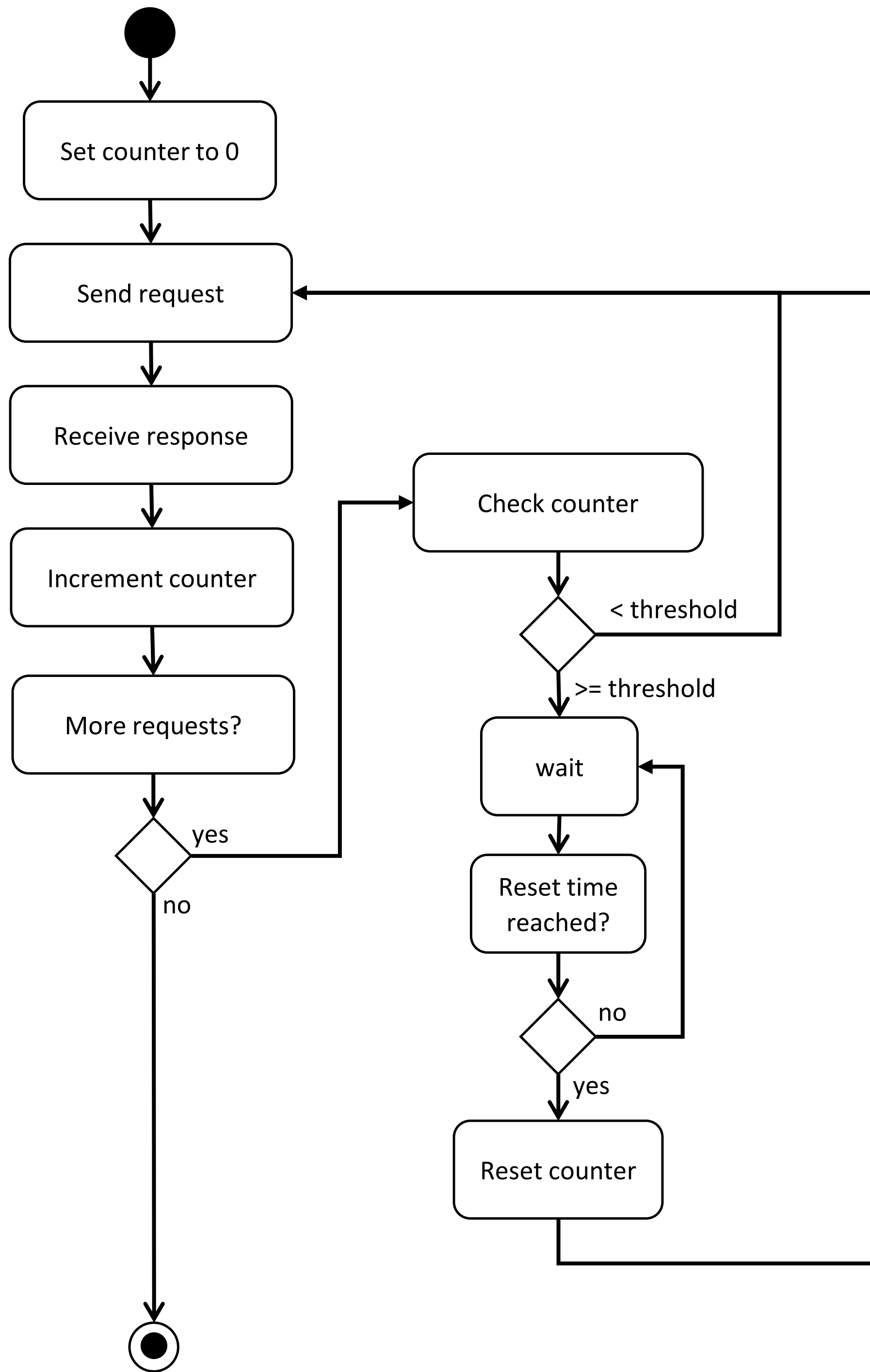

Figure 4.2: Activity diagram of a rate limit counter solution

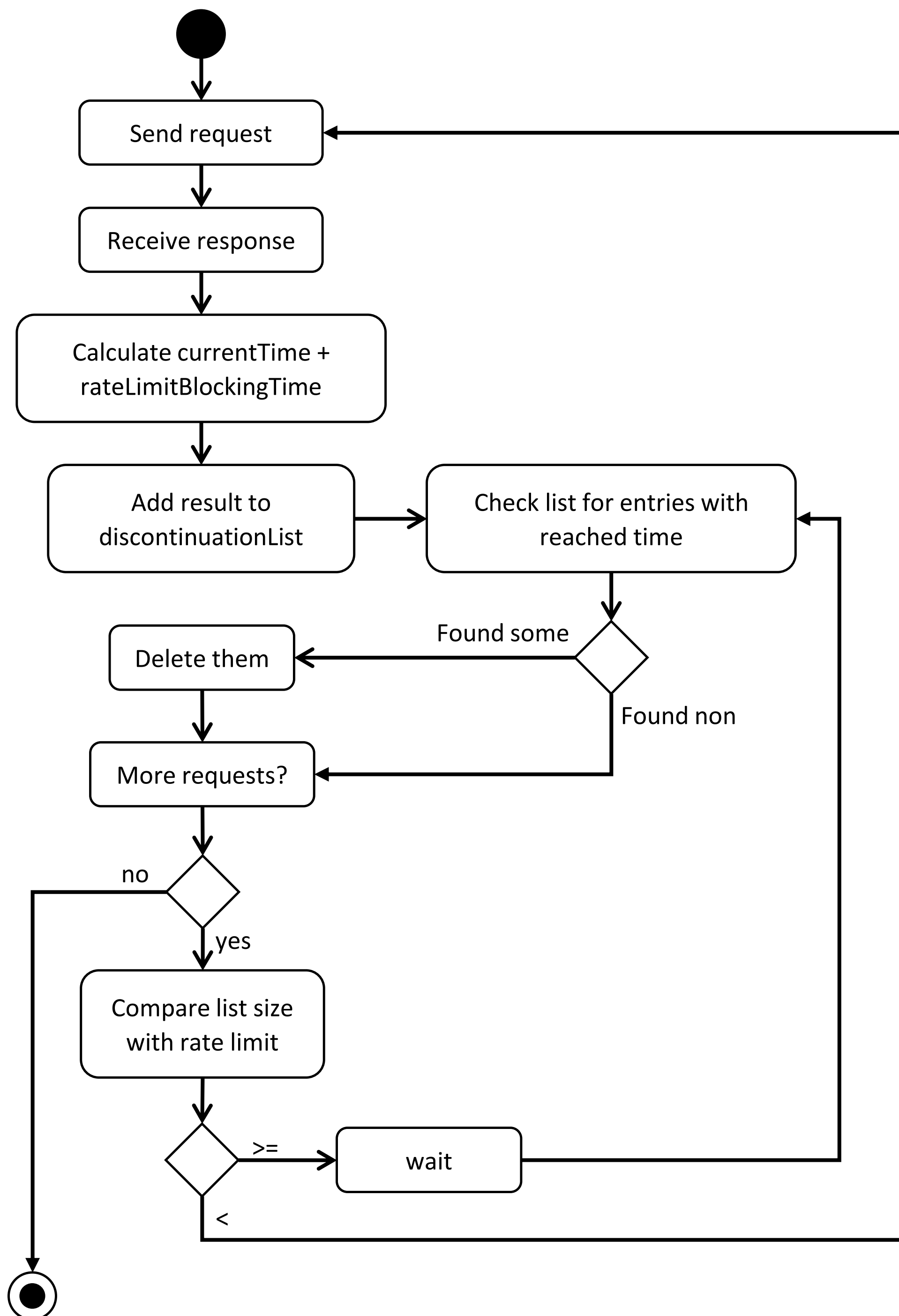

Figure 4.3: Activity diagram of a rate limit discontinuation time solution

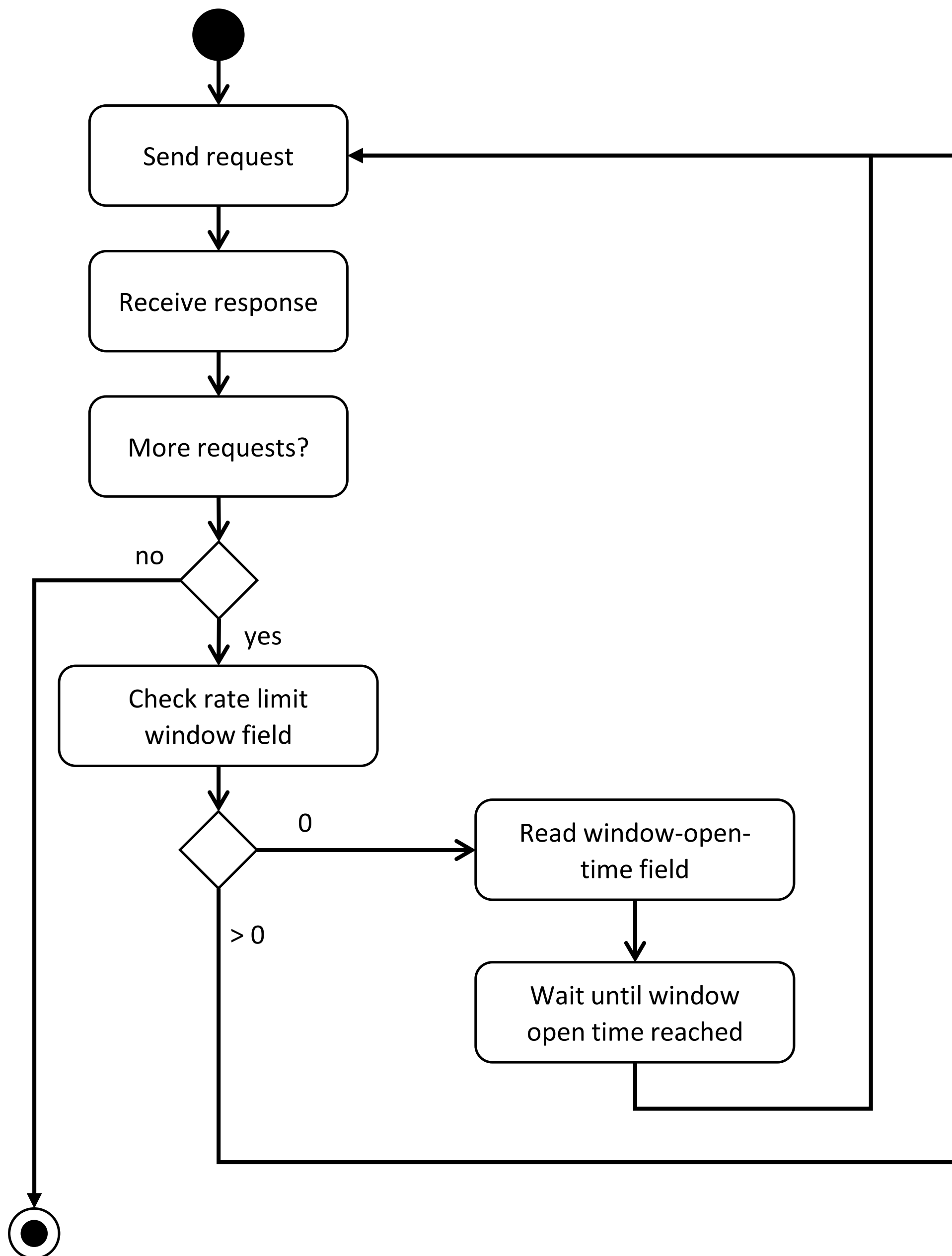

Figure 4.4: Activity diagram of a rate limit open window solution

Listing 4.1 Dockerfile and SQL FROM instructions

```
// dockerfile
FROM ubuntu

// SQL
SELECT *
FROM ubuntu

```

To replace the missing links between dockerfiles or repositories of the third challenge the search function of the accessed service can be used. To deal with limited result sets continuation tokens can be used, if offered by the accessed service. Ideally, the online service do not add the requested content directly in the response. The response just contains links to the actual location of the content, which can then be requested individually. This makes the responses smaller and unnecessary to split responses.

4.3.2 Software Repository Services

A possible group of services which can be used as source for this approach are software repository services like GitHub[1]. They provide version control system repositories for software systems. Dockerfiles can be found in these repositories, because they are part of software systems.

Repositories have a file structure. In principal, a dockerfile can be located everywhere in a file structure of a repository and it can have any name. They are not tagged or otherwise marked. The only possibility to find them is using string-compare. Docker allows any name for a dockerfile, but the default of the *docker build* command is “Dockerfile” without any extension. By searching for the filename dockerfiles with non-default names could be missed. Additional searches for content might find them. However, this has three disadvantages:

- Execution time: Searching through the complete content of files needs more time then only searching for filenames.
- Rate Limit: Additional search queries need additional requests to the online service. The rate limit exhausts faster. This increases the execution time.
- False positives: It is hard to identify dockerfiles by their content. Dockerfiles consist of instructions as in Section 4.4.1 described. The only mandatory instruction is “FROM”. This instruction can easily be mixed up with other types of instructions like SQL instructions as the example in Listing 4.1 shows. The second line uses *ubuntu* as base image in a dockerfile. The other two lines select everything from a table called *ubuntu* by using SQL. Also the other dockerfile instructions can be mixed up, because they are all only single words. The arguments of the instructions can be chosen completely free by the author of the dockerfile. This concludes to many false-positives.

[1] https://github.com

4.4 Analyzer

The analyzer is the main part of this discovery approach. As first step all relevant lines must be extracted based on docker instructions. Next the components have to be identified. Then the relations between the components must be determined.

Section 4.4.1 gives a closer look at the dockerfile instructions which are necessary to be able to decide which lines are relevant for the approach of this work. Section 4.4.2 compares two different basic possibilities to build an analyzer. Section 4.4.3, 4.4.4, 4.4.5 and 4.4.6 depict challenges which occur due to peculiarities of dockerfiles.

4.4.1 Docker Instructions

The instructions of dockerfiles determine how the content has to be interpreted. Docker provides a detailed overview over all instructions [Doc19b]. This overview is summed up in Table 4.1. The last column indicates, if this instruction has a potential influence on the software layers of the docker image. The influence can be direct or indirect. An indirect influence is, for example, a definition of a variable, which indicates a software version of an installation. The installation itself is a direct influence.

Instructions without a potential influence on the software layers of the docker image do not need to be considered further. The remaining instructions must be checked to decide if they are relevant for the analyze for discovery and extraction of deployment components. The instructions *CMD*, *ADD*, *COPY* and *ONBUILD* are not relevant:

- CMD: The executable given by this instruction is called at runtime at the start of the container, not at build time. The purpose of the instruction is to define the start-executable together with arguments [Doc19b]. It is possible to add a software component to the docker image, because it can execute any executable, but it is not intended to do that.
- ADD and COPY: With these instructions it is possible to load and save software components as files to the filesystem of the image. These files are not installed or marked as executables. If an executable file is copied by one of this instructions it has to be marked as executable by a *RUN* instruction afterwards. In this case this instruction needs to be analyzed.
- ONBUILD: This instruction allows the definition of trigger instructions, which are execute when this image is used as base for another image, that means that other instructions which might be relevant for the approach of this work are indirectly executed by this instruction. As this is not relevant for the docker image defined in the current dockerfile, but is relevant for another later docker image, the instruction does not need to be analyzed for the current dockerfile.

The remaining instructions are relevant for the analysis. A closer look at them is given in Section 5.2.2.

Instruction	Description	Potential Influence on Software Layers?
FROM	Defines the base image on which this image builds on	YES
RUN	Adds new layers of software on top of the base image	YES
CMD	Provides defaults for the execution of the container	YES
LABEL	Adds metadata	NO
MAINTAINER	Sets the author field	NO
EXPOSE	Specifies network ports	NO
ENV	Sets environment variables	YES
ADD	Copies (remote) files and directories to the filesystem of the container	YES
COPY	Copies files and directories to the filesystem of the container	YES
ENTRYPOINT	Defines the start point of a container, which is used as executable	NO
VOLUME	Creates mountings	NO
USER	Sets user properties	NO
WORKDIR	Sets the working directory	NO
ARG	Sets variables, which can be defined as argument at build-time	YES
ONBUILD	Adds triggers, which are executed when the image is used as base for another image	YES
STOPSIGNAL	Sets the exit system call signal	NO
HEALTHCHECK	Defines possibilities for the container to check that it is still working	NO
SHELL	Sets the default shell	NO

Table 4.1: Overview over docker instructions

4.4.2 Comparison between Generic Argument Analyzes and Command-Specific Analyzes

Software components of docker images are added using command line commands. Therefore, all possible commands can occur in a dockerfile. To be able to analyze them two solutions are possible: (i) generic argument analyzes which analyze the files based on a heuristic and (ii) command-specific analyzes which analyze based on the syntax of the analyzed commands. Both have advantages and disadvantages.

A generic approach has the advantage to be prepared for every possible command. It uses a heuristic interpretation of the given commands. Different command-packages offer the possibility to install software components. Single packages often have several possible syntax variants of the same logic.

Listing 4.2 Syntax overview for *npm install* [npm19]

```
npm install (with no args, in package dir)
npm install [<@scope>/]<name>
npm install [<@scope>/]<name>@<tag>
npm install [<@scope>/]<name>@<version>
npm install [<@scope>/]<name>@<version range>
npm install <git-host>:<git-user>/<repo-name>
npm install <git repo url>
npm install <tarball file>
npm install <tarball url>
npm install <folder>

aliases: npm i, npm add
common options: [-P|--save-prod|-D|--save-dev|-O|--save-optional] [-E|--save-exact] [-B|--save-bundle] [--no-save] [--dry-run]

```

Listing 4.3 Example of incorrect command syntax

```
apt-get [Option(en)] install PAKET1 [PAKET2]

apt-get install -y python-psycopg2

```

For example, Listing 4.2 shows the syntax variants, aliases and options for *npm install*. Furthermore, different operating systems further increase the amount of possible commands. A heuristic analyze capture all of them.

Besides the large amount of different possible command-packages a command used in a dockerfile might have an incorrect syntax, but still is correctly interpreted by the runtime of the command-package. An example is shown in Listing 4.3: Line one depict the correct syntax [Deu19] with options located in front of *install*. Nevertheless, the command in the third line with the option *-y* located behind *install* is interpreted correctly by the apt-get runtime. A heuristic is not reliant on the correct syntax and can analyze incorrect commands.

Command-specific analyzes need an own implementation for every command-package and have to consider every syntax variant. This means not only high, but also recurring effort in case of new or changing command-packages. This is comparable to the problem with application specific automatic discovery [MDW+00] as described in Section 3.2. But it is easier to develop an analyzer for specific well-defined and consistent command-packages then a heuristic analyzer, which have to deal with every possible command and still should deliver a good quality. It depends on the amount of command-packages that need to be added for the command-specific analyzer which approach needs lower effort to be developed.

There are many package-managers available. Some of them have a wider distribution than others. Support of the most widely used package-managers can be already sufficient to cover most installation commands in dockerfiles. If this assumption holds is checked in Section 6.2.

Listing 4.4 Potential false-positives for a heuristic

```
apt-get download install-lib

npm config set installConfigValue 5

yum list install

install --help

```

Listing 4.5 Potential false-negatives for a heuristic

```
npm i elixir

npm add foundation

yum localinstall wget

```

The problem of application specific automatic discovery [MDW+00] mentioned above is solved by Machiraju et al. with generic application discovery. They have one major difference to this work: They can make use of additional available standardized data sources like the operating systems which hosts the applications [MDW+00] or network traffic with default protocols of the applications [CZMB08]. This work can not make use of any additional standardized data sources, because the dockerfiles and the commands included in them are the only available data source. This results in two major drawbacks of generic argument analyzes: (i) a high false-positive rate and (ii) a high false-negative rate as described hereinafter.

Heuristic approach results are not free of mistakes [CW01]. They inherently produce false-positives as well as false-negatives. Listing 4.4 shows four examples of potential false-positives. Command one downloads the package *install-lib*, but does not install anything. The second command sets the configuration value *installConfigValue* to *5*, but do not install anything, too. The third command checks for the existence of the package *install*. The last command opens the help for the linux package *install*. The third and the last commands do not install anything either, although all of the four commands contain the word *install*.

By applying very strict rules to avoid false-positives instead a higher amount of false-negatives may occur. Listing 4.5 shows three potential false-negative examples. The first and second command both use aliases of *install*. The third command uses a special install command of yum.

To ensure RQ5 “Robustness” this work uses small parts of generic analyzes inside of command-specific analyzers. This combines the advantages of both generic and command-specific solutions: Accurate results together with the possibility to interpret syntactical incorrect commands. The approach can deal with misleading content, because it pays attention to the syntax of commands.

4.4.3 Resolve Docker Images Referrals

The references between dockerfiles are given by the name of the referenced docker image. The names of dockerfiles and docker images can be freely chosen by the authors of them. Every name and therefore also false or misleading names are possible. By resolving the chain of dockerfiles and analyzing the base images it is possible to ensure the correctness of the names of the underlying images, because the dockerfiles of the base images can be analyzed. Then it is not necessary to rely on correct names.

The names given by FROM instructions are referrals to images located at docker image libraries. One example is Docker Hub[2], which is the largest available service. This service only contains the docker images, but not the underlying dockerfiles. References to the dockerfiles are not given. This makes it impossible to resolve the dockerfile chains and check for correct naming.

A similar problem exists for ONBUILD instructions in base images. A docker image name can correctly represent its behavior as top image. But if it contains an ONBUILD instruction, which is triggered when the image is used as base for another image, the behavior can change. This might not be represented by the name. This approach cannot consider this because the dockerfile of the base image with the ONBUILD instruction cannot be resolved.

4.4.4 Multi-Stage Builds

Docker offers the possibility to add several image definitions to one dockerfile. This is recognizable at multiple FROM instructions in one dockerfile. The result of a multi-stage build dockerfile is nevertheless only one image. Only the last part of the complete dockerfile, which starts at the last FROM instruction, is used for the docker image. This is called final image of a multi-stage build dockerfile. The other parts can be used in the final image as additional component [Doc19a].

The relationship between the additional multi-stage component and the other components of the final image are not clear without further information like documentations. Artifacts of an earlier build stage can be copied and reused in a later build stage, but they do not have to [Doc19a]. Furthermore, multi-stage builds can be stopped at every build stage. This is determined in the build command and can not be retrieved from the dockerfile.

Listing 4.6 shows an example of a mutli-stage build dockerfile. Lines seven to eleven result in the final image. This image uses the first image defined in lines one to five. It copies *app* in line ten, which was created in the first image. In the build-command, which builds an image out of the dockerfile, it can be determined to only build the first image (“gobuilder”) and ignore the second one.

It is difficult to analyze and discover the relations between the single stages of multi-stage build dockerfile due to the different possibilities to reuse or not reuse artifacts of previous stages. Besides, the complete dockerfile inclusively the final image might be only intended for special cases. An example is a debug build, which depend on a special execution environment. This information can not be retrieved form the dockerfile, but from scripts or usage instructions. Because of this, this approach consider the single stages of multi-stage builds as independent docker definitions.

[2] https://hub.docker.com/

Listing 4.6 Multi-Stage Build example

```
1  FROM golang:1.12.4-stretch as gobuilder
2  WORKDIR /go/src/
3  RUN go get golang.org/x/net/html
4  COPY app.go    .
5  RUN CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o app .
6
7  FROM ubuntu:18.04
8  RUN apt-get install -y ca-certificates
9  WORKDIR /root/
10 COPY --from=builder /go/src/app .
11 CMD ["./app"]
12
```

Listing 4.7 Comment examples in a dockerfile

```
1  FROM ubuntu:14.04
2  # Ubuntu 14.04 LTS
3  RUN apt-get install curl
4  # RUN apt-get install python
5  RUN echo '# python not installed'
6
```

4.4.5 Comments

Comments are marked with a hash symbol in dockerfiles. These lines must be ignored. It is necessary to check if the hash symbol is located at the first position of a line, otherwise it is not a comment. Arguments of instructions like commands may contain hashes at any other position of a line except the first character. In Listing 4.7 lines two and four are both a comment and have to be ignored, although line four is a valid *apt-get* command. Line five does not include a comment, the hash is part of the *echo* command.

4.4.6 Identification of Components

Many installation commands offers different syntax possibilities. The possibilities for *npm install* are shown in Listing 4.2. The variants, which take the name of the component are easy to analyze: The name is obviously given. The variants that take URLs and git-links are much harder to analyze. The name has to be extracted out the URL. Often the last part of a URL can be taken as name. This applies to the first command in the example of Listing 4.8. However, the second and third command in the example show, that this does not apply always. The second command installs the express version of the *abcWebserver*. The third command installs a specific release version of *apache-maven*. In both cases the name is given in the second last part of the URL.

Listing 4.8 Examples of URLs in *npm install* commands

```
1  npm install http://www-eu.apache.org/dist/maven/maven-3/3.3.9/binaries/apache-maven-3.3.9-
   bin.tar.gz
2
3  npm install git+ssh://git@github.com/abcWebserver/express.git
4
5  npm install http://www-eu.apache.org/dist/maven/binaries/apache-maven/release-3.3.9-bin.tar.
   gz
6
```

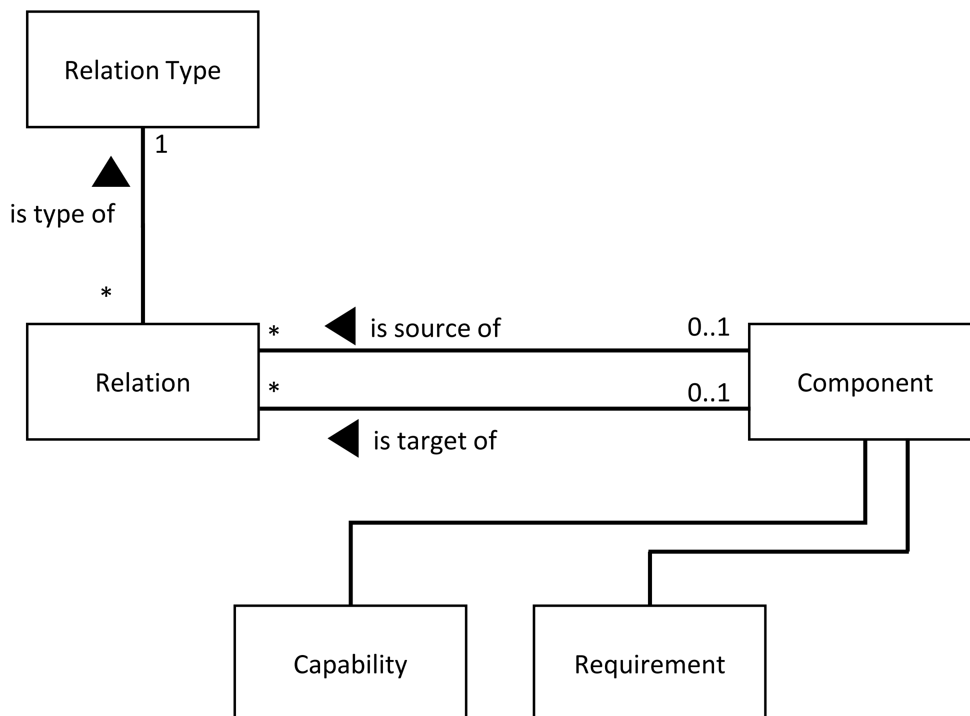

Figure 4.5: Metamodel of this approach's model

4.5 Model

The model part is the last part of the discovery approach of this work. The results of the analysis need to be structured and depicted in an understandable way to fulfill RQ6 "Preparation of Results". For a technology independent and generic representation, a metamodel defines how the correlations between several analyzed components are modeled. The model defined by this metamodel can be classified in the application architecture domain, because it needs to show explicit software components and their logical relations without any detailed features.

Figure 4.5 shows the metamodel of the data structure used to save the results of this work. Every discovered software component is modeled as *Component* element. A Component can have two different types of *Relations*. One type is a Relation to a base Component ("is target of") and one to a top Component ("is source of"). If an *Nginx* webserver is installed on an *Ubuntu* the Ubuntu-component is source of the Nginx-component. The Ubuntu-component has a relation of the type "is source of" to the Nginx-component. The nginx-component has a "is target of" relation to the Ubuntu-copmonent.

A component contains *Requirements* which define the needs of this component and therefore specify the possibilities of the "is target of" relation. The Nginx-component has a requirement "Ubuntu" as it needs an operating system to be executed. The Nginx-component can have more requirements, for example *Debian*, which are OR-associated. That means, not all requirements need to be fulfilled. A component has also a *Capability*, which defines the service this component offers to other components. This service is always the component itself. The Ubuntu-component offers the service "Ubuntu".

Figure 4.6 shows an example model of this approach. Each component is depicted as rectangle. In each component the capability of this component is located at the top, the requirements are located at the bottom. In this example *Voting App* of version *1.0* has two requirements: It can run based on *Apache HTTP Server* with a version greater than *2* as well as based on *Nginx* with a version greater than *0.8*. The *Apache HTTP Server* of version *2.4.35* has the matching capability to serve for *VotingApp* and can run on *Ubuntu* with at least version *14.04*. *Nginx* with version *1.15* has also the capability to serve for *VotingApp* and it has the requirements *Ubuntu* greater version *16.04* as well as *Debian* greater version *9*. *Ubuntu* has the capability to serve for *Apache HTTP Server* and *Nginx*. *Debian* has only the matching capability for *Nginx*, because *Apache HTTP Server* needs a *Debian* with version greater *9.8*.

This metamodel can be mapped to Topology Orchestration Specification for Cloud Applications (TOSCA).

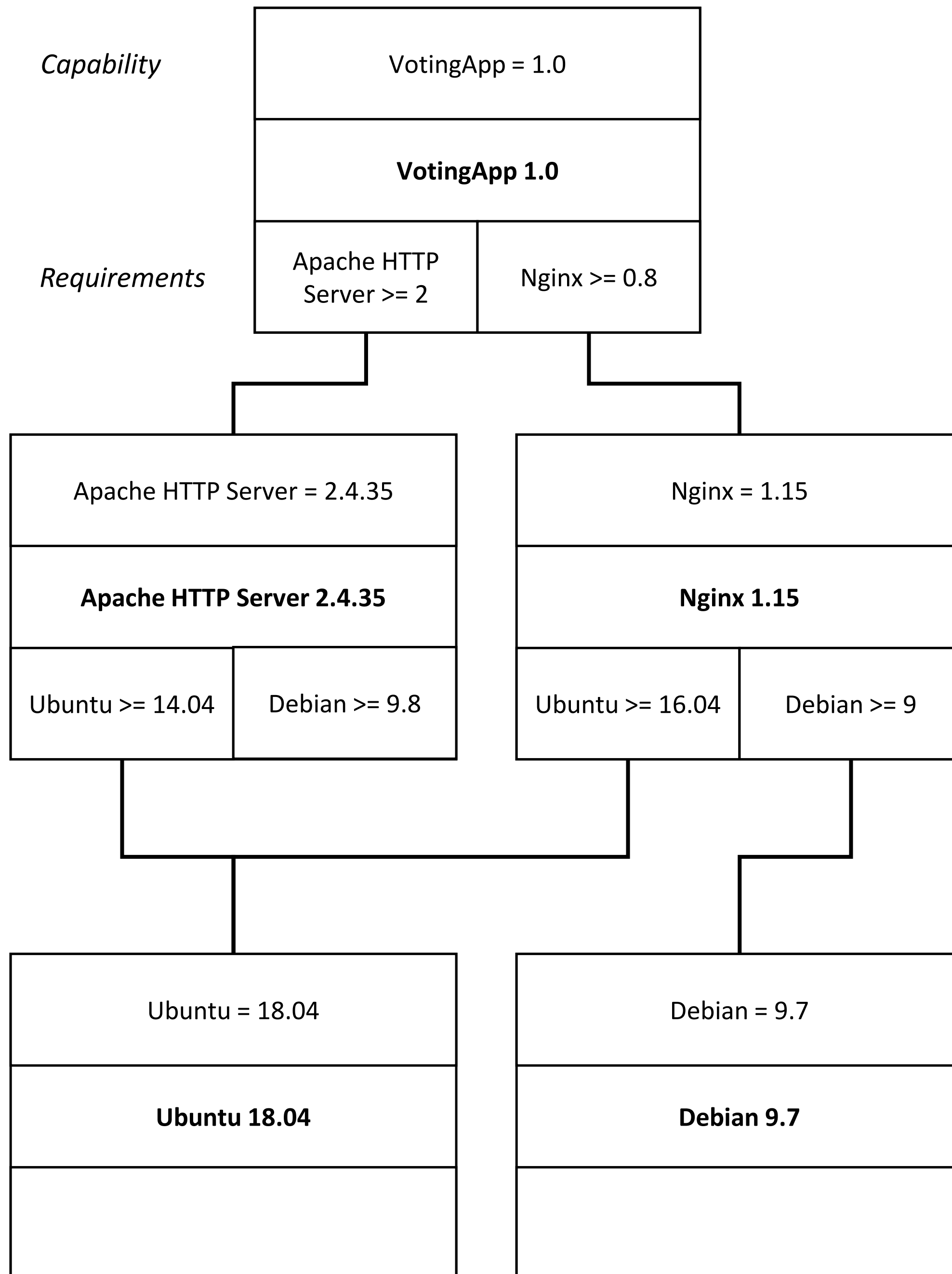

Figure 4.6: An example model of this approach

5 An Approach for Automated Discovery Based on Dockerfiles

This chapter describes the concept and solutions of the discovery approach of this work based on Chapter 4. Important solution details are presented.

The solution is built upon the algorithm presented in Section 4.2. The steps of the algorithm are bundled into detached parts. This ensures the capability of all parts to operate independent of each other. In addition, it allows to create different numbers of instances of the parts to scale them independently. This ensures RQ2 “Scalability”.

Section 5.1, 5.2 and 5.3 depict the solutions of the three parts *Crawler*, *Analyzer* and *Model*. Subsequently, Section 5.4 describes the complete and detailed algorithm consisting of all three parts of the discovery approach of this work.

5.1 Crawler

The crawler fulfills RQ3 “Collection of dockerfiles”. It collects the dockerfiles to be able to analyze them. This needs to be done automatically to fulfill RQ1 “Automated process”. Section 5.1.1 and 5.1.2 outline two architectural details of the crawler of the discovery approach of this work. Section 5.1.3 describes the crawler solution of this approach.

5.1.1 Splitting of Crawling and Accessing of Dockerfiles

Many web services restrict the access to their infrastructure to a specified number of requests in a specific time interval. This is known as rate limit. The service providers protect their infrastructure against load peaks and Denial-of-Service attacks. A crawler needs to limit the request to comply with the rate limit. In contrast, the further processing of the crawled results is done as fast as possible. To detach both operating speeds and optimize the usage of the given rate limit of different service providers both processes, crawling and the subsequent processing, have to be split and detached.

The concept is designed with a crawler management class and crawler instances and is visualized in Figure 5.1. A crawler instance is specific to a web service and takes care of the rate limit of this service. It operates on the maximum possible speed based on the rate limit. All results are written into a buffer of the crawler management class which delivers single dockerfiles on request. This offers the possibility to completely detach the time of crawling and processing of the dockerfiles. To further support this, all dockerfiles are stored on the filesystem and can be read by the “LOCAL” crawler-type at any later time.

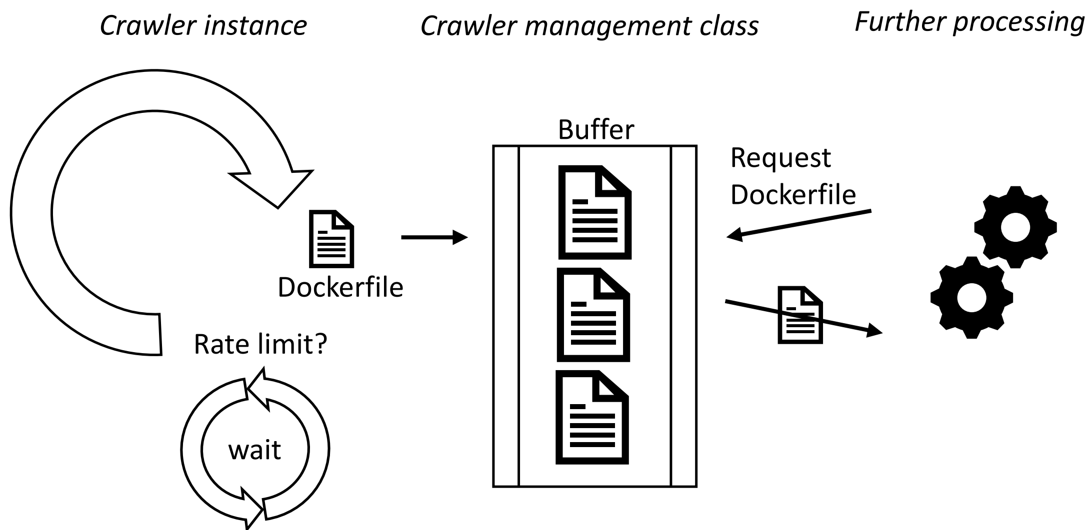

Figure 5.1: Split crawler functionality to comply with rate limits

5.1.2 Different Crawler-Types

As stated in Section 4.3 dockerfiles are a specific content to crawl. A page-specific crawler fits the best for this purpose, because many web services do not offer this type of content at all.

RQ4 "Expandability" states, that the approach must be extendable. New services which currently may not be known must be able to be supplemented. As a result, it is necessary to offer a crawler template in the implementation, which can be extended with any service-specific crawler.

This can be done by an interface for a crawler implementation, which defines a method to crawl dockerfiles. It returns a list of dockerfiles. This complexity is hidden to any user of the crawler. A crawler management class receives the type of the crawler which the user of the crawler wants to use. It creates the specific crawler instance and offers the dockerfiles to the user independent of the used crawler implementation.

5.1.3 GitHub Crawler

One possible source for dockerfiles is the open source web platform GitHub[1]. GitHub hosts a large number of projects and many of them include dockerfiles. GitHub offers a Representational State Transfer (REST) API to access their content programmatically. The API does not include any function to find dockerfiles directly. The access has to be split into several steps. They are visualized in Figure 5.2 and described in the following:

[1] https://github.com

1. **Get a repository**: Every project is called git-typical repository. No specific repository is needed, instead an iteration over all repositories meets the requirement RQ3 “Collection of dockerfiles”. All repositories are numbered by a consecutive number. However, not all repositories are publicly available. The API offers a function to get a reference to the next public available repository based on the last requested repository. This is the first step of the *GitHubCrawler* in the figure (“getNextRepository”).
2. **Search through the repository for dockerfiles**: With the reference to a repository it is possible to search for files in it. A search expression is passed to the API which includes the filename “Dockerfile”. This API request responds the number of found files with references to them as well as download URLs. This is the second step of the *GitHubCrawler* in the figure (“search(‘Dockerfile’, Repository)”).
3. **Download dockerfiles**: With the given URLs all found dockerfiles are downloaded as text. This is the inner loop of the figure.

The discovered dockerfiles are saved. These steps are repeated until a predefined number of dockerfiles is discovered. The actual count of discovered files can be higher, because all dockerfiles from a repository are downloaded, before the count is checked. The actual count is returned to the caller (*Discovery Process* in the figure). The dockerfiles can be retrieved one by one.

This approach only searches through the filenames of the repositories. As depicted in Section 4.3.2 it is also possible to search through the complete content of files. Only searching through the filenames is much simpler and decreases the amount of needed requests to the service provider significantly. The false-positive rate is smaller. As a result it decreases the execution time rapidly. Although, it can be extended to support file content searching by changing the given crawler or adding additional crawler.

GitHub API is subject to a strict rate limit as described in Section 5.1.1. It offers two modes with different rate limit thresholds: A non-authenticated and an authenticated mode. For non-authenticated requests only 60 requests per hour are allowed. The threshold for authenticated requests is much higher. Up to 5000 requests per hour are allowed. Authentication is performed with a user account of GitHub.

Search requests are treated separately. They are causing particularly high load on the infrastructure of the service. That is why GitHub enforces a better temporal distribution. Only 10 requests are allowed per minute for non-authenticated requests and 30 requests per minute for authenticated requests.

To comply with the rate limit of the service an implementation of this approach has to deal with the thresholds. Every response of the GitHub API contains the current open rate limit windows. The GitHub crawler has to read both values for the normal and the search rate limit from every response and pause the requests if necessary. This corresponds to the third solution of Section 4.3.1.

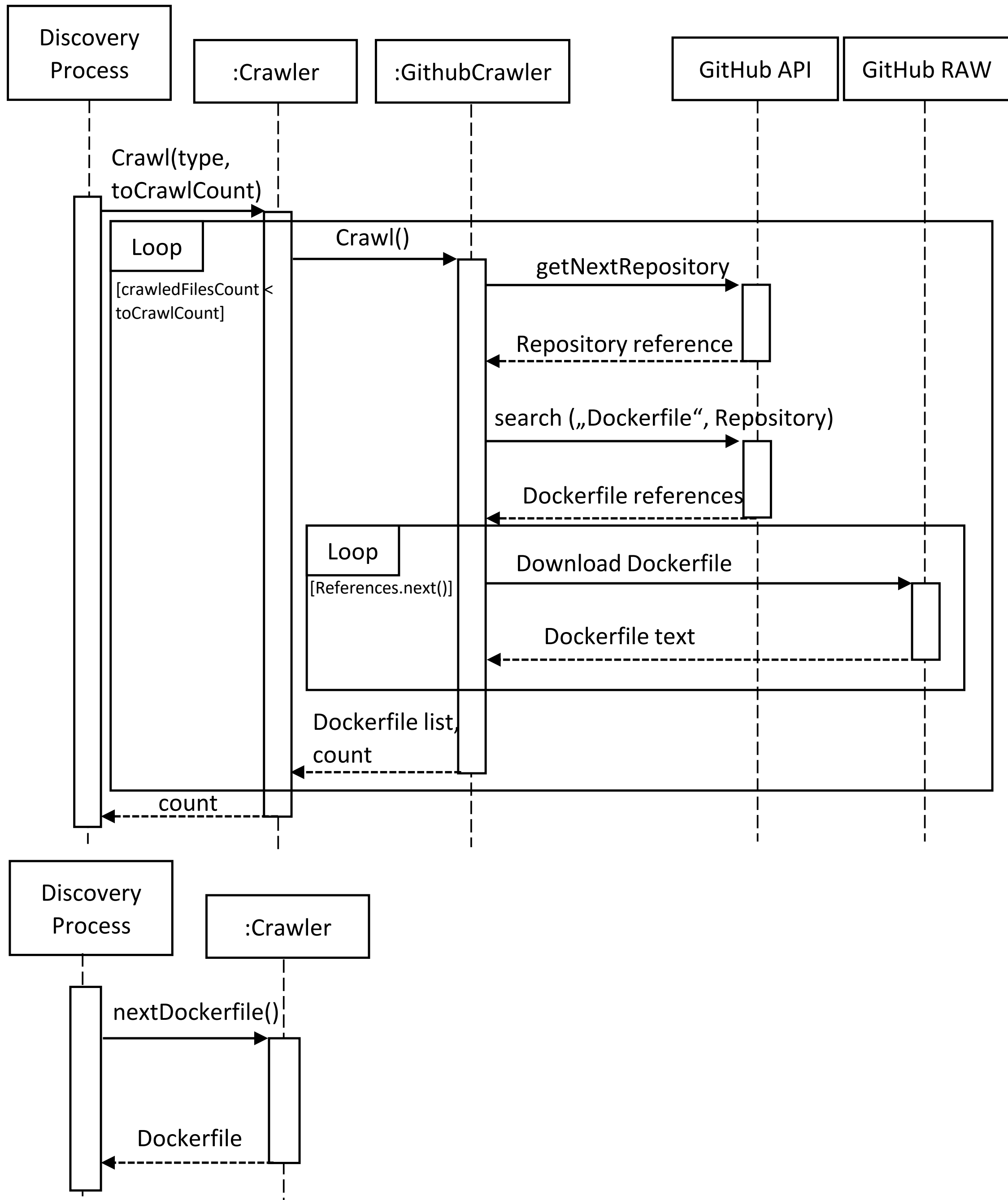

Figure 5.2: Sequence diagram of the GitHub crawler implementation

Listing 5.1 Two examples of multi-line commands

```
1  RUN apt-get install libsm6 \
2  libxrender1 \
3  libxext-dev \
4  python3-tk
5
6  RUN apt-get update -qq -y \
7  && apt-get install -y libsm6 libxrender1 \
8  && apt-get clean
9
```

5.2 Analyzer

The analyzer is responsible for extracting the relevant parts of the dockerfiles. This needs to be done automatically to fulfill RQ1 “Automated process”. RQ5 “Robustness” always have to be considered, misleading comments or other misleading content must be filtered out. This discovery approach uses a command-specific analyzer with small parts of generic-analyzes inside as outlined in Section 4.4.2.

Figure 5.3 shows the activity diagram of the analyzer. At first, the Pre-Analyzer prepares the dockerfile. The Pre-Anylzer is described in Section 5.2.1. Then the software component defined by the FROM instruction is extracted. After that, all other relevant lines are analyzed. The decision as to which lines are relevant is outlined in Section 5.2.2. Depending on the command found in a line the corresponding command-specific analyzer is used. Section 5.2.3 describes them in detail. The results are combined as top components together with the base component from the FROM line.

5.2.1 Pre-Analyzer

To be able to analyze individual docker instructions several preparation tasks need to be done. This is depicted in an activity diagram in Figure 5.4. First, the dockerfile needs to be split into single instructions. Usually a line equals a logical line in a dockerfile. However, it is possible to extend a logical line with a backslash at the end of a line. An example is given in Listing 5.1: The first four lines are one logical line. The other three lines are also one logical line, independent of the three commands, which are merged into one line based on Linux-syntax. Nevertheless, every command needs to be treated separately, that is why these three commands are split into three lines afterwards. Additionally, the Pre-Analyzer filters out all comment lines and blank lines. A comment line is marked with a hash at the first character of the line.

In case of multi-stage build dockerfiles all stages need to be split into single dockerfiles. The reason for that is explained in Section 4.4.4. This is done by splitting at every FROM instruction.

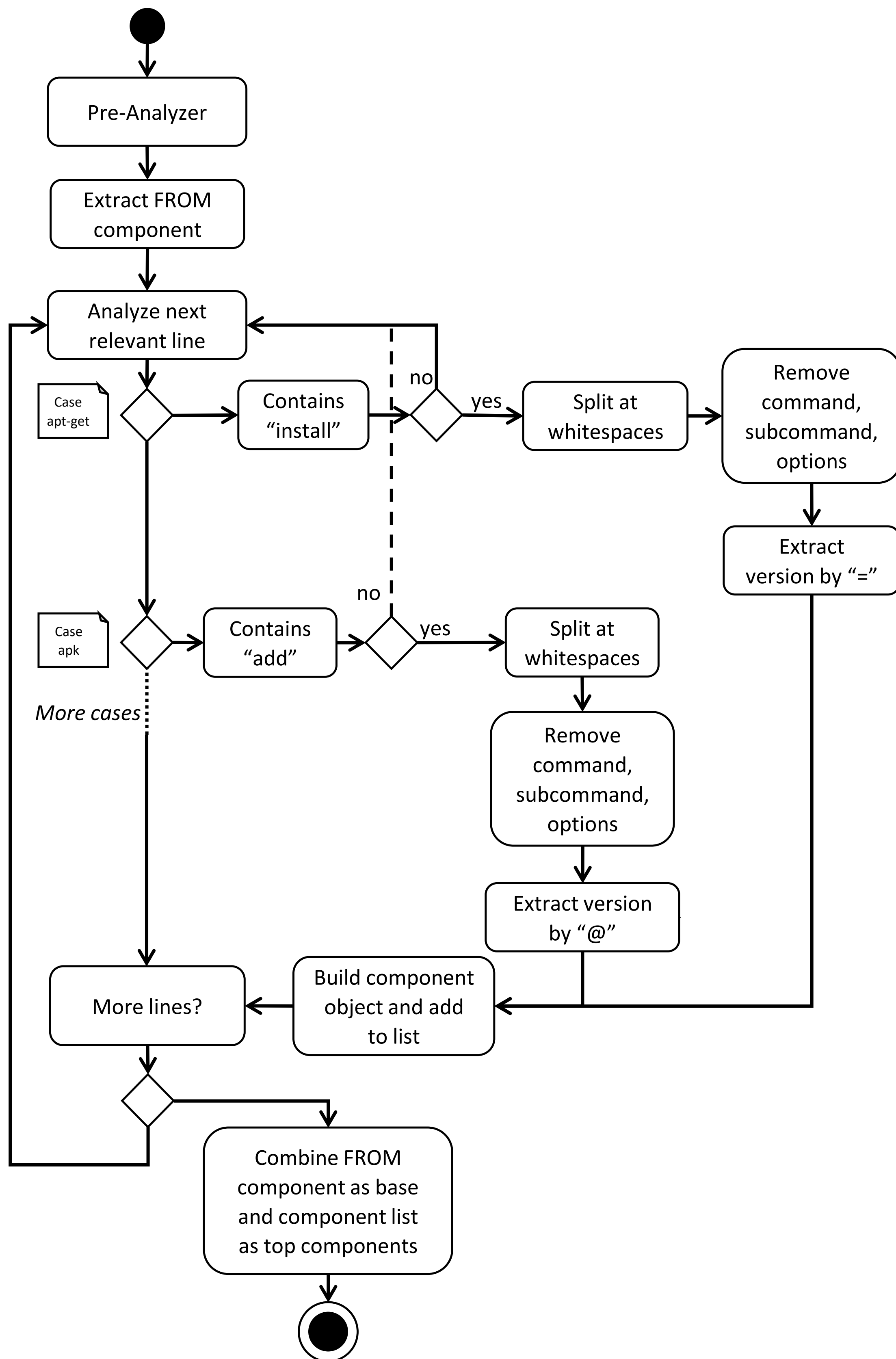

Figure 5.3: Activity diagram of the analyzer implementation

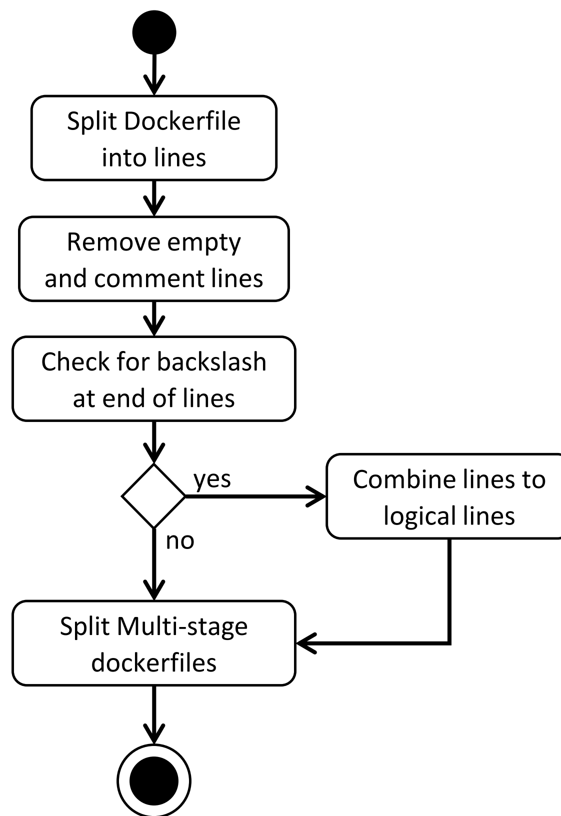

Figure 5.4: Activity diagram of the Pre-Analyzer part

Listing 5.2 Syntax variants of FROM instruction [Doc19b]

```
FROM <image> [AS <name>]
FROM <image>[:<tag>] [AS <name>]
FROM <image>[@<digest>] [AS <name>]

```

5.2.2 Relevant Parts of Dockerfiles

To be able to decide, which parts of a dockerfile are relevant, all docker instructions need to be known. They are described in Section 4.4.1.

To outline the application dependencies the “based on” information of dockerfiles, which defines the base image, can be used. This information is given by the FROM instruction of a dockerfile. The instruction is followed by the name of the docker image as parameter and optionally the version of it. The possible syntax variants are listed in Listing 5.2.

The second relevant part of dockerfiles as source for information for application dependencies are the installation commands, which can add additional software components to a docker image. This information is given by the RUN instructions. They contain shell commands which are executed on top of the image defined by the FROM instruction. Relevant for application dependency analyzes

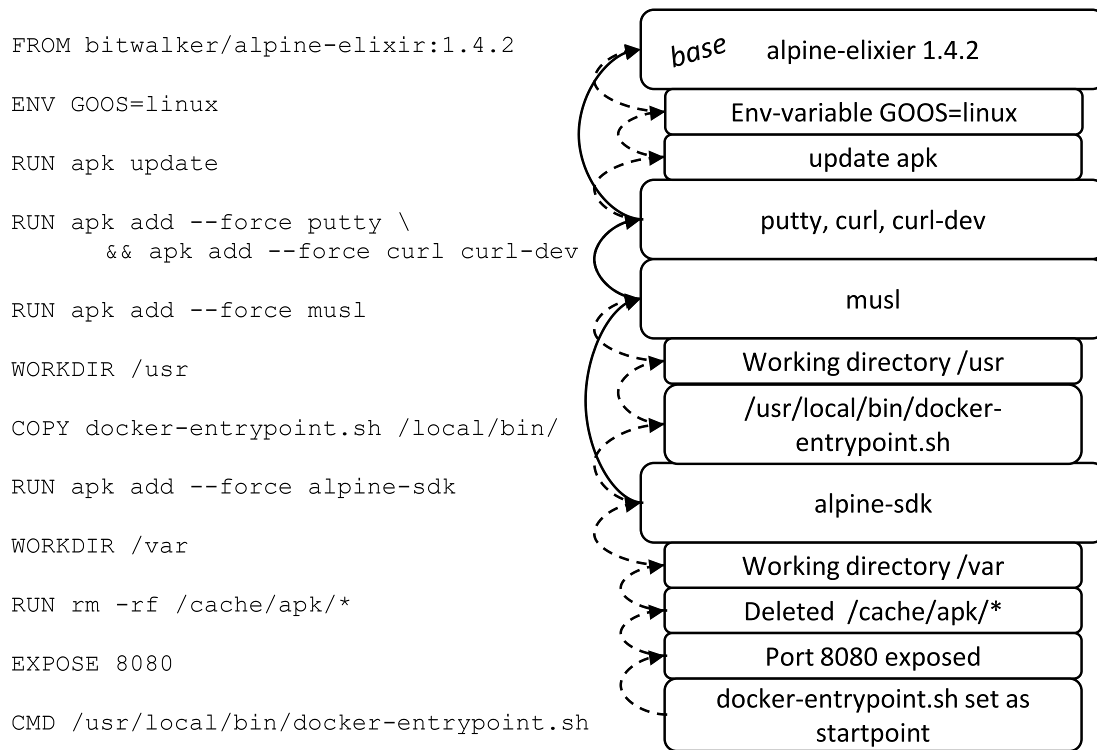

Figure 5.5: Docker image layer associated to the corresponding dockerfile instructions

are any kind of installation commands like package-manager commands. Examples are *apt-get*, *pip3* and *yum*. However, only install instructions are useful for the analysis, therefore other instructions of the package-managers like *remove* or *clean* need to be filtered out. In addition to packet manager commands other possibilities to define executables are interesting. An example is *chmod +x*, which marks a file as executable which is done in a RUN instruction, if it is intended to execute this file in the docker image.

As every instruction generates a new layer in the docker stack [Doc19a] a RUN instruction builds upon a previous RUN instruction. The example in Figure 5.5 shows for every dockerfile instruction the corresponding image layer (same height). For this approach relevant layers are bigger. The dotted arrows show their technical links, the solid arrow the resulting technical relations between installed software components. In a logical consideration these relations are not all correct, since some of the software components are independent of others. For example, *alpin-sdk* does not require *putty*, *curl*, *curl-dev* and *musl*. Docker images have a strict layer structure, it exist no possibility to add a software component not based on the previous layer, also if the components are independent. It is not possible to analyze automatically without further knowledge of the corresponding software components if two components are actually dependent on each other or not. Therefore, this approach does not add any relations between components extracted out of RUN instructions, but builds relationships only between the image base (FROM instruction) and the installed software components (RUN instructions).

Listing 5.3 ARG variable definition with and without default value

```
1  FROM busybox
2  ARG versionLibA
3  ARG versionLibB=2.5.3
4  RUN apt-get install libA=$versionLibA
5  RUN apt-get install libB=$versionLibB
6
```

ARG and ENV instructions influence FROM instructions. Additionally ARG influences RUN instructions, but not ENV. Both define variables, that may include version names and numbers of components used in FROM and RUN instructions. The variables must be resolved if possible. It is impossible to resolve an ARG variable, if it has no default value, but is only defined at build time as parameter. In Listing 5.3 *versionLibA* does not have a default value, while *versionLibB* does have a default value. The variable in line four cannot be resolved, the variable in line five can be resolved.

5.2.3 Component Extraction

This section describes how the software components are extracted out of docker instructions.

The syntax of FROM instructions is well-defined [Doc19b]. The component name is given behind "FROM". The version is optionally given behind a colon or an at-sign. An alias is optionally given behind an "AS", but is ignored by this analyzer.

The syntax of RUN instructions is much more complicated. RUN takes every command and tries to execute it on top of the underlying image. Therefore, anything can be given as parameter for a RUN instruction. This includes not only installation commands which are relevant for this approach, but also every possible other command. To only analyze installation commands and interpret them in the correct way, every command-line tool, that may install something, need an own sub-analyzer as outlined in Section 4.4.2. To fulfill RQ4 "Expandability" the discovery approach of this work contains an abstraction of a sub-analyzer. Behind this abstraction sub-analyzers can be added. The sub-analyzers have similar logic implemented, but differ in details, because the syntax of every installation command differs.

The discovery approach of this work already contains some relevant command-line tools, mainly package-managers:

- **Apt-get**: This is a widespread package-manager used amongst others by Debian and Ubuntu. The relevant sub-command is *apt-get install*. Optionally options are given, which are led by a minus. They are ignored by this analyzer. Afterwards the components, that are installed, are given separated by white spaces. A version can be given after an equal sign.
- **Pip3**: This is the default installation package for python packages. The syntax is very similar to apt-get, but can also contain URLs and paths to version control systems, additional files and archives. An interpretation of those is waived (Section 4.4.6). Versions can be given at the end together with version operators.

- **Yum**: This is a package-manager used, for example, by RedHat. The syntax is comparable to apt-get, but several sub-commands are relevant for installation: *install*, *localinstall*, *groupinstall* and *langinstall*. A version can be given after a minus.
- **Npm**: This is a packet-manager for the JavaScript runtime-environment node.js. The *install* sub-command has two aliases: *i* and *add*. Both have to be considered. The components are separated by white spaces. A version can be given after an at-symbol.
- **Apk**: This is a packet-manager for Alpine Linux distribution. It has a uncommon install sub-command: *add*. The components are separated by white spaces. The version-operator is an at-symbol.
- **Chmod**: The access and right administration tool can be used to mark a file as executable. This is displayed by the option *+x*. Other options are ignored by this approach. The file is given after the options.

The names and versions of discovered software components are read and used to create component objects. The component objects are saved to process them further. All software components extracted from RUN instructions in a dockerfile are saved with a relationship to the component extracted from the FROM instruction from the same dockerfile. The component from the FROM instruction is the base component for all components from the RUN instructions.

Download and archive extraction tools like *wget* or *tar* are not considered. Usually it is necessary to mark files after downloading and extracting as executable by *chmod*, which is already considered.

5.3 Model

The approach saves the discovered components in an internal format, which assigns a list of components together with their frequency to the corresponding base components. This assignment represents the relation between a base component and the components on top of it.

The data from the internal format can be retrieved according to the metamodel of Section 4.5 to fulfill RQ6 “Preparation of results”. Every component is represented by a node. The nodes does not include any references on each other, but include requirements and a capability. Requirements and the capability represent the possible relations between the components.

5.4 Algorithm

Algorithm 5.1 shows the overall algorithm as pseudocode. It is explained in the following paragraphs. The functionality details of the single parts are not included in this pseudocode.

The discovery approach of this work operates asynchronous and based on rounds. It never terminates by its own. After one round of the crawler finishes, another round is triggered automatically. At the same time the already found dockerfiles are analyzed. This enables the crawler to permanently crawl while at the same time the analyzer can process the previously found dockerfiles. One crawler

round is over when a predefined number of dockerfiles is found. The automatic start of a new round can be prohibited by calling a stop method. The approach stops then after the current crawler round finishes.

The algorithm (Algorithm 5.1, Figure 5.6) starts with a check, if it was stopped. If not, the crawler starts its work. It retrieves the next repository. In this repository all dockerfiles are searched and downloaded. If at least one dockerfile was crawled, it is added to the list of already crawled files of this crawler round. As long as the predefined amount of dockerfiles are not crawled, the crawler continues. In this example, this number is set to *5*.

After the crawler finishes, the next round is triggered.

Simultaneously, the analyzer starts to process all previously found dockerfiles. It first has to retrieve the first dockerfile from the crawler. The crawler returns them in the same order as it founds them. To ensure this, accesses to the list of dockerfiles by the crawler need to be synchronized. The pre-analyzer split the dockerfile into single lines. The analyzer iterates over all lines and if it finds a FROM-line (docker instruction) it is analyzed and the extracted component saved as base component. Then it is iterated over all following lines up to the next FROM line. Multiple FROM can occur in multi-stage build dockerfiles. If a RUN line is found, it is analyzed and the extracted components are saved as top components. Between the found base component and the top components a relationship exist. This is saved into the approach's internal result format. The analyzer continues as long as more dockerfiles from the crawler round are available. Dockerfiles, which might be found from the next crawler round while the analyzer processed the previous files are not analyzed. They are treated by the analyzer of the next round. The number of dockerfiles of one round might be higher than the predefined amount of dockerfiles of one round, because the crawler always crawls all files from one repository before it checks, if it found enough files for the current round.

All found component relations are saved and can be retrieved according to the metamodel afterwards.

Algorithm 5.1 Algorithm of this approach

```
procedure DeployableComponents.execute(crawlerType)
    if isNotStopped then
        while crawlerCount < 5 do
            newRepsoitory ← crawler[crawlerType].getNextRepository()
            fileList ← crawler[crawlerType].searchRepoForFile("Dockerfile")
            if fileList.isNotEmpty() then
                dockerfiles ← Download(fileList)
                crawler[crawlerType].dockerfileList.add(dockerfiles)
                crawlerCount+ = dockerfiles.count
            else
                continue
            end if
        end while

        StartNewThread(DeployableComponents.execute(crawlerType))

        for i over crawlerCount do
            newDockerfile ← crawler[crawlerType].nextDockerfile()
            lines ← SplitDockerfile(newDockerfile)
            for line over lines do
                if line.isFromLine then
                    baseComponent ← ExtractBaseComponent(line)
                    imageLines ← ExtractLinesUntilNextFrom(lines, line)
                    for imageLine in imageLines do
                        if imageLine.isRunLine then
                            commandType ← ExtractCommandType(imageLine)
                            components ← Analyzer[commandType].analyze(imageLine)
                            topComponents.add(component)
                        end if
                    end for
                    componentsWithRelations ← BuildRelations(baseComponent, topCompo-
nents)
                end if
            end for
            componentsWithRelationsList.add(componentsWithRelations)
        end for
        listOfKnownComponentRelations.add(componentsWithRelationsList)
    end if
end procedure
```

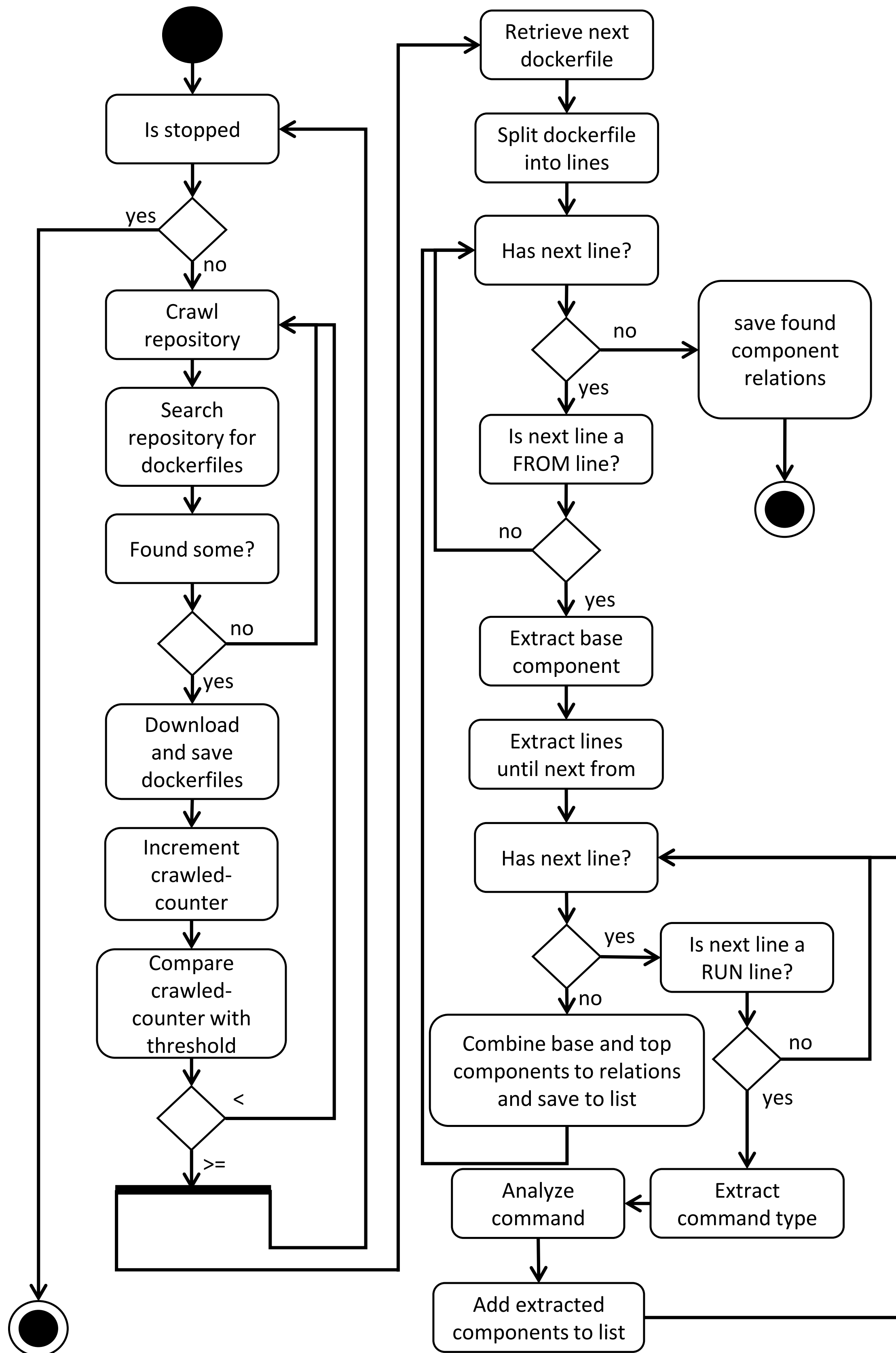

Figure 5.6: Activity diagram of the algorithm as depicted in Algorithm 5.1

6 Prototype and Evaluation

In this chapter, an implementation of the concept presented in Chapter 5 based on considerations in Chapter 4 is explained (Section 6.1), validated against the requirements established in Section 4.1 and evaluated. The validation and evaluation is presented in Section 6.2.

6.1 Extension to Eclipse Winery

The software component discovery approach of this work is implemented into the OpenTOSCA winery fork [Ope19] from the eclipse project winery [Ecl19][1]. This offers the possibility to use the TOSCA model implementation from this project for the implementation of the metamodel (Section 4.5).

The approach is implemented asynchronous, modular and extensible. Crawler module (Section 5.1) and analyzer module (Section 5.2) can be instantiated independent. Both provide interfaces for their business logic classes that can be extended to extend the support of this approach. The asynchronous algorithm (Section 5.4) guarantees optimal usage of rate limits due to a parallel execution of crawler and analyzer. The round based structure of the algorithm (Section 5.4) is implemented with threads. After the crawler finished one round a new thread with a new crawler round is started. Then the analyzer for the old round is executed in the old thread. Compared to an alternative algorithm based on two permanent threads for the crawler and the analyzer the round based algorithm has an advantage in the implementation: Communication between a crawler thread and an analyzer thread is not necessary. Crawling and analyzing is done in the same thread. This results in a simpler implementation.

6.2 Evaluation

The validation of the requirements is presented in Section 6.2.1. In Section 6.2.2 and 6.2.3 the crawler and analyzer modules are evaluated.

6.2.1 Validation of Requirements

In the following the implementation is validated against the requirements of Section 4.1.

[1]The implementation is available on GitHub: https://github.com/OpenTOSCA/winery/pull/131

Requirement 1 (RQ1): Automated process. All three parts of the implementation (Crawler, Analyzer, Model) operate fully automated. Especially the complicated part, the Analyzer, does not need any manual help to discover and extract software components. The algorithm ensures the automated processing of all three parts as a chain. The requirement is fulfilled.

Requirement 2 (RQ2): Scalability. The implementation is built modular. Crawler and Analyzer can be instantiated independent of each other and therefore scaled independent. The implementation does not contain any static or singleton parts which would prevent multiple instances. However, the algorithm of the current implementation does not intend to create several instances as it runs on one machine. Since the bottleneck of the current implementation is the rate limit (Section 6.2.2) scaling is only useful over several machines. The prerequisites for this requirement are fulfilled.

Requirement 3 (RQ3): Collection of dockerfiles. The implementation is able to gather dockerfiles from the online service GitHub and can save them on a local file system for later usage. Additional web services can be added. The requirement is fulfilled.

Requirement 4 (RQ4): Expandability. Both Crawler and Analyzer offers interfaces to add additional web services as data source respectively additional command-specific analyzers to extend the recognized component installation instructions. The requirement is fulfilled.

Requirement 5 (RQ5): Robustness. The implementation uses command-specific analyzer to avoid false-positives as well as false-negatives. Comments in dockerfiles are filtered out. The requirement is fulfilled.

Requirement 6 (RQ6): Preparation of results. The results of the analyzes can be retrieved according to a metamodel to ensure comprehensibility. The requirement is fulfilled.

6.2.2 Crawler Evaluation

Due to the rate limits described in Section 5.1.3 the crawler implementation for GitHub is the bottleneck of the implementation. That is why the operation speed of it is interesting. In the following results based on seven times one hour execution time are presented. The complete result set is given in Appendix A.

Table 6.1 shows numbers of the test execution of the GitHub crawler implementation. In seven hours 12.090 repositories were searched. The number of searched repositories per hour was constant. The minimum was 1690, the maximum 1789 repositories (Figure 6.1). The crawler returned 237 results. Two of them are false-positives: One is a documentation how to write dockerfiles, the other is a JavaScript file which processes dockerfile. Two more results are dockerfiles with invalid syntax. The other 233 results are valid dockerfiles.

In mean the crawler returned 33,9 results per hour and 1727,1 repositories were searched per hour. Although the crawler operation speed suffered from high latency from the test machine to *api.github.com* of approximately 100ms, the crawler had to wait for two-thirds of the time. The search-API rate limit was reached after 20 seconds of every minute. This rate limit resets every minute. The API rate limit was never reached. It is not possible to decrease the number of requests, because for every repository only one search-API and one API request are used. The download of found files had only little impact on the operation speed, because the latency of the downloads (*raw.github.com*) was noticeably lower. It was only approximately 10ms.

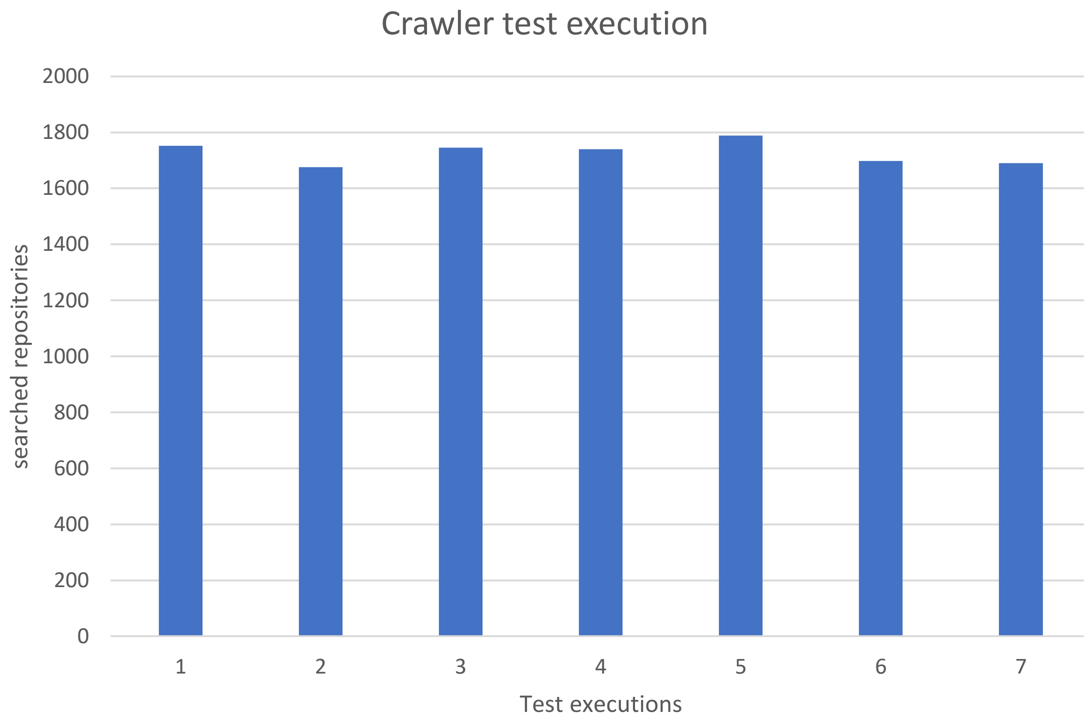

Figure 6.1: Number of searched repositories of all seven test executions

Execution time	7 hours
Searched repositories	12.090
Found files	237
False-positives	2
Dockerfiles with invalid syntax	2
Correct dockerfiles	233
Crawled files per hour	33,9
Searched repositories per hour	1.727,1
Mean number of repositories to search until found file	51

Table 6.1: Numbers of a crawler execution test

Execution time	7 hours
Covered repositories	41.148
Repositories of GitHub in total[2]	123.639.040
Covered percentage	0,0332808%
Years to search all repositories	240,1

Table 6.2: Calculations for a complete crawl of GitHub

Table 6.2 shows extrapolations for a complete crawl of GitHub. The test execution of the crawler covered 41.148 repositories. The difference to the searched repositories arises through private repositories. They cannot be searched, but are covered by the crawler, because all repositories are numbered and the crawler iterates over them based on the Identifiers (IDs). GitHub hosts approximately 124 million repositories. To crawl over all GitHub repositories the crawler would need 240 years with the operation speed of the test execution. To accelerate this, the number of machines need to be increased. However, it is not necessary to crawl all repositories of GitHub to get reliable results. The necessary number of dockerfiles as data set depends, if relationships of common software components are supposed to be discovered or relationships of uncommon components. The test execution already resulted in many relationships of common components, for uncommon components 233 dockerfiles are not enough.

6.2.3 Analyzer Evaluation

Due to the decision for command-specific analyzer and against generic analyzer the discovery approach of this work does not cover all possible installation commands. In the following the evaluation results of the 233 dockerfile results of the crawler test execution are presented.

The 233 dockerfiles contain a total of 1.269 RUN lines. If several commands are bound together with "&&" they are counted individually. 409 lines of them contain an installation command. 323 of them are covered by the command-analyzer included in the implementation. This means 78,97% of all installation commands of the evaluated data set are covered by this implementation. Figure 6.2 visualizes these results.

The total number of discovered software components is significantly higher than the number of covered installation commands, because many commands include several components for installation.

[2]Calculated based on https://github.blog/2018-11-08-100m-repos/

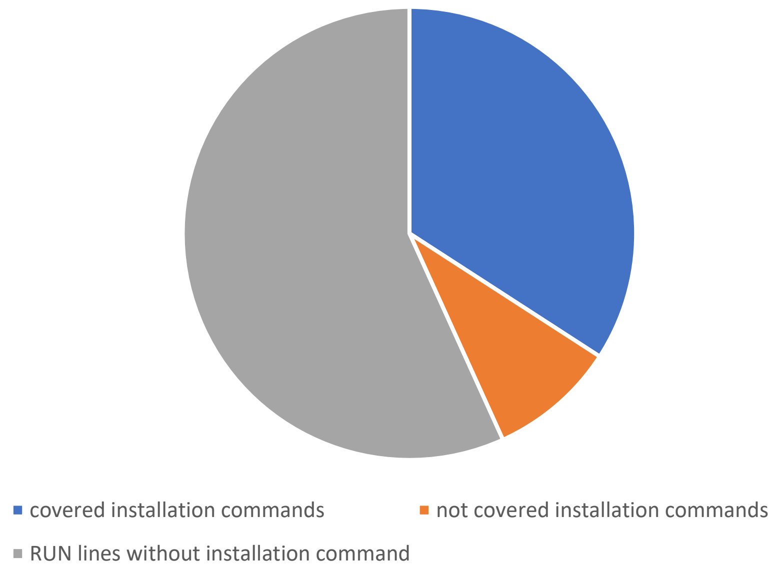

Figure 6.2: Results of the evaluation of installation commands

7 Conclusion and Future Work

This thesis presented an approach for discovery and extraction of software components in dockerfiles. It provides the data sets to build topology overviews for software components. It enables other systems to verify deployment structures at design time.

The approach uses public available dockerfiles as source to achieve the discovery. The concept of the approach contains a crawler to gather the dockerfiles. The crawler is intended as a service-specific crawler, because there are online services available, which provide a huge amount of dockerfiles. To discover and extract the software components according to the relations constructed in the dockerfiles the concept contains command-specific analyzers which operate based on the syntax of installation commands. A metamodel forms the base for a understandable preparation of the results. An algorithm brings all three parts together and ensures a smooth operation.

For validation of the concept an implementation was created. It includes a crawler for the online software repository service GitHub and command-specific analyzers for several wide-spread Linux installation command-line tools. In a test execution 80% of all installation commands were covered.

In comparison with other application topology discovery approaches which operate on running systems this approach operates on build-files independent of a running system. This eliminates any side effects of the discovery to a running system and makes the results independent of the discovery execution time.

The implementation can be extended in future work to support more online services as crawler-target. Furthermore, more installation commands of Linux as well as Windows-based commands can be supported. The implementation provides appropriate extension possibilities.

Components in URLs of installation commands are not yet extracted. This could be helpful to easier group the discovered components by name. Currently the same components can have different names, because if an URL is used in the installation command the complete URL is used as name.

In case of multi-stage builds in dockerfiles the relations between docker images could be taken into account as future work. They are given as COPY-instruction in dockerfiles.

Bibliography

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Automated Discovery and Maintenance of Enterprise Topology Graphs". In: *2013 IEEE 6th International Conference on Service-Oriented Computing and Applications* (Dec. 2013), pp. 126–134. DOI: 10.1109/SOCA.2013.29 (cit. on pp. 25, 28).

[Béz05] J. Bézivin. "On the unification power of models". In: *Software & Systems Modeling*. Vol. 4. 2. Berlin: Springer, 2005, pp. 171–188. DOI: 10.1007/s10270-005-0079-0 (cit. on pp. 21, 38).

[BFL+12] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm. "Formalizing the Cloud through Enterprise Topology Graphs". In: *2012 IEEE Fifth International Conference on Cloud Computing* (2012), pp. 742–749. DOI: 10.1109/CLOUD.2012.143 (cit. on p. 24).

[Bin15] T. Binz. *Crawling von Enterprise Topologien zur automatisierten Migration von Anwendungen – eine Cloud-Perspektive*. Dissertation. Stuttgart: Institut für Architektur von Anwendungssystemen der Universität Stuttgart, 2015 (cit. on p. 28).

[BKK01] A. Brown, H. Kar, A. Keller. "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment". In: *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No. 01EX470)*. 2001, pp. 377–390. DOI: 10.1109/INM.2001.918054 (cit. on p. 25).

[BLNS12] T. Binz, F. Leymann, A. Nowak, D. Schumm. "Improving the Manageability of Enterprise Topologies Through Segmentation, Graph Transformation, and Analysis Strategies". In: *2012 IEEE 16th International Enterprise Distributed Object Computing Conference* (Sept. 2012), pp. 61–70. DOI: 10.1109/EDOC.2012.17 (cit. on p. 25).

[BNS17] A. Brogi, D. Neri, J. Soldani. "DockerFinder: Multi-attribute Search of Docker Images". In: *2017 IEEE International Conference on Cloud Engineering (IC2E)* (Apr. 2017), pp. 273–278. DOI: 10.1109/IC2E.2017.41 (cit. on pp. 34, 35).

[BNS18] A. Brogi, D. Neri, J. Soldani. "A microservice-based architecture for (customisable) analyses of Docker images". In: *Software: Practice and Experience* 48.8 (Apr. 2018), pp. 1461–1474. DOI: 10.1002/spe.2583 (cit. on pp. 33, 34).

[Cis14] Cisco. *NetFlow Overview*. 2014. URL: https://www.cisco.com/c/en/us/td/docs/ios/12_2/switch/configuration/guide/fswtch_c/xcfnfov.html (visited on 01/17/2019) (cit. on p. 29).

[CW01] G. Cockton, A. Woolrych. *Understanding Inspection Methods: Lessons from an Assessment of Heuristic Evaluation*. Ed. by A. Blandford, J. Vanderdonckt, P. Gray. London: Springer London, 2001, pp. 171–191. ISBN: 978-1-4471-0353-0. DOI: 10.1007/978-1-4471-0353-0_11 (cit. on p. 51).

[CZMB08] X. Chen, M. Zhang, M. Mao, P. Bahl. "Automating network application dependency discovery: experiences, limitations, and new solutions". In: *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (Dec. 2008), pp. 117–130 (cit. on pp. 25, 28–32, 51).

[CZZ12] R. D. Cosmo, S. Zacchiroli, G. Zavattaro. "Towards a Formal Component Model for the Cloud". In: *International Conference on Software Engineering and Formal Methods* (2012), pp. 156–171. DOI: 10.1007/978-3-642-33826-7_11 (cit. on pp. 23, 24).

[DCF13] C. Ding, Y. Chen, X. Fu. "Crowd Crawling: Towards Collaborative Data Collection for Large-scale Online Social Networks". In: *Proceedings of the first ACM conference on Online social networks* (Oct. 2013), pp. 183–188. DOI: 10.1145/2512938.2512958 (cit. on pp. 36, 37).

[Deu19] ubuntu Deutschland e.V. *apt-get Wiki*. 2019. URL: https://wiki.ubuntuusers.de/apt/apt-get/#apt-get-install (visited on 04/17/2019) (cit. on p. 50).

[Doc19a] Docker. *Docker*. 2019. URL: https://docker.com (visited on 02/08/2019) (cit. on pp. 19, 52, 64).

[Doc19b] Docker. *Dockerfile reference*. 2019. URL: https://docs.docker.com/engine/reference/builder/ (visited on 02/08/2019) (cit. on pp. 19, 48, 63, 65).

[Ecl19] Eclipse. *Eclipse winery*. 2019. URL: https://github.com/eclipse/winery (visited on 03/22/2019) (cit. on p. 71).

[Ens99] C. Ensel. "Automated generation of dependency models for service management". In: *Workshop of the OpenView University Association (OVUA'99*. 1999 (cit. on p. 25).

[FC13] G. Farah, D. Correal. "Analysis of Intercrossed Open-Source Software Repositories Data in GitHub". In: *2013 8th Computing Colombian Conference (8CCC)* (Aug. 2013), pp. 1–6. DOI: 10.1109/ColombianCC.2013.6637537 (cit. on pp. 33, 34, 36).

[Git19] GitHub. *GitHub*. 2019. URL: https://github.com (visited on 02/04/2019).

[HN99] A. Heydon, M. Najork. "Mercator: A scalable, extensible Web crawler". In: *World Wide Web* 2.4 (July 1999), pp. 219–229. DOI: 10.1023/A:1019213109274 (cit. on p. 33).

[JPRD10] N. Joukov, B. Pfitzmann, H. Ramasamy, M. Devarakonda. "Application-Storage Discovery". In: *Proceedings of the 3rd Annual Haifa Experimental Systems Conference* (May 2010), p. 19. DOI: 10.1145/1815695.1815720 (cit. on pp. 25, 31).

[KGE06] A. Kind, D. Gantenbein, H. Etoh. "Relationship Discovery with NetFlow to Enable Business-Driven IT Management". In: *2006 IEEE/IFIP Business Driven IT Management* (May 2006), pp. 63–70. DOI: 10.1109/BDIM.2006.1649212 (cit. on pp. 25, 28–30).

[Kla99] M. Klar. *A semantical framework for the integration of object oriented modeling languages*. Dissertation. Technische Universität Berlin, Apr. 1999 (cit. on p. 22).

[KNS92] G. Keller, M. Nüttgens, A.-W. Scheer. “Semantische Prozessmodellierung auf der Grundlage ëreignisgesteuerter Prozessketten (EPK)””. In: *Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWi), Universität des Saarlandes* (1992) (cit. on p. 22).

[LLC98] H.-C. Lin, S.-C. Lai, P.-W. Chen. “An Algorithm for Automatic Topology Discovery of IP Networks”. In: *ICC’98. 1998 IEEE International Conference on Communications. Conference Record. Affiliated with SUPERCOMM’98 (Cat. No. 98CH36220)*. June 1998, pp. 1192–1196. DOI: 10.1109/ICC.1998.685197 (cit. on p. 25).

[LYL99] H.-C. Lin, C. Ye, C. Lin. “Automatic topology discovery and virtual connection trace for ATM networks using SNMP”. In: *Integrated Network Management VI. Distributed Management for the Networked Millennium. Proceedings of the Sixth IFIP/IEEE International Symposium on Integrated Network Management.(Cat. No. 99EX302)*. May 1999, pp. 939–940. DOI: 10.1109/INM.1999.770735 (cit. on p. 25).

[MDJV08] K. Magoutis, M. Devarakonda, N. Joukov, N. Vogl. “Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems”. In: *IBM Journal of Research and Development*. Vol. 4. 5. July 2008, pp. 367–377. DOI: 10.1147/rd.524.0367 (cit. on pp. 30–32).

[MDM07] K. Magoutis, M. Devarakonda, K. Muniswamy-Reddy. “Galapagos: Automatically Discovering Application-Data Relationships in Networked Systems”. In: *2007 10th IFIP/IEEE International Symposium on Integrated Network Management* (May 2007), pp. 701–704. DOI: 10.1109/INM.2007.374831 (cit. on pp. 25, 30).

[MDW+00] V. Machiraju, M. Dekhi, K. Wurster, J. Holland, M. Griss, P. Garg. “Towards generic application auto-discovery”. In: *NOMS 2000. 2000 IEEE/IFIP Network Operations and Management Symposium’The Networked Planet: Management Beyond 2000’(Cat. No. 00CB37074)*. Apr. 2000, pp. 75–87. DOI: 10.1109/NOMS.2000.830376 (cit. on pp. 22, 25–27, 50, 51).

[MLJC17] Y. Ma, H. Li, X. Jiyao, Y. Chen. “Mining the Network of the Programmers: A Data-Driven Analysis of GitHub”. In: *Proceedings of the 12th Chinese Conference on Computer Supported Cooperative Work and Social Computing* (Sept. 2017), pp. 165–168. DOI: 10.1145/3127404.3127431 (cit. on pp. 33, 36).

[npm19] npmInc. *npm-install documentation*. 2019. URL: https://docs.npmjs.com/cli/install (visited on 04/17/2019) (cit. on p. 50).

[Obj19] ObjectManagementGroup. *Unified Modeling Language*. 2019. URL: http://www.uml.org/ (visited on 01/30/2019) (cit. on p. 21).

[OCC17] OCCI. *OCCIware*. 2017. URL: https://www.occiware.org (visited on 02/04/2019) (cit. on p. 39).

[Ope19] OpenTOSCA. *OpenTOSCA winery*. 2019. URL: https://github.com/OpenTOSCA/winery (visited on 03/22/2019) (cit. on p. 71).

[PCAM16] F. Paraiso, S. Challita, Y. Al-Dhuraibi, P. Merle. “Model-DrivenManagement of Docker Containers”. In: *2016 IEEE 9th International Conference on cloud Computing (CLOUD)* (June 2016), pp. 718–725. DOI: 10.1109/CLOUD.2016.0100 (cit. on pp. 37–40).

[Sch11] C. Schweda. *Development of Organization-Specific Enterprise Architecture Modeling Languages Using Building Blocks*. Dissertation. München: Lehrstuhl für Informatik XIX Technische Universität München, Apr. 19, 2011 (cit. on p. 21).

[Sch13] A.-W. Scheer. *ARIS — Modellierungsmethoden, Metamodelle, Anwendungen*. Berlin: Springer, 2013 (cit. on p. 22).

[SS13] S. Sagiroglu, D. Sinanc. "Big data: A review". In: *2013 International Conference on Collaboration Technologies and Systems (CTS)* (May 2013). DOI: 10.1109/CTS.2013.6567202 (cit. on p. 32).

[The19] TheOpenGroup. *TOGAF*. 2019. URL: https://www.opengroup.org/togaf (visited on 01/30/2019) (cit. on pp. 21, 22).

[Tur15] J. Turnbull. *IEEE Software: Docker*. Interview. Apr. 23, 2015 (cit. on p. 19).

[Zac99] J. Zachman. "A framework for information systems architecture". In: *IBM Systems Journal Vol. 38* (1999), pp. 454–470. DOI: 10.1147/sj.382.0454 (cit. on pp. 21, 22).

A Appendix

Table A.1 contains the result set of all seven test execution of the crawler implementation. *Status* depicts if the dockerfile is valid, incorrect or a false-positive. The column *R* gives the number of RUN instructions, *I* gives the number of install lines and *C* gives the number of covered install lines.

Table A.1: Test execution results

Repository and Directory of dockerfile	Status	R	I	C
legshampoo.videoManager	valid	1	1	1
thanhson1085.VET	valid	9	4	4
thanhson1085.VET.netapis	valid	6	3	3
Jc2k.hpfeeds3	valid	6	4	3
ataylorme.docker-node-ataylor.me	valid	29	12	12
ravikumari.dockerdemo	valid	0	0	0
roeehershko.ps.build	valid	7	4	3
roeehershko.ps	valid	7	4	3
roeehershko.ps.images.meteor	valid	4	3	2
TheCodingLand.OtApi-microservice	valid	2	1	0
TheCodingLand.OtApi-microservice.project	valid	0	0	0
TheCodingLand.OtApi-microservice.swagger	valid	0	0	0
joethompson1961.CarND-Capstone	valid	14	7	6
anaglyph.screenly-dynamic-content	valid	1	1	0
slateci.container-gan-app	valid	4	1	1
Ucandoit.template-angular2	valid	0	0	0
netcsc.docker-jenkins-slaves	valid	4	2	2
aaronprince05.IoTEdgeMessaging.MessageGeneratorModule.Docker.linux-x64	valid	0	0	0
aaronprince05.IoTEdgeMessaging.MessageGeneratorModule.Docker.linux-x64.Dockerfile	valid	3	1	1
aaronprince05.IoTEdgeMessaging.MessageGeneratorModule.Docker.windows-nano	valid	0	0	0
aaronprince05.IoTEdgeMessaging.MessageReceiverModule.Docker.linux-x64	valid	0	0	0
aaronprince05.IoTEdgeMessaging.MessageReceiverModule.Docker.linux-x64.Dockerfile	valid	3	1	1
aaronprince05.IoTEdgeMessaging.MessageReceiverModule.Docker.windows-nano	valid	0	0	0

Will be continued on the next page

Table A.1 – continued from the previous page

Repository and Directory of dockerfile	Status	R	I	C
slateci.container-globus-connect	valid	8	4	4
johnbiundo.node-db-docker	valid	5	3	3
aghassabian.dind-test	valid	4	1	1
gopenguin.gitlab-bot	valid	1	1	1
zenstain.SWGEmu-Core3.MMOCoreORB.docker	valid	21	4	3
freetonik.shelfish	valid	5	3	1
badboy95.newone1	valid	12	5	4
ultrafez.dockerfiles.kotlinc	valid	7	1	1
chengyil.ember101.borrowerapi	valid	3	1	1
nikolvs.dockerfiles.salt-minion.2017.7.2	valid	6	2	2
bardiir.web-optim.docker.dash-hls	valid	10	2	1
bardiir.web-optim.docker.tools	valid	3	1	1
bardiir.web-optim	valid	0	0	0
antomate.container-deepdive._sources	valid	2	0	0
antomate.container-deepdive._sources.dockerfile-stupidwebsitev2	valid	3	0	0
Antoniolm.serviceInFlask	valid	11	6	6
javadevmtl.elastic6-marathon	valid	4	1	0
badboy95.newone	valid	12	5	4
lamtharnhantrakul.GestureRNN-ML4Lightpad.dockerfile.cpu	valid	4	4	4
nate-johnston.spintest	valid	0	0	0
tomassundvall.poc-dotnetcore-rest	valid	1	0	0
pteich.docker-fluentd-gelf	valid	10	3	2
35hunter35.test2	valid	4	2	2
35hunter35.test2.scripts.genetic_algo	valid	43	7	5
drbozdog.Tagzy.TagzyBackend	valid	2	1	1
kube-HPC.monitor-server.dockerfile	valid	3	1	1
BarristerBro.jhipsterSampleApplication.src.main.docker	valid	0	0	0
coinext.silverstring-exchange.silverstring-web.docker	valid	1	0	0
tyler-cromwell.Examples.docker	valid	5	2	2
James-Dao.testcicd	valid	5	2	1
brancz.kube-rbac-proxy	valid	5	3	2
brancz.kube-rbac-proxy.examples.example-client	valid	1	1	1
brancz.kube-rbac-proxy.examples.grpcc	valid	3	3	2
peterewills.sagemaker_notebooks.breast_cancer_sklearn.container	valid	8	2	2
peterewills.sagemaker_notebooks.breast_cancer_sklearn.container-dupe	valid	8	2	2
tinybat02.Assignment4.Assignment.hello-world-service	valid	5	2	2
tinybat02.Assignment4.Assignment.product-descp-service	valid	5	2	2
tinybat02.Assignment4.Assignment.product-price-service	valid	5	2	2

Will be continued on the next page

Table A.1 – continued from the previous page

Repository and Directory of dockerfile	Status	R	I	C
tinybat02.Assignment4.Assignment.server	valid	5	2	2
DavidMarslandia.Tests	valid	7	2	2
Mongey.mesos-riemann	valid	1	1	0
divide2.sms-service.src.main.docker	valid	5	2	2
JoinCode17.shadowsocks-libev.docker.alpine	valid	11	3	2
JoinCode17.shadowsocks-libev.docker.ubuntu	valid	8	2	1
netCommonsEU.PeerStreamer-docker	valid	17	4	3
bourgeoa.nelson	valid	5	3	2
cidemo-integration-tests.auth-service-mock	valid	1	1	1
djordn.simple-crud.php-docker	valid	1	1	0
FridaW.mernio-blog-auth-acl	valid	2	1	1
FridaW.mernio-blog-auth-acl.Dockerfile-development	valid	2	1	1
fastZhe.docker.base6.8	valid	13	2	2
fastZhe.docker.base7	valid	13	2	2
fastZhe.docker.cdh.cdhbase	valid	8	1	1
fastZhe.docker.express_node.express	valid	1	1	1
fastZhe.docker.javatomcat	valid	2	1	1
fastZhe.docker.myubuntu	valid	2	1	1
fastZhe.docker.nginxconf	valid	0	0	0
fastZhe.docker.nodelixian	valid	2	0	0
fastZhe.docker.nodenginx	valid	6	3	2
fastZhe.docker.storm-hz-docker.base	valid	6	1	1
fastZhe.docker.storm-hz-docker.storm	valid	6	0	0
fastZhe.docker.storm-hz-docker.storm-nimbus	valid	2	0	0
fastZhe.docker.storm-hz-docker.storm-supervisor	valid	11	3	3
fastZhe.docker.storm-hz-docker.storm-ui	valid	1	0	0
fastZhe.docker.storm-hz-docker.zookeeper	valid	4	0	0
fastZhe.docker.storm-mono-docker.base	valid	6	1	1
fastZhe.docker.storm-mono-docker.storm	valid	7	0	0
fastZhe.docker.storm-mono-docker.storm-nimbus	valid	2	0	0
fastZhe.docker.storm-mono-docker.storm-supervisor	valid	11	3	3
fastZhe.docker.storm-mono-docker.storm-ui	valid	1	0	0
fastZhe.docker.storm-mono-docker.zookeeper	valid	4	0	0
fastZhe.docker.ubuntu_node_repo_install	valid	7	4	4
AppScriptIO.reverseProxyServer.script.container.containerBuild	valid	0	0	0
tokutoku3.superset-with-google-oauth	valid	4	2	2
cian-w.bookings-microservice	valid	1	1	1
arthur-nesterenko.react-skeleton	valid	5	1	0
cidemo-integration-tests.express-demo	valid	1	1	1
Hydrospheredata.hydro-serving-sidecar	valid	1	1	1

Will be continued on the next page

Table A.1 – continued from the previous page

Repository and Directory of dockerfile	Status	R	I	C
cisagov.lambda_functions.pshtt	valid	1	1	1
cisagov.lambda_functions.sslyze	valid	1	1	1
cisagov.lambda_functions.trustymail	valid	1	1	1
3tty0n.opam	valid	3	1	1
Strayer.docker-bitcoind-full-node	valid	9	0	0
roncrivera.docker-cicd	valid	1	1	0
WeAreOpenSourceProjects.NodeAngular.Node	valid	14	5	4
WeAreOpenSourceProjects.NodeAngular.Node.Dockerfile-production	valid	14	5	4
akzk.dockerhub.centos.hadoop.mac	valid	15	5	5
akzk.dockerhub.centos.py34	valid	18	4	4
akzk.dockerhub.centos.spark.mac	valid	10	1	1
akzk.dockerhub.ubuntu.14apt	valid	0	0	0
akzk.dockerhub.ubuntu.hadoop.mac	valid	13	2	2
akzk.dockerhub.ubuntu.hadoop.mac.Dockerfile.bak	valid	13	3	3
akzk.dockerhub.ubuntu.java8	valid	8	1	1
akzk.dockerhub.ubuntu.py34	valid	5	2	1
akzk.dockerhub.ubuntu.spark.mac	valid	10	1	0
akzk.dockerhub.ubuntu.tensorflow	valid	1	1	1
nlppln.ocrevaluation-docker	valid	7	1	1
netcsc.docker-jenkins-master	valid	15	8	8
Arnor.symfony-docker.symfony1..docker.php-fpm	valid	3	2	1
Arnor.symfony-docker.symfony2..docker.php-fpm	valid	3	2	1
njoy.Docker-images.njoy.base	valid	11	4	1
njoy.Docker-images.njoy.latest	valid	4	2	1
njoy.Docker-images.njoy.llvm	valid	11	7	3
njoy.Docker-images.njoy21.latest	valid	16	1	0
njoy.Docker-images.njoy2016.21	valid	10	1	0
njoy.Docker-images.njoy2016.23	valid	10	1	0
njoy.Docker-images.njoy2016.24	valid	10	1	0
fireworq.fireworqonsole.script.docker.fireworqonsole	valid	9	1	1
jonassteinberg1.docker.drkiq	valid	5	2	1
jonassteinberg1.docker.ecs_base_image	invalid	0	0	0
jonassteinberg1.docker.ecs_base_image.Dockerfile.test.1	invalid	0	0	0
jonassteinberg1.docker.tomcat8_Dockerfile	valid	0	0	0
fireworq.fireworq.script.docker.code	valid	2	0	0
fireworq.fireworq.script.docker.fireworq	valid	6	1	1
fireworq.fireworq.script.docker.mysql	valid	0	0	0
KillerBee05.Ice.client.src.assets.codemirror.mode.dockerfile.dockerfile.js	false-positive	0	0	0

Will be continued on the next page

Table A.1 – continued from the previous page

Repository and Directory of dockerfile	Status	R	I	C
Vaschenko-Volodymyr.goeuro-test	valid	4	1	1
dwojciec.go-example.cmd.server	valid	0	0	0
killianlevacher.Apollo	valid	4	2	1
autopilotpattern.minio-manta	valid	21	2	2
ryysud.screenshot2slack	valid	5	3	2
griffithlab.docker-pvactools	valid	15	2	2
GeppettoTeam.DhinaiconAndroid_10008.projects.db	valid	0	0	0
GeppettoTeam.DhinaiconAndroid_10008.projects.desktop	valid	1	1	1
TOSUKUi.bittrex-trailer	valid	1	1	0
wudixm.wudixm.github.io._posts.2018-08-02-Dockerfile.md	false-positive	0	0	0
pvrforpranavvr.spring_mvc_hibernate	valid	0	0	0
adamchenwei.appercut.service.code-generator	valid	3	2	2
rodolfopeixoto.ticketee	valid	5	4	2
whatsclose.whatsclose-band-service	valid	0	0	0
coyg7.node-todo-api	valid	1	1	1
c8112002.pouch-web	valid	2	2	1
ybekdemir.django-rest-oauth-example	valid	2	1	0
itaydressler.image-diff	valid	1	1	0
jbinglei.StoreSomeFile	valid	0	0	0
coutantal.devopsB3.A	valid	4	2	2
munen.dashboard.board	valid	0	0	0
munen.dashboard.router	valid	0	0	0
munen.dashboard.web	valid	4	2	1
ITSec-UR.praktomat-utils	valid	6	2	2
littleguyy.Proxy0Net	valid	7	2	2
RizkiMufrizal.Docker-Spring-Cloud.Admin-Dashboard	valid	1	0	0
RizkiMufrizal.Docker-Spring-Cloud.API-Gateway	valid	1	0	0
RizkiMufrizal.Docker-Spring-Cloud.Catalog-Service	valid	1	0	0
RizkiMufrizal.Docker-Spring-Cloud.OAuth2-Service	valid	1	0	0
RizkiMufrizal.Docker-Spring-Cloud.Transaction-Service	valid	1	0	0
blachlylab.mucor3	valid	12	9	5
jsosa.LFPtools.docker	valid	2	2	1
indigo-iam.esaco.esaco-app.docker	valid	2	0	0
GeppettoTeam.Freshandroid_10008.projects.db	valid	0	0	0
GeppettoTeam.Freshandroid_10008.projects.desktop	valid	1	1	1
Scalingo.ruby-addon-provider-boilerplate	valid	3	2	1
julscoob.tp-devops.A	valid	4	2	2
josephharding.imagesim.Dockerfile-cluster	valid	4	2	1
josephharding.imagesim.Dockerfile-vectorize	valid	6	2	1
benoitf.openwhisk-workspace	valid	11	4	4

Will be continued on the next page

Table A.1 – continued from the previous page

Repository and Directory of dockerfile	Status	R	I	C
danjust.cifarDocker	valid	5	2	1
himred.hcd	valid	13	4	3
cswcl.arch-docker-with-libkml	valid	6	0	0
javorka.artin-dna.docker	valid	33	2	0
pegaops.amazonlinux	valid	2	1	1
WingT.docker-lnmp.mysql	valid	12	1	1
WingT.docker-lnmp.nginx	valid	4	1	1
WingT.docker-lnmp.php	valid	6	1	1
fingerpich.grafana-farsi.docker.blocks.collectd	valid	4	3	3
fingerpich.grafana-farsi.docker.blocks.graphite	valid	14	2	1
fingerpich.grafana-farsi.docker.blocks.graphite1	valid	25	9	4
fingerpich.grafana-farsi.docker.blocks.mysql_opendata	valid	6	2	2
fingerpich.grafana-farsi.docker.blocks.openldap	valid	5	1	1
fingerpich.grafana-farsi.docker.blocks.prometheus	valid	0	0	0
fingerpich.grafana-farsi.docker.blocks.prometheus2	valid	0	0	0
fingerpich.grafana-farsi.docker.blocks.smtp	valid	3	1	1
fingerpich.grafana-farsi.docker.buildcontainer	valid	3	1	1
fingerpich.grafana-farsi.docker.debtest	valid	2	1	1
fingerpich.grafana-farsi.docs	valid	0	0	0
fingerpich.grafana-farsi.scripts.build	valid	20	8	8
ougar.jenkinsMultiBranch	valid	0	0	0
zhangliyuan7.using-docker.identidock	valid	2	1	1
zhangliyuan7.using-docker.identijenk	valid	10	3	3
zhangliyuan7.using-docker.identiproxy	valid	0	0	0
cyhon.yaml_merge_rs	valid	0	0	0
coord-e.cart	valid	14	3	2
jakubbanas.phpunit_workshop	valid	7	1	1
evanchodora.samba-container	valid	7	2	2
dockerremaker.LearnYii2.vendor.codeception.base	valid	13	4	1
lazerion.hz-tls-verification.server	valid	2	1	1
GontseNtshegi.YuQarGroupWebsite.src.main.docker	valid	0	0	0
apetrovYa.spring-petclinic-containerized	valid	6	2	2
colibridigital.learning-java-9	valid	0	0	0
deepfakes.faceswap.Dockerfile.cpu	valid	9	5	4
deepfakes.faceswap.Dockerfile.gpu	valid	13	5	4
GeppettoTeam.Tablettab_10008.projects.db	valid	0	0	0
GeppettoTeam.Tablettab_10008.projects.desktop	valid	1	1	1
gitszboy.jhipsterTest.src.main.docker	valid	0	0	0
JadenDream.NCDE04-SpecifyTheLogOutput	valid	2	0	0
josemesan.PruebaBaseDatos.src.main.docker	valid	0	0	0

Will be continued on the next page

Table A.1 – continued from the previous page

Repository and Directory of dockerfile	Status	R	I	C
likai1130.bank	valid	0	0	0
lyalka.coreTodo	valid	2	0	0
matgand.gcp-cd-codelab	valid	0	0	0
m-creations.docker-zookeeper	valid	2	1	0
observerio.observer-server	valid	13	8	8
observerio.observer-server.Dockerfile.build	valid	9	3	1
observerio.observer-server.Dockerfile.release	valid	4	1	1
observerio.observer-server.web	valid	3	2	2
observerio.observer-server.web.Dockerfile.build	valid	2	2	2
observerio.observer-server.web.Dockerfile.release	valid	0	0	0
odiazdom.jenkins-ansible.roles.setupJenkinsMaster.files.docker.jdk.1.8	valid	4	1	1
odiazdom.jenkins-ansible.roles.setupJenkinsMaster.files.docker.jenkins-master.2.90	valid	2	1	1
odiazdom.jenkins-ansible.roles.setupJenkinsMaster.files.docker.tomcat.8	valid	9	2	2
spiritabsolute.psr	valid	4	3	1
wangjin.ngrok	valid	2	2	2
woahbase.alpine-go.Dockerfile_armhf	valid	5	1	1
woahbase.alpine-go.Dockerfile_x86_64	valid	5	1	1
WorldVirus.db	valid	32	9	9

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature