TU WIEN Informatics

Updating Service-based Software Systems in Air-Gapped Environments

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Oleksandr Shabelnyk, BSc
Matrikelnummer 01263258

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Prof. Schahram Dustdar
Mitwirkung: Dr. Christos Tsigkanos
Dr. Pantelis Frangoudis

Wien, 18. Mai 2021

Oleksandr Shabelnyk   Schahram Dustdar

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

Updating Service-based Software Systems in Air-Gapped Environments

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Oleksandr Shabelnyk, BSc
Registration Number 01263258

to the Faculty of Informatics

at the TU Wien

Advisor: Prof. Schahram Dustdar
Assistance: Dr. Christos Tsigkanos
Dr. Pantelis Frangoudis

Vienna, 18th May, 2021

Oleksandr Shabelnyk   Schahram Dustdar

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Oleksandr Shabelnyk, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 18. Mai 2021

Oleksandr Shabelnyk

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

Disclaimer: The views expressed herein are those of the author(s) and do not necessarily reflect the views of the CTBTO Preparatory Commission.

I want to express my sincere gratitude to all those among my friends and family who have supported me throughout this breathtaking journey. Undoubtedly, the key success factor has been the outstanding support of my highly skilled supervisors, Dr Christos Tsigkanos, Dr Pantelis A. Frangoudis and Prof Schahram Dustdar. Besides getting professional and always fast feedback, I have been constantly supplied with new ideas and guided according to the best practices in the scientific community. All together it created so far the best experience in composing a scientific work for me.

Additionally, I want to commend my dearest colleagues Dr Aled Prys Rowlands (Remote Sensing Officer, CTBTO) and Julius Kozma (Operations Officer, CTBTO). Aled has been extensively supporting me by providing his expertise in different fields, helping with the elaboration of the evaluation scenario and many more, while Julius contributed tremendously to the requirements collection. Additionally, I want to thank Irina Hofstetter for her extraordinary support in preparing the memorandum to receive CTBTO's approval for using the Organization's affiliation, as well as publishing the thesis. Thereupon, I want to express my appreciation to all those who approved the related memorandum: Dr Lassina Zerbo (Executive Secretary), Julian Tangaere (OSI Coordinator) and Vadim Smirnov (Director of the OSI Division).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

Contemporary component-based systems often manifest themselves as service-based architectures, where a central activity is the management of their software updates. However, stringent security constraints in mission-critical settings often impose compulsory network isolation among systems, also known as an air gap; a prevalent choice in different sectors including private, public and governmental organizations where data security is a fundamental requirement. This raises several issues involving updates, stemming from the fact that controlling the update procedure of a distributed service-based system centrally and remotely is precluded by network isolation policies. A dedicated software architecture is thus required, where key themes are: dependability of the update process, interoperability with respect to the software supported and auditability regarding update actions previously performed. We adopt an architectural viewpoint and present a technical framework for updating service-based systems in air-gapped environments. We describe the particularities of the domain characterized by network isolation and provide suitable notations for service versions, whereupon satisfiability is leveraged for dependency resolution; those are situated within an overall architectural design. Finally, we evaluate the proposed framework over a real case study of an international organization, and assess the performance of dependency resolution procedures for practical problem sizes.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Abstract **ix**

Contents **xi**

1 Introduction **1**
- 1.1 Updating service-based air-gapped systems . . . 2
- 1.2 Methodological Approach . . . 3
- 1.3 Research Questions . . . 4
- 1.4 Structure . . . 5

2 Related Work **7**
- 2.1 Software Evolution Management . . . 7
- 2.2 Air-gapped Environments . . . 8
- 2.3 Beyond The State Of The Art . . . 9

3 Fundamentals **11**
- 3.1 Component-Based Development . . . 11
- 3.2 Software Dependencies . . . 13
- 3.3 Software Versioning . . . 17

4 Requirements and Design **19**
- 4.1 Requirements Engineering . . . 19
- 4.2 First interview . . . 19
- 4.3 Second interview . . . 22
- 4.4 User Stories . . . 24
- 4.5 Example software system . . . 25
- 4.6 Global vs local knowledge . . . 28
- 4.7 Predictability . . . 29
- 4.8 Maintaining global knowledge . . . 30
- 4.9 Dependency resolution and validation . . . 33
- 4.10 Numerical representation of versions . . . 33
- 4.11 Modelling . . . 34

5 Dependency Resolution with SMT **45**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.1 Problem Formulation with SMT 45
5.2 Dependency Resolution 48
5.3 PySMT Implementation 51

6 Software Architecture for Air-Gapped Updates **59**
6.1 Security aspects 60
6.2 Artefacts Storage 61
6.3 Docker in Production 63
6.4 Plugin support 68
6.5 Final architecture description 68
6.6 Graphical User Interface 71

7 Evaluation **73**
7.1 Applicability 73
7.2 Revisiting original requirements 81
7.3 Performance 82
7.4 Lessons Learned 87

8 Conclusion **93**
8.1 Discussion 93
8.2 Summary 94
8.3 Future Work 96

A Abbreviations **99**

B Screenshots **101**

C Listings **109**
C.1 Docker swarm configuration 109
C.2 Performance testing 110

List of Figures **119**

List of Tables **121**

List of Algorithms **123**

Bibliography **125**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

Contemporary software architectures reflect decades-long software engineering research and practice, where separation of concerns with respect to the wide-ranging functionality available throughout a software system is strongly emphasized. This leads to systems formed through the composition of loosely coupled independent software components, which are also often distributed. The trend towards breaking down software into increasingly smaller pieces introduces numerous advantages; however, it increases overall system complexity, including over its maintenance and managed evolution. This component-based view has culminated in service orientation, where service-oriented architectures (SOA) have seen wide applicability.

Software systems however are not static, but rather *evolve*, undergoing continual change, with software maintenance thus constituting a major activity [Leh80]. This is evident also in service-oriented component-based architectures, where software is designed, developed and maintained by different teams often applying Agile methodologies. As such, software updates are a central theme in such contemporary systems. Those challenges are exacerbated in mission-critical settings where stringent security constraints impose compulsory network isolation among distributed systems, also known as *air gap*. Even though network isolation does not counter all security concerns [Byr13, GKKE14, GZE19], such design is a prevalent choice in different sectors involving critical systems, be it in private, public or governmental organizations. However, challenges arise when there is a need to initially provision and later update distributed service-based software products – an update of a software service/component may introduce breaking changes to other dependent products. [1] Air-gapped isolation generally imposes challenges in the lifecycle management of service-based software systems, the lack of constant availability of resources being a major issue, and is in contrast with the spirit of modern DevOps practices [MYV18]. In working environments where an air gap is in place, the lack of internet connectivity also harms productivity [WW18].

[1] Throughout the thesis "component" and "service" are used interchangeably, unless otherwise specified.

In particular, air-gapped service-based systems typically reside in physical locations while being isolated from other networks. In such a scenario, the software is developed in some organizational setting, and software deployment involves physically transferring services/components to the air-gapped system. Accordingly, this impacts the development process and the procedure by which software is updated; including: (i) older versions of services may already be deployed in the air-gapped system, especially relevant since (ii) services exhibit dependencies between them; (iii) while the configuration of services already deployed in the air-gapped system may be unknown. Naturally, software updates, their modelling and dependency resolution are problems that have been treated by the community extensively and in several forms [MBC+06, ADCBZ09, ADCTZ12, ACGZ20].

However, updating air-gapped systems raises several issues from a software architecture perspective, especially given the overall mission-critical setting; those include: (i) the configuration of components produced to update the system should be verifiably correct, since there is significant cost-to-repair for incorrect updates, (ii) service-based architectures entail containerized services, with support of different runtime environments, and (iii) update actions should be recorded in a traceable manner, in order to support auditability and regulatory compliance.

We adopt an architectural viewpoint and address the challenges imposed by the particularities of updating air-gapped service-based systems. In this thesis, a technical framework for updating distributed software systems in air-gapped environments is presented. Our main contributions are as follows:

- We detail the domain characterized by network isolation and provide suitable modelling notations for component versions;
- We leverage satisfiability for dependency resolution, providing alternative strategies with correctness guarantees and address the trade-off between their execution time and resolution quality;
- We evaluate the proposed framework over a realistic case drawn from an international organization and assess the performance of the dependency resolution procedures for practical problem sizes.

1.1 Updating service-based air-gapped systems

An *air gap* is a security measure employed to ensure that a computer system is physically network-isolated from others, such as the internet or local area networks. The air gap design may manifest in computers having no network interfaces to others, while residing in a physically isolated location. This is because a network connection – often used to update software – represents a security vulnerability or regulatory violation. To transfer data (or programs) between the network-connected world and air-gapped systems, one would typically use a removable physical medium such as a hard drive, while access is regulated and controlled [TPGN18]. The key concept is that an air-gapped system can

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

generally be regarded as closed in terms of data and software and cannot be accessed from the outside world. However, this has implications regarding contemporary software systems, which may need to be upgraded as part of software maintenance activities.

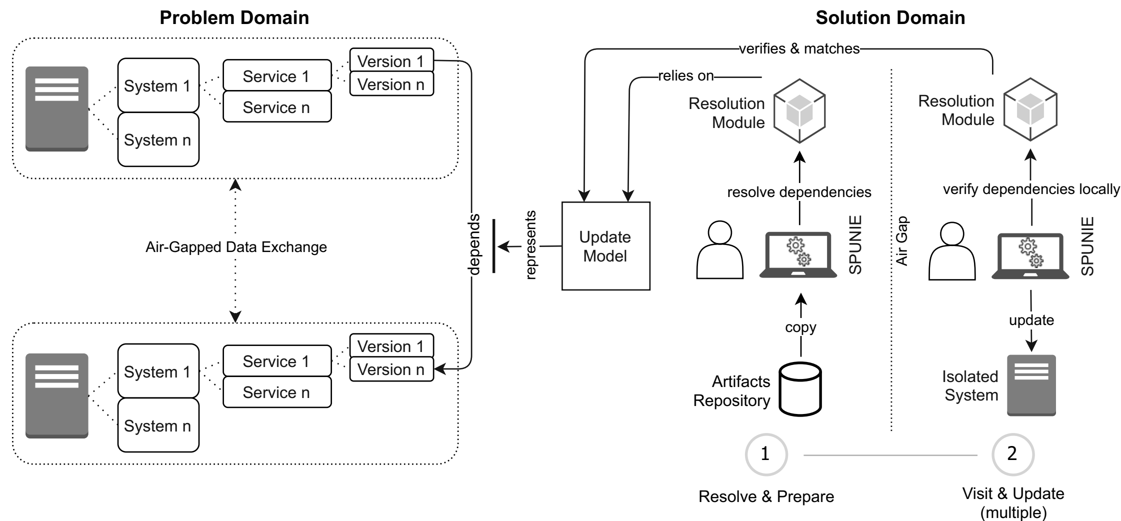

Figure 1.1: Updating service-based air-gapped systems – an overview

Figure 1.1 provides a schematic overview of the proposed approach. On the problem domain (left part of Figure 1.1) a series of air-gapped systems host software services, each having some version, in all comprising the service version configuration state of each air-gapped system, which is assumed to be known or adequately communicated. Software development takes place off-site – as such, services may need to be updated. Services – as software components – have dependencies, specified at development time. To perform an update on an air-gapped system (right part of Figure 1.1), the developer resolves dependencies of services per air-gapped system, building a valid service configuration taking into account its current configuration state – appropriate artefacts (such as containers) are pulled from development repositories and stored in a physical medium. Subsequently, the air-gapped system is visited, the service configuration is verified against the local state, and the update is performed.

1.2 Methodological Approach

The methodological approach consists of the following steps:

1. Case study

   To form a case study that tackles the outlined problem, requirements elicitation and final evaluation based on the on-site inspection (OSI) pillar of the Comprehensive Nuclear Test-Ban Treaty Organization (CTBTO) is used. We adopt the Volere

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

template for efficient and structured requirements engineering procedure [RR00]. One CTBTO staff member is appointed as the main contact person.

2. Modelling Dependency Resolution and Software Versioning

   The problem setting implies us tackling such topics as software versioning and dependency resolution. To solve the issue with verifying version consistency across an air gap, we develop a suitable model of a dependency resolution algorithm capable of satisfying our needs. As a consequence, software versions must be modelled too in order to make them compatible with the mentioned algorithm.

3. Research and literature review

   An extensive literature review is done primarily in the context of service-based and distributed systems. We are specifically focused on the following fields of studies: software evolution management, software versioning, dependency resolution, version consistency, deployment in air-gapped environments and software delivery with Docker containerization technology.

4. Proof-of-concept Development

   Plan, design, and construct a software solution as a proof of concept which solves the stated problem.

5. Evaluation

   We illustrate the applicability of the proposed solution by applying it to a typical scenario. To derive a realistic scenario and later evaluate the results, we engage in interviews with domain experts and stakeholders. We conduct a quantitative performance evaluation of the dependency solving procedure. The evaluated metrics are execution time to resolve dependencies and memory footprint. Additionally, we show those metrics using varying input data to cover the whole range of cases: from most simple to extreme ones.

1.3 Research Questions

This work aims to answer the following research questions:

- RQ1: How can provisioning and updating distributed service-based software systems be performed in air-gapped environments?
- RQ2: How can the domain be modelled to support dependency resolution?
- RQ3: Given the domain model, how can dependencies be resolved to support update in air-gapped systems?
- RQ4: How can software dependency resolution be performed without reliably knowing what components are installed and which are their respective versions?
- RQ5: How can the dependency resolution be implemented efficiently?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.4 Structure

This thesis is structured as follows:

- Related work is discussed in Chapter 2.
- Fundamental concepts, terms and technologies which serve as the basis are covered in Chapter 3.
- In Chapter 4 we introduce requirements engineering aspects, user stories, dependency resolution and versioning considerations, modelling and rationale for technologies selection.
- Chapter 5 is dedicated to one of the main contributions of this work – satisfiability modulo theories (SMT) based dependency verification and resolution module, covering modelling and implementation.
- In Chapter 6 we discuss the proposed software solution's architecture. More specifically, we tackle different architectural aspects such as security, artefacts storage, containerization technology in production, plugins, etc.
- Chapter 7 discusses evaluation, consisting of qualitative and quantitative aspects, also outlining lessons learned.
- Finally, we conclude in Chapter 8, discussing possible directions for future work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

Related Work

The architectural framework proposed is founded on the general area of updating software systems. Accordingly, in this chapter we classify related work into software evolution management (Section 2.1) and related approaches dealing with air-gapped environments (Section 2.2). In these sections we only present the related work without comparing it to ours. In the final Section 2.3 we elaborate on how our work differs from the current state of the art.

2.1 Software Evolution Management

The process of updating component-based distributed software systems has been extensively studied primarily in the context of the so-called dynamic updates [BGMLM16, PLM12, ALS06]. [1] This is in line with the booming popularity of cloud-based deployment, where the the focus is on reducing downtime while an update is performed. There are different techniques and approaches to achieve this.

As an example, the authors of [PLM12] present a model-driven approach to support software evolution of component-based distributed systems. It requires building a model, interface automata, to identify the specific class of update automatically. The class is derived based on information locally available in the component and indicates in which state and under which environmental conditions the system can be correctly updated.

Baresi et al. [BGMLM16] proposes version consistency as a criterion for safe and efficient dynamic updates. Their approach is mainly focused on updates during a distributed transaction. Since such a transaction is distributed, version consistency cannot be checked on all components participating in the transaction, as they are not co-located. Thus there is a need to identify a condition that is verifiable locally i.e., on the component(s)

[1] We denote component-based distributed software systems as those systems which, for example, adopt either service-oriented or microservice architecture. Please refer to Chapter 3 for more details.

TU WIEN Bibliothek Your knowledge hub

to update, and yet ensures global version consistency. The authors present a solution to the problem – a distributed algorithm for checking version consistency. It formalizes the proposed approach through a graph transformation system and verifies its correctness through model checking.

An alternative solution to distributed software updates over a network, which is not focused on reducing or avoiding downtime, is to take advantage of the mobile agent paradigm [BDNL02, Lan98]. The software packages are updated on a central server, then the mobile agents installed on the client receive the update. It is arguably a classical approach to deliver updates in a company to the employees' computers. We think however that it is only partially related to this work. Firstly, as pointed out, this approach usually targets client computers of end-users and not servers. Secondly, it is not meant to be applied in air-gapped networks, as it relies on at least occasional network connectivity for transferring updates.

Software updating usually relies on dependency resolution (also known as dependency solving) to identify suitable components and compatible versions. Dependency resolution has been approached by using various types of solvers such as boolean satisfiability (SAT), Mixed Integer Linear Programming (MILP), Answer Set Programming (ASP), Quantified Boolean Formulae (QBF), etc. [ADCTZ12, ADCBZ09, LBP08, LB10]. There is high competition among the solvers in terms of performance, as the dependency resolution problem is considered to be NP-Complete [Rus16, ADCTZ12].

Abate et al. [ADCTZ12] argue dependency solving should be treated as a separate concern from other component management concerns. They propose a modular software construct with the main goal to decouple the evolution of dependency solving from that of specific package managers and component models. One of their contributions is a Domain Specific Language (DSL) called CUDF. It positions itself as the "interface" between package managers and dependency solvers. For example, it can be used to encode components metadata and user updating request.

2.2 Air-gapped Environments

Gravity is an open-source application delivery system that lets engineers deliver and run cloud-native applications in regulated, restricted, or remote environments without added complexity [gra]. [2] It allows packaging complex Kubernetes clusters into portable images to later deliver them to a cloud-hosted environment independently of the provider. The main objective is to reduce the time and operational overhead related to Kubernetes, but also to avoid vendor lock-in, as multiple cloud providers can be used. Among others, they specifically target deployment of cloud-native applications in air-gapped environments [Mar19]. They allow to package a whole cluster, including the dependencies, to just one tarball file. It eliminates the need in utilizing an internet connection during the installation.

[2] More information on https://goteleport.com/gravity

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Azab et al. [AD16] refer to a very similar problem this thesis aims to face. They deal with TSD (Tjenester for Sensitive Data), an isolated infrastructure for storing and processing sensitive research data, e.g. human patient genomics data. In their work, they try to establish a procedure to provision software inside an isolated environment with the help of Docker containers and STROLL file system. A use case of provisioning Galaxy Bioinformatics Portal is presented. Due to its high complexity, this software should be a suitable example of applicability of the proposed solution approach. The authors also identify some security-related disadvantages of Docker containers and try to mitigate them. For example, Docker does not control what users can mount, which host directories to the containers. It poses a serious threat of data breach. They overcome this issue by controlling the mounting with a custom script. Overall, the experiment showed that the software could be successfully provisioned using Docker, although not without creative workarounds. The authors state Docker is not yet mature in terms of security, however, the article dates to 2016 and this may have been changed.

Air-gap isolation can impose a different type of challenges which is indirectly, but related to this thesis. In the working environments where an air gap is in place, the lack of internet connectivity can have a negative impact on the workers' productivity. Wong et al. [WW18] conducted a survey at a commercial software company that uses network isolation as a security measure. Their main goal was to identify how such working environment impacts the software development productivity. The results showed it is indeed an issue that is often neglected and left untreated. In particular, multiple affected areas were identified. Usually, in build dependency management, it is common to rely on the constant availability of resources. For instance, Maven central repository cannot be used without an internet connection as it is intended to. Approving certain packages may be a solution, but a potentially high number of transitive dependencies again makes it very difficult. Moreover, software engineers lose direct access to documentation; multiple tools become unusable, such as a plugin store in an integrated development environment. Also, collaboration tools, like those for bugs tracking, must be often duplicated to be available both inside the secured network and outside of it.

2.3 Beyond The State Of The Art

Discussions related to software updates are clearly primarily directed towards reducing the downtime of an application while an update is performed. The term **dynamic updates** was coined therefore. As the research shows, there are numerous attempts to reduce the downtime. Performing an update in the middle of a distributed transaction ranks among the biggest challenges in dynamic updates. In contrast, our problem setting is not about the reduction of downtime at all. Nevertheless, Baresi et al. [BGMLM16] inspired us to adopt their terminology – local vs global verification. The authors refer to it in the context of distributed transactions. In contrast to that, we introduce the notion of local reliable knowledge versus global unreliable one to come up with a suitable dependency resolution approach (see Section 4.6).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

At first glance, the introduced Gravity application delivery system seems to be applicable to the problem this thesis addresses. Nevertheless, it runs only on top of the Kubernetes platform and most importantly, it does not provide version consistency checking between dependant air-gapped applications. By contrast, our work targets different environments with Docker being the main one. On top of this, we do provide a mechanism to verify version consistency across an air gap. Our flexible plugin-based architecture allows integration of Kubernetes as an orchestration tool instead of the default Docker Swarm one. However, our solution does not intend to package and transfer complex Kubernetes clusters, as the Gravity application does.

We consider [AD16] it to be a valuable finding. The authors showed that Docker could be indeed successfully applied to provision software in network-isolated environments. This encourages us to follow this path, especially because CTBTO already uses Docker to run their applications. Our work does target the Docker platform as well, however it is not limited exclusively to it. Moreover, our problem setting is different, because we deal with provisioning and updating of spatially distributed server-hosted applications, whereas the authors of [AD16] are focused more on traditional desktop software. The authors' remark regarding Docker's security issues and other research related to Docker secure deployment [Yas18, XJR+18] makes us aware of what security aspects should be taken into account.

The contribution of Wong et al. [WW18] is only distantly related to our work, because it is human-centred as opposed to our purely technical treatment of the problem. However, our proposed software solution should indeed ease the working conditions of software engineers in air-gapped environments.

This work most importantly differs from the related research efforts described, as follows:

1. The setting that we treat does not require reducing downtime during an update. Therefore dynamic updates are not of interest in our case.

2. We seek dependencies resolution for isolated distributed components. The major challenge is that there could be no reliable knowledge of what versions and components are installed, unless the isolated servers are physically visited one by one.

3. We seek a well-defined procedure to safely and reliably deliver updates across one or multiple air gaps targeting different execution platforms with Docker being the most important one.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Fundamentals

3.1 Component-Based Development

Component-Based Development (CBD) facilitates reusing well-designed and tested software components to avoid code duplication and speed up the development process[Bro00, CLC05]. Many different attempts have been made towards CBD, varying from a simple encapsulation of source code into modules to advanced microservice-based architectures. In this Section we briefly cover different relevant component-based software development approaches.

3.1.1 Software services

If we look at the CBD developments within the last two decades, there is a clear trend towards modularization and breaking code into smaller and smaller entities. The so-called Service-Oriented Architecture (SOA) belongs to one of the milestones on this path. SOA was especially highly discussed in the first decade of the 21st century. According to Arsanjani, this concept is based on an architectural style that defines an interaction model between three primary parties: the service provider, who publishes a service description and provides the implementation for the service, a service consumer, who can either use the uniform resource identifier for the service description directly or can find the service description in a service registry and bind and invoke the service [Ars04]. Sprott et al. emphasize the similarity between SOA and object-oriented programming (OOP). Components in SOA, like objects or classes in OOP, do the following: they combine information and behaviour, hide the internal workings from outside intrusion and present a relatively simple interface to the rest of the organism [SW04]. Moreover, a service in SOA is usually published and discovered by the outside world in order to be consumed. This is again similar, for example, to finding and importing an external library in an object-oriented programming language in order to use it. Web services in SOA can be described formally using Web Services Description Language (WSDL), Simple Object

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Access Protocol (SOAP) for communication or other web-related standards. In this work, however, we do not focus on the technical side of SOA, but rather on its global effect on software engineering, operations, availability, etc. We do back up the opinion that SOA is not just an architecture of services seen from a technology perspective, but the policies, practices, and frameworks by which we ensure the right services are provided and consumed [SW04].

In the second decade of the 21st century, a new architectural style emerged, making it the next highly discussed topic in the community – microservice-based architecture. The main difference to SOA is a much finer granularity. A microservice is usually fairly small and lightweight, making it easy to be managed by a small team of engineers. Moreover, if needed, its small size allows it to be easily re-implemented, as well as scaled up. The communication protocol for services to talk to each other is also usually much more lightweight and flexible. A "trendy" choice is Representational State Transfer (REST) protocol which is leveraged by HTTP. Fowler and Lewis [FL14] significantly contributed to the sometimes difficult definition of microservice-based architecture:

> "In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies".

While as multiple sources imply (see [NSS14, FL14]) one of the biggest advantages is that a microservice is fully independent and can be deployed or updated separately. Albeit, it imposes the need to adopt new deployment techniques and strategies. In the next Section we shed light on containerization technology which serves as the main driver for deployment and operations of software services.

3.1.2 Container-based virtualization

In traditional virtualization of operating systems, a virtual machine (VM) hosts a fully independent system using the so-called hypervisor. It is often done to optimize the utilization of a computing unit's resources. For example, in a data centre, the resources of a server can be dynamically allocated to multiple customers depending on their needs. Before virtualization it meant that the whole server would be allocated to a customer, which is not always optimal. Compared to virtual machines, containers offer similar functionality. They allow hosting a guest operating system on top of another system. The biggest difference is that they are much more lightweight and demonstrate better performance. The reason is that containers share the OS kernel instead of fully emulating it, as it is done in a VM.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Currently, the most prominent technologies on the market enabling container-based virtualization are Linux Containers (LXC) and Docker. To be more specific, Docker leverages and runs on top of LXC technology. It adds a very convenient additional level of abstraction which greatly simplifies container management. Container-based and hypervisor-based virtualization do not directly compete with each other but rather pursue different goals. Scheepers [Sch14] makes a macro-benchmark performance comparison between two implementations of these technologies – Xen (full virtualization) and LXC (container-based). The outcome is that both technologies cover different use cases. If it is important that resources are distributed equally, and performance should be less dependent on other tasks, executed on the same system, Xen is a good option. However, if you want to optimise hardware or if you wish to execute a lot of small isolated processes, LXC is a better option, since it introduces less overhead and has higher performance [Sch14]. Thus, the portability of a container makes it very suitable for running just one single application. Combined with a convenient workflow introduced by Docker to package containers with the needed dependencies, it has become a popular choice for microservice-based architecture.

3.2 Software Dependencies

Re-usability of software components has always been part of the best software engineering practices. The idea of creating components that are self-contained, portable, hence also reusable, is as old as software development itself. In the software engineering community, a package or a piece of external software, which another software is dependant on, is called a dependency. Another term for dependency which is commonly used is inter-dependency [DPS+19]. Reusing components implies numerous advantages, as we just briefly mentioned, but there is also one major disadvantage – dependability. The reused component may introduce a breaking change in a newer version, meaning it is not backwards-compatible[RvDV17]. It makes reusing components challenging and needs special attention to the versioning approach.

We differentiate between different types of dependencies: application-level and component-level. Application-level dependencies are those that are required by software to be successfully built and functioning at runtime. A good example is the use of open-source frameworks or libraries. Component-level dependencies are needed during runtime to consume/provide data or computational power. For instance, in SOA each service is independently built and run, however, to fully leverage their functionality communication among them is needed. This kind of dependency is at the component-level.

3.2.1 Application-level dependencies

It is ubiquitous for software engineers to make use of Free and Open Source Software (FOSS) in their projects. This means adopting frameworks, libraries, etc. which are openly accessible and free of charge. We denote this kind of dependencies as application-level ones. German et al. [GGBR07] classifies application-level dependencies as follows:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Type** can be explicit or abstract. Explicit dependencies are those that the software engineers directly specify. For example, in an Apache Maven project, this would be adding JUnit with a version to the pom.xml file. Abstract dependencies are those which can usually be easily exchanged. For example, usually, ORM frameworks such as Hibernate can work with multiple different database implementations.
- **Importance** dictates if dependencies are required or optional. Optional ones would enable some additional feature(s). Required dependencies is a must, since the software cannot be built otherwise.
- **Stage** refers to the time when the dependencies are needed. It can be build-time, installation or run-time. Build-time dependencies are needed, for example, by the compiler to build the software. Installation ones refer to additional tools used when installing the application, for instance, to modify some settings. Lastly, run-time ones are required to execute a software or part of the software.
- **Usage methods** describe the method how a dependency is used. These are stand-alone programs (database management systems), middleware-based (httpd or web application server), plug-ins in a plug-in architecture, and linkable libraries (dlls in C++), etc.

3.2.2 Component-level dependencies

In contrast to application-level dependencies, we define another class of dependencies/-components – component-level dependencies. In SOA such dependencies are frequently used. They can be built fully separately; however, they may or may not be dependant on each other at run-time. An example of an optional dependency would be the following. Imagine a software application consisting of two components which can communicate over a network and thus be deployed fully separately. Let us assume, one of the two components is the main one, being the heart of the application. Another one is responsible only for calculating statistical data. The main component views the other one as purely optional, since it only provides data to it, but does not depend on it. Vice versa, the statistics component does depend on the main one. We want to point out that it can be started up isolated without any problems, but it will be useless, as it will not be able to process any data.

Pawan et al. studied inter-dependencies in SOA-based systems by using formal modelling through coloured Petri nets [RK15]. Alongside they introduced four types of service dependency: control dependency, data dependency, sequence dependency and composition dependency.

- Control Dependency

  "A control dependency between two services Si and Sj specifies the conditions under which service Sj is allowed to enter a state stj based on the state sti of service Si". The following example describes control dependency. Supposing, a customer

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

makes an order through the order service, the order service passes the control to the payment service so that the customer can pay for the good. Once the payment is successful, the control is back at the order service.

- Data Dependency

  "Execution of S1 needs some data which can provide service S2? In this case, service S1 is data-dependent on service S2. Service S1 calls service S2 and service S2 provides information needed by service S1". For example, if one service requires to get some data in order to proceed with the execution, this service is data-dependent on another service.

- Sequence Dependency

  "Sequence dependency among services means the execution of services takes place one by one, i.e. output of the first service is the input of the second service". Let us assume a customer can make an order only after he/she has logged in. This would mean the login service needs to confirm successful login first, so that the order service can proceed.

- Composition Dependency

  "When a service S is constructed using the services S1, S2, S3 ... Sm, service S is composed of services S1, S2, S3...Sm". Multiple services can be aggregated in a way that they operate as one service.

The authors of [KC00] differentiate between two distinct kinds of dependencies: requirements for loading an inert component into the runtime system, fitting our definition of application-level dependencies type, and dynamic dependencies between loaded components in a running system. The dynamic dependencies embrace the inter-connectivity of components, thus, fit our definition of component-level dependencies. Kon et al. introduce hooks, components and clients. Each component C has a set of hooks other components can be connected to. The hooked components are dependencies of C. There are also clients which depend on C. Additionally, there is a basic set of events such as DELETED, FAILED, RECONFIGURED, REPLACED, and MIGRATED. For example, the JVM implementation is updated, so a REPLACED event is propagated to all clients which are dependant on this component. When replacing a component, it might be feasible to transfer the current state. Also the communication to the old component must be stopped, redirected, etc.

According to [GNAK03], dependency relation between two components can be expressed with a strength (p). If component A depends on B with strength p, then p is the probability a given activity period of A contains an activity period of B. The authors state the above type is a true dependency. They argue the number p comes very close to the probability that a given execution of A invokes, directly or indirectly and at least once, the component B, and finishes only after the invocation to B returns. So if A calls B on each request, then per their definition, p is 1.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.2.3 Discovery

Discovering dependencies can be an uneasy task in some cases. For example, by inspecting a JavaScript application that uses npm package manager, the explicit dependencies are clearly visible and can be automatically retrieved. However, some other types of dependencies, like build-time ones, are usually enumerated in descriptive files such as in README, or INSTALL ones [GGBR07]. There are also numerous attempts to discover dependencies automatically by analyzing the source code [OBL10, GG01].

An alternative way to approach dependency discovery can be in using existing performance monitoring infrastructure available in middle-ware, such as web application or database servers [GNAK03]. A special data-mining algorithm using this kind of data can obtain "probabilistic" dependencies between components.

3.2.4 Resolution

Dependency resolution, also known as dependency resolving, is a process of finding the right combination of software components while preserving certain constraints such as version compatibility. The underlying dependency resolution problem is NP-Complete [Rus16, ADCTZ12] in most of the cases which makes it challenging to tackle. In Component-Based Software Systems dependency resolution is especially important. In case of Free and Open Source Software (FOSS) distribution, the software components are commonly called packages and are handled by package managers. Despite the fact that package managers differ in how they handle dependency resolution, some common traits can be extracted. According to Abate et al. [ADCBZ09, ADCTZ12] dependencies/packages usually have these properties defined:

1. Name and version

   The name and version must be set to make a package uniquely identifiable.

2. Dependencies

   Dependencies or positive requirements describe what components of what versions must be installed.

3. Conflicts

   In contrast to positive requirements, conflicts express negative ones – components/versions which must be absent.

4. Features

   Features are names of virtual components provided by a component.They may be used to satisfy the dependencies of other components.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3 Software Versioning

Software versioning is used to uniquely identify different states of a software while it evolves. Most commonly, software is versioned by adopting semantic versioning [PW19]. This approach uses three numbers separated by a dot, e. g. "2.4.1". The first number indicates **major** release and usually introduces breaking changes, making it incompatible with the previous major version release. The middle number reflects **minor** version change. Typically, it signalizes new functionality that has been added, however, full backwards compatibility is preserved. The last number stands for **patch** or **micro** changes which indicates bug fixes. Patch changes are also fully backwards compatible.

Software dependencies do not necessarily have to be strict. As discussed, minor and patch version numbers ideally do not introduce breaking changes. Thus, software engineers can choose to not depend on one specific version of the software, but rather, for example, only on major versions. This yields automatic adoption of any minor or patch release without explicitly stating it. Dietrich et. al. [DPS+19] devised a comprehensive list of classifications of version constraints, as shown in Table 3.1.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Classification	Description
fixed	A dependency on a fixed version (of another package), such as 1.2.3.
soft	Dependency on a fixed version, such as 1.2.3, but the package manager may choose another version in order to resolve dependency constraints, using some notion of closeness or similarity.
var-micro	Uses a wildcard for the micro part of the version string, such as 1.2.*, indicating that the project may depend on any version of a package with a version number starting with 1.2. This may include additional bounds, such as 1.2.* , < 1.2.4.
var-minor	Uses a wildcard for the minor part of the version string, such as 1.*, indicating that the project may depend on any version of a package with a version number starting with 1. This may include an additional bound.
any	Indicating that a project may use any version of the package it depends on, the package manager has unconstrained freedom to decide which one.
at-least	Indicating a dependency on any version following a specific version (inclusive or exclusive), such as [1.2.3,*].
at-most	Indicating a dependency on any version up to a specific version (inclusive or exclusive), such as [*,2.0.0].
range	A custom range, such as [1.2.0,2.0.0), indicating that a project depends on any version from 1.2.0 to 2.0.0. Either range bound can be inclusive or exclusive.
latest	The dependency should always be resolved to the latest version available, possibly with some qualifier (such as latest-stable, excluding beta versions.
not	A dependency is declared that explicitly excludes a certain version, usually this is caused by a known issue in this version. Some custom pattern, for instance, a complex boolean formula combining any of the resolved category.
unresolved	The dependency string contains unresolved variable references. This case occurs because libraries.io scans project metadata such as (Maven) POMs without understanding their semantics. A dependency might be declared in a POM by a reference to a variable declared elsewhere, such as $project.version.
unclassified	Default value of none of the mapping rules can be applied.

Table 3.1: Different classifications of version constraints as according to [DPS+19]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

Requirements and Design

In this chapter we discuss the requirements and design of the software solution presented in this work.

4.1 Requirements Engineering

To gather requirements in a structured and well-established way, we use the so-called Volere template [RR00]. We follow the guidelines provided by its creators on how to apply this template most efficiently [RR12]. The Volere template, as the name suggests, is a document providing a skeleton for a structured requirements engineering approach. It consists of five main building blocks: project drivers, project constraints, functional requirements, non-functional requirements and project issues. For our requirements elicitation, we decided to leave out the last one. It covers conditions under which the project will be done and, since we do not question the necessity of the project, it does not apply to us.

In the first session of the requirements engineering, an interview was conducted with key stakeholders to collect initial requirements and draft user stories. The following reflects the interview broken down by individual Sections according to the Volere template.

4.2 First interview

4.2.1 Project Drivers

The Purpose of the Product

One of the main purposes of the product is to improve the process of provisioning and updating software in air-gapped environments.

Client, Customer, Stakeholders

The main stakeholders are OSI surrogate inspectors including the Data Flow Management Officer (DFO), Technical Secretariat and member states of the CTBT.

Inspectors are all those who are part of an inspection team. An inspection team may be operational during an actual inspection or a training event. The DFO belongs to the inspection team. He or she is in charge of supporting seamless and secure data flow between the field, receiving and working areas at the Base of Operations (BoO), software operations and IT infrastructure. Also within the sphere of responsibility of the BoO is to support the chain of custody of electronic media including the transfer of data of sensitive data classified as "highly protected".

Users of the Product

One of the main users of the product is the DFO. After the CTBT's entry into force, Technical Secretariat shall also be responsible for software provisioning and updating.

4.2.2 Project Constrains

Mandated Constraints

One of the major restrictions is that internet connection is prohibited, and connections to other networks restricted. This makes software provisioning and updating challenging. Internet connection is generally prohibited on the premises during an inspection and while it may be possible to request replacement duplicate equipment form headquarters it remains unclear how software failures could be resolved once deployed. Only limited communication with the Headquarters in Vienna is allowed; for example, to receive a weather report, etc. During a training event, internet connection is permitted for support personnel; however, it is still highly unwanted to expose deployed servers to the internet.

Naming Conventions and Definitions

The used naming conventions and definitions are in line with the ones defined for the whole thesis and can be found in Appendix A.

Relevant Facts and Assumptions

Optimally, the software systems used during an inspection are maintained and up-to-date. For this purpose, the Organization uses dedicated software to make sure software and hardware maintenance are performed regularly. A final verification of software would take place during the "launch phase" of an inspection, after an inspection request is submitted by a member state. During this period, limited time would be available to resolve any issues identified during the verification of systems.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.2.3 Functional Requirements

The Scope of the Product

The product shall be seamlessly integrated into existing set of software installation. It is assumed to be robust and easy to use. The importance of the latter requirement should not be underestimated since, in contrast with similar international organisations, the CTBTO does not have permanent inspectors but relies on surrogate inspectors nominated by member states. While skilled and trained, the fact that surrogate inspectors do not have daily access to software systems means that any software developed on their behalf should be intuitive and easy to use.

The Scope of the Work

The domain of the product is any of those where software is operated and maintained in network-isolated environments. Usually, this restriction exists due to high-security requirements.

Functional and Data Requirements

It should be possible to define what versions of what components work together. One of the main use cases would be to update software, especially with breaking changes to other components. It should be done with as little manual involvement as possible.

4.2.4 Non-functional Requirements

Look and Feel

It is desirable by the stakeholders that the application has a similar look and feel to other applications in the OSI applications suite. However, since the solution should be of general use, its look and feel will not be tailored to CTBTO's applications. The graphical design should be clear, simple and neutral.

Usability

Since the installed product will have to be physically carried around and connected to various servers, it should be runnable on a portable device and have its own UI.

Performance

Performance is not critical; however, it should not be totally neglected. Downtime for performing updates of up to 5 minutes is acceptable. Robustness, version consistency, as well as security, are of greater concern.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Operational

Performing updates is expected to be done mostly indoors. The systems to be updated must not be exposed to the internet.

Maintainability and Portability

One of the concerns is to what extent technical knowledge is needed to support the application. Can it be done by members of the inspectors, specifically the DFO? Who will be responsible for defining the dependency constraints?

Security

A high level of security is expected. A desire was expressed to implement a two-factor authentication when performing an update or accessing the systems.

Cultural and Political

All the components i.e., libraries or frameworks used, shall be of one of the member states origin, as well as open-source.

Legal

It should be in line with the CTBT, the OSI operational manual and subsidiary documents such as standard operating procedures.

4.3 Second interview

The following text crystallises the requirements identified during a second interview with another stakeholder. Duplicate requirements are omitted.

4.3.1 Project Drivers

The Purpose of the Product

No changes.

Client, Customer, Stakeholders

Potentially a Security Officer shall be appointed to define an update strategy or internet connection policy. He or she is more of an approval authority than a hands-on user.

Users of the Product

An inspector may be the user of the product.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3.2 Project Constrains

Mandated Constraints

Currently, there is a VPN connection to the headquarters. Data can be sent there, if the ISP agrees. So it is not generally prohibited. Yet there is no complete agreement on what can be sent without approval.

Naming Conventions and Definitions

No changes.

Relevant Facts and Assumptions

After an inspection is conducted, only data and information in the "preliminary findings document" prepared by the inspection team can be returned to headquarters. Whether system/software logging information can be returned to headquarters remains the matter of discussion among member states.

It is important to prove the product does not collect potentially classified data while performing an update.

Again, it is unclear what would happen if a critical bug was detected during an inspection. In principle, if the inspected state party (ISP) agrees, an update could be sent from the headquarters to the site. At that moment, servers would not be connected to the internet or to the headquarters via VPN to receive a software update. It is also envisaged that each update would require approval. In the case of a critical bug, there is no agreed update procedure; however, connecting to headquarters via VPN would be the optimal solution.

4.3.3 Functional Requirements

The Scope of the Product

No changes.

The Scope of the Work

It is not the goal of the project to maintain other software packages (for example, proprietary desktop software packages for the processing of inspection data such as seismic or shallow geophysics data) or OS updates.

Functional and Data Requirements

It is desirable to collect logging information for auditing purposes and store it externally. The hardware used during an OSI is kept afterwards by the inspected state, so this information is lost otherwise.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.3.4 Non-functional Requirements

Look and Feel

No changes.

Usability

No changes.

Performance

No changes.

Operational

No changes.

Maintainability and Portability

No changes.

Security

No changes.

Cultural and Political

No changes.

Legal

All used software products during an OSI should be ideally open source.

4.4 User Stories

The structure of the user stories shall be: "As a [role], [where/when] I want [feature] so that [reason]". Additionally, we name each of the stories to make them better identifiable.

4.4.1 UPDATE-HQ

As PTS/OSI Data Flow Management Officer, at headquarters I need to update a software, because the contractors made an update and a new version is available.

4.4.2 UPDATE-TEST

As PTS/OSI Data Flow Management Officer, during a test event I need to update a software ASAP, because a fault has been found and corrected.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.4.3 UPDATE-OSI

As inspector, I need to update software during an inspection, because an urgent update has been released due to a serious fault.

4.4.4 DEFINE-DEPENDENCIES

As a software engineer, before releasing a new version I need to define dependencies, so that they are taken into account when deployment is being performed.

4.4.5 NO-DATA-COLLECTION

As a user, during an inspection I want to demonstrate to the ISP the software does not collect data so that he/she allows me to use it.

4.4.6 OVERVIEW

As a user, location independently I want to know which version of the software is installed at location X, so that I have an overview and can decide if an update is needed.

4.4.7 VERIFY

As an inspector or Data Flow Management Officer, during launch phase I need to verify everything is in place and update software if needed, so that inspection deployment can be safely done.

4.5 Example software system

Based on the collected requirements, some major design considerations are discussed in the following sections. We present an example software system that should apply to most of the user stories. It is created and discussed to facilitate design decisions by referring to it as needed. Figure 4.1 shows such an example system. All depicted software products make an ecosystem of different software installations which support the operations as a whole. The communication between the systems is of different types, depending on the restrictions in place. We distinguish these kinds of communication:

- Two-way air-gapped data exchange (external storage)

  Strict network isolation is required between the Receiving Area (RA) and Working Area (WA), thus data transfer can be done only through an air gap. It is done by securely copying arbitrary data to an external storage, moving it across the air gap, then importing it on the other end.

- Two-way on-demand data transfer (LAN cable)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

On-demand connection denotes a kind of connection that is established only occasionally. In some cases it must be approved beforehand. The data could be transferred in both directions. A good example is the connection between the RA and a field tablet. Each day before going out to the inspection area to gather inspection data, synchronization is needed to download the inspection team Daily Plan onto the tablets. Upon the team's return from the field, the collected data are reviewed and classified and are accordingly transferred to either the protected or the highly protected servers in the RA. It is clear that the connection is not always needed and that the data flows in both directions.

- One-way on-demand data transfer (LAN cable)

  A good example of a one-way data transfer would be how LabField Application communicates with RA. Similarly to the Field Application installed on field tablets, LabField Application downloads the Daily Plans in RA. However, there is no data transfer in the opposite direction.

- One-way QR-code-based data transfer

  Wireless connections of any kind shall be avoided during an OSI. Therefore, Field Application transfer data to LabField Application via a QR code. This way data can transferred wirelessly without using technologies such as Wi-Fi, Bluetooth, etc.

- Constant LAN connection

  As the name states, this type of connection is constant. For example, the core Laboratory Application (Lab App L) is constantly connected to a Java Application (Java App J1). The Java Application is an agent that runs on a Windows Server machine and listens to particular events emitted by another third-party software.

Further, let us cover each of the components in more detail. To provide a better overview, we will discuss each group of components separately.

- Planning (WA) App PWA

  This group consists of a web-based application (W) that stores the data in the database (D), displays map tiles served by the geospatial server (G) and uses a routing server (R) for calculating optimal vehicle movements. This application is used for inspection planning purposes and facilitating inspection search logic.

- Planning (RA) App PRA

  This group consists of near identical components to those in the WA group. The difference lies in the subset of available functions. The RA version is a stripped-down WA version in terms of functionality.

- Inventory App I

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The structure of this group is straightforward. There are only two components – a core web-based application (W) and a database for data persistence (D). This application is used for inventory purposes to keep track of the equipment used during an OSI.

- Field App F

  It consists of an Android application (A) which is connected to a database (D). The database is embedded within the app; however it is set as a separate entity on the diagram. This application serves the purpose of collecting and aggregating data from the field.

- LabField App LF

  The structure is very similar to the one of Field App F. The difference lies within the connection types when communicating with other components. The application is an intermediate between the Lab App L, a web-based application, and two other systems – Planning (RA) App PRA and Field App F. It is mostly used for data transfer and consolidation.

- Lab App L

  This group is structured like Inventory App I, which was described previously. The main function of this application is mobile Laboratory data processing and remote management of auxiliary equipment for detecting levels of radioactivity of environmental samples collected in the field.

- Documentation App D

  This group is structured like Inventory App I, which was described previously. The application stores documentation in a structured way for simple search and retrieval.

- Java App J2

  This is a Java-based application that provides a RESTful interface. It is used inside a virtual machine for passing input data of measurements for further analysis. Also, it collects output data – analysis reports provided by users.

- Java App J1

  As Java App J2, it is a Java-based application that provides a RESTful interface. It triggers measurements on external equipment, as well as reads output data when a measurement is done.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

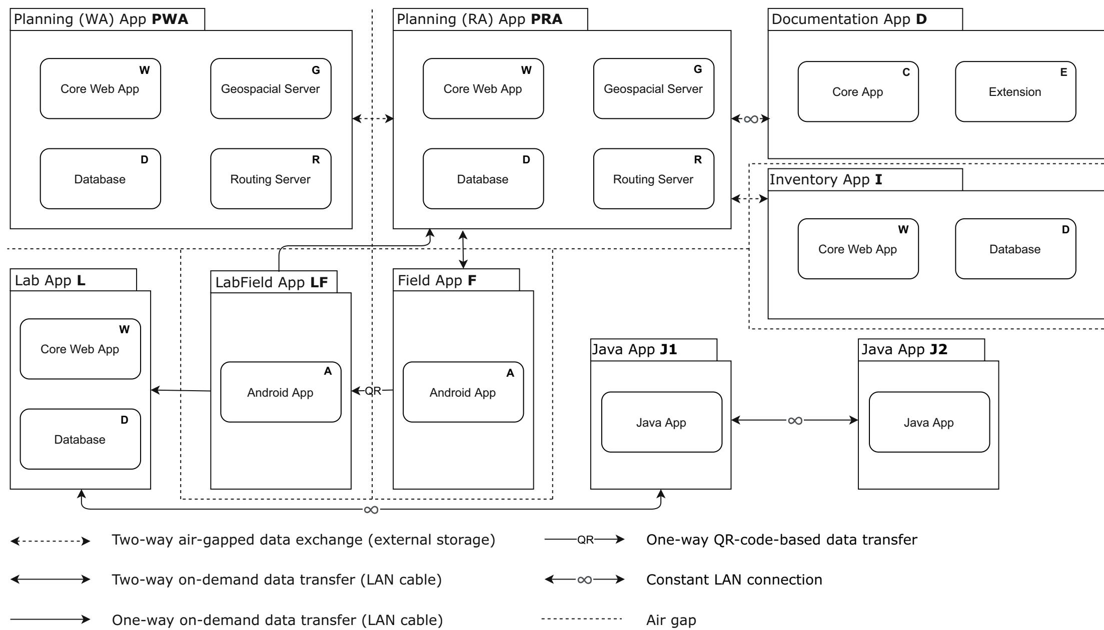

Figure 4.1: Example software setup

4.6 Global vs local knowledge

Air-gapped environments assume complete isolation of networks or software systems. To achieve resolution of application-level dependencies on a completely isolated node, global knowledge about all related nodes must be combined with the local one. We denote global knowledge as an assembly of information on the installed software component(s), which is irregularly collected/updated, manually or automatically. To assess such information automatically, each of the nodes can be visited separately. Network isolation is preserved since none of the nodes is really exposed to the outside networks. The problem however, is that locally a component can be deleted, updated or altered in any way, so that globally this change is not propagated. Therefore, local knowledge shall always represent the true state of a component, whereas, globally, the knowledge may or may not be true. We say *global knowledge* is based on assumptions, and *local knowledge* is the single source of truth. To denote a set of all pieces of local knowledge combined, we say *global true knowledge* which is usually unknown.

Further, we formalize the introduced notion of local and global knowledge. To avoid any misunderstandings regarding the used wordings, when we say "system", we mean a set of "applications", and each "application" consists of further components/services. For instance, PRA from the example system described in 4.1 would be an "application" consisting of multiple "components" (backend, frontend, database, etc.). Additionally, each component can have multiple versions; thus when we say "installed components", we mean actual concrete component versions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Let PRA_v be a set of combinations of all possible valid component versions in respect to a particular component's version v of PRA application. Let $INST_{pra}$ be a set of all currently installed components of the PRA application (local knowledge), $INST_g$ be a set of all applications/components presumably installed globally (global knowledge) and $INST_{gt}$ a set of applications truly installed globally (true global knowledge). If an update is applied to one of the application PRA components to version v, the compatibility with the rest of the installed applications should be verified. However, there is no certainty which applications and components do $INST_{gt}$ contain without visiting each node beforehand. Thus the question arises what component versions should be shipped or taken while visiting the nodes.

Let $SHIP_{pra_v}$ be a set of applications/components to ensure a successful update of PRA's component to version v. In other words, it is a shipment of artefacts representing components versions. Considering we have only global unreliable knowledge when the decision is carried out, it is not straightforward which artefacts of which components must be included in $SHIP_{pra_v}$. Shipping all possible versions of all artefacts may actually be a legitimate solution, if (a) time used for copying is not an issue and (b) the storage has enough capacity to store all the artefacts. However a more efficient solution is highly desired, so a reduced set RS_{pra_v} where $\exists S \subseteq RS_{pra_v} \mid S \in PRA_v$ of needed components should exist. A minimal reduced set, $MRS_{pra_v} \subseteq \mathrm{RS}_{pra_v}$, of needed components to include into $SHIP_{pra_v}$ is defined as follows:

$$MRS_{pra_v} \mid S = MRS_{pra_v} \mid \exists S \in PRA_v \wedge S = SHIP_{pra_v}$$

Thus, for an update to succeed, the shipped artefacts have to be a subset of the minimal reduced set: $SHIP_{pra_v} \mid MRS_{pra_v} \subseteq SHIP_{pra_v}$.

As discussed previously, *global true knowledge* $INST_{gt}$ is unknown unless explicitly collected, and *global knowledge* $INST_g$ is based on assumptions. Since the minimally reduced set MRS_{pra_v} is unreliable to calculate on mere assumptions and shipping all possible versions of all artefacts may be impossible as well, let alone inefficient, shipping all the artefacts of PRA_v may be a feasible solution, as $MRS_{pra_v} \subseteq S \mid \exists S \in PRA_v$ However, the size of PRA_v is variable and can reach a significant magnitude.

The introduced global and local knowledge helps us to answer the RQ4 from Section 1.3.

4.7 Predictability

We argued before, $INST_g$ is based on assumptions and is unreliable, but it can be fully accurate as well. This accuracy or predictability can be expressed by an arbitrary value $p_g \in (0, 1]$. Lower value means less accuracy and less coverage of the the minimal reduced set MRS_{pra_v}, 0 represents no coverage at all. Higher value means more accuracy and more coverage of the the minimal reduced set MRS_{pra_v}, 1 represents a full coverage. Similarly, c_{pra} with the same range from 0 to 1, can express the completeness of PRA_v. In other words, a ratio of how many elements of sets of PRA_v are included in $SHIP_{pra_v}$,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

however the elements with higher probability to be used are included first. For this, let us define a function $f(c)$ which accepts the completeness ratio c of an arbitrary set like PRA_v and returns a set of elements with the highest probability to be used. So the elements to include into the shipment from PRA component of version v are defined as follows:

$$SHIP_{pra_v} = f(c_{pra}) \mid 0 \leq c_{pra} \leq 1$$

Even though c may be set to 0, and f(c) would return no elements, this behaviour is not desirable; it is impossible to update a component without shipping any elements at all. Another lower bound for c must be defined, so that c > 0. Thus, we introduce *cmin* to express the minimal ratio, so that at least the elements based on global knowledge $INST_g$ are included:

$$SHIP_{pra_v} = f(c_{pra}) \mid cmin_{pra} \leq c_{pra} \leq 1$$

We want to point out that only if we ship all possible combinations for installation of a component of version v, such shipment is guaranteed to be fully accurate: $(c = 1) \implies (p_g = 1)$. However, it may happen that $(c < 1) \wedge (p_g = 1)$ holds, thereafter it is true that $(c = 1) \not\equiv (p_g = 1)$. Based on this, by increasing c we generally increase the value of p_g.

4.8 Maintaining global knowledge

Even though global knowledge may be unreliable and incomplete, it is still a valuable source of information. Further, a suitable approach to collect and maintain the global knowledge is discussed.

We believe our solution should be non-intrusive, meaning no local agents should be installed and maintained. Another challenge is the mandatory air gap between most of the systems which does not allow constant network connections for global knowledge management. Thus, only on-demand connections can be utilized. Moreover, such connections are almost inevitable during an update procedure. As a result, global knowledge can be collected/updated piece by piece only. Let us call the abstract entity, which is responsible for the collection of local knowledge *agent* for now.

The agent must satisfy the following properties based on previously discussed restrictions and the collected requirements in Section 4.1:

1. Mobility

   The agent is expected to be carried around and physically be attached to different systems/servers. Thus, it has to be portable enough. Having its own source of power is highly desirable.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2. Computational capabilities

   The agent requires computational capabilities, since locally dependencies must be resolved and installed to reach a satisfiable state.

3. Usability

   It is expected the agent handles most of the tasks automatically. However, user interaction is still required. Therefore, the agent must have an external display, and some mean for the user to input a password, IP address, etc. – a physical keyboard or virtual one on the screen.

4. Security & data protection

   Since the agent is allowed to be connected to otherwise air-gapped systems, it must adhere to high-security standards. For example, secure authentication, encrypted communication, etc. Moreover, it is of the most serious concern to prevent any data breach, especially the one collected during an OSI.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek
WIEN Your knowledge hub

Further, we shall discuss how to transform the abstract agent we just described into a real implementation. Firstly, the choice of hardware shall be made. A traditional server is to be excluded right away, since it is not mobile at all. Any mobile server which requires a constant source of external power is not suitable as well. Thus, the left choices are (a) mobile server with its own source of power, (b) traditional/workstation laptop, or (c) small computing device like Raspberry Pi. The a option is the most expensive one, which provides great computational power and potentially great storage capacity. However, there is not much choice on the market, and the extended computational power is probably not needed. The option c seems to be suitable, but an additional external touch display is required. The limited computational power of a small computer like Raspberry Pi is questionable, but may be sufficient. An external source of power, like a USB power bank, can be easily attached. Nevertheless, option b seems to be the best option so far, because a laptop is portable, has its own source of power, has a display and a keyboard, and there is a vibrant choice on the market so that suitable computational power can be easily reached.

The next question to be clarified is should the agent be managing both local and global knowledge or simply the local one. In other words, should it rely on some other entity that has global knowledge or maintain its own? According to the collected requirements, a set of the same software products can be installed at multiple locations; however, potentially, they will differ in versions. To satisfy the user story OVERVIEW in Section 4.4, an overview of what is installed at what location is highly desirable. Thus, it is feasible to provide a separate entity, a server, which should maintain global knowledge across multiple locations based on global knowledge parts collected by the agent per location. Let us call this entity *master*. Since multiple locations are in questions, multiple agents are needed; one agent simply may not always be shared, because locations can be thousands of miles away from each other. To provide global knowledge parts to the

master, e.i. to contribute to its global knowledge, an agent should connect to the master. Obviously, no constant connection is possible due to the air-gapped networks an agent will be connecting to. Moreover, during an OSI, network connection policy to the outside world has not been yet well defined. Thus, an agent cannot rely on the master's global knowledge.

Let us sum up the discussion on agent and master. The architecture of an example setup is depicted on Figure 4.2. There is always at least one agent per location, and multiple locations can share an agent if it is reasonable. An agent is responsible for maintaining global knowledge of the location(s) it is based at. Thus, it collects and updates local knowledge by visiting the nodes when an update or initial provisioning is needed. Master is responsible for providing an overview of all installations across all the locations. Agents connect to the master occasionally and update it with the latest changes of the nodes they serve at their location(s). The example on the Figure 4.2 shows four locations named A, B, C and D. Location A is the main location that hosts the master. Optionally it can have an agent as well. Location B hosts only an agent. Location C and D share one agent, as it is logistically reasonable.

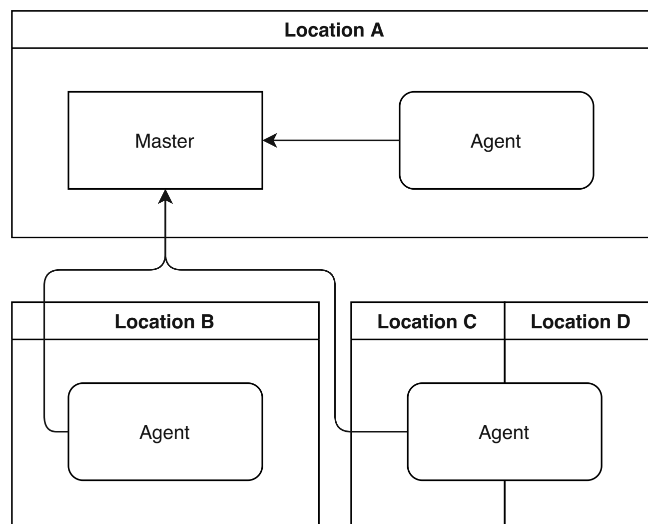

Figure 4.2: Master/agent architecture

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

4.9 Dependency resolution and validation

In the Sections 4.6 and 4.7 we discussed the problem of not having the global true knowledge and what artefacts should be shipped or prepared in advance. The current subsection covers the dependency resolution and validation, which also helps us to answer RQ3 from Section 1.3.

At first, let us describe a couple of scenarios where dependency resolution is applicable.

1. Dependency model validation

   When dependencies of software artefacts are defined, it is needed to validate if the model is satisfiable. For example, component A of version 1.2.* requires component B of version 4.*.*. [1] However, component C requires component B of version 5.1.*. It is clear this model is not satisfiable, as there is no combination of the components which would satisfy all the dependency constraints. It is beneficial to discover and fix such inconsistencies right when the model is defined.

2. Artefacts shipment

   When the user decides to update a node to a certain version or perform initial provisioning, it must be decided what artefacts are needed. Previously in Section 4.7 we extensively discussed how to make shipments as complete as possible so that an update succeeds based on unreliable global knowledge. Thus, at this stage, dependency resolution may deliver more than one suitable set of components.

3. Performing an update or initial provisioning

   When an agent connects to a node to provision or update it, given local knowledge, e.i. what is currently installed on the node, and given shipped artefacts, e.i. what is available for installation, dependency resolution must be made.

A widely adopted way to approach dependency resolution is by incorporating the power and flexibility of SAT-solvers [ADCTZ12, ADCBZ09, LBP08, LB10]. However, the solution approach mainly needs to make a comparison between versions, thus, arguably, an SMT-solver is a better choice. SMT stands for Satisfiability Modulo Theories and adds the ability to handle arithmetic theories on top of the standard SAT-solver capabilities [BT18]. To compare versions, which are essentially strings, lexicographical comparison support would be of great help, but SMT needs extended implementation to encode and use lexicographical comparison constraints [EF14].

4.10 Numerical representation of versions

To simplify the process of lexicographical comparison using SMT solvers, we decided to make version strings convertible to integers to be easily compared and processed by

[1] The star(*) denotes any of the version on the respected level, "micro" or "patch", is accepted.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a solver. Due to the high adoption of semantic versioning, this will be our targeted versioning approach. The creators of semantic versioning do not limit the version numbers, but they say it is meaningful to preston2019semantic. We propose our own conversion procedure. As input it accepts a string of pattern "major.minor.patch", optional pre and post-fixes, and converts it to a number. Each of the three parts is delimited by "." 12 bits are reserved with a range of values: 0 to $12^2 - 1$. The highest version would be then "4095.4095.4095". Optionally 14 bits before (0 to $14^2 - 1$) and 13 after (0 to $13^2 - 1$) are left for custom use, making 63 bits in total. Most programming languages usually use the 64th bit for negative numbers according to the two's complement, a mathematical operation on binary numbers used for signed number representation. To avoid confusion and additional conversion, it was decided to use exactly 63 bits, thus, staying in the positive numbers only. Again, the optional 14 and 13 bits are left for custom usage. For example, the 13 postfix bits can be used to translate commonly used version extensions such as "alpha", "beta", "rc", etc. [2] The Listing 4.1 shows a Java 11 implementation for the proposed conversion approach. In the example implementation we omitted the precondition checks. Those would be, for example, are the values within the allowed ranges or does the string comply with the simver guidelines. This is done on purpose to make the main contribution which is the conversion itself to stand out.

Listing 4.1: Java example

```
private long convert(int pre, String str, int post) {
long[] nums = Arrays.stream(str.split("\\."))
      .mapToLong(Long::parseLong).toArray();
int totalBits = 63, preBits = 14, midBits = 12, postBits = 13;
return pre * (long)Math.pow(2, postBits + midBits*3) |
      nums[0] * (long)Math.pow(2, postBits + midBits*2) |
      nums[1] * (long)Math.pow(2, postBits + midBits) |
      nums[2] * (long)Math.pow(2, postBits) |
      post;
}
```

For example, we convert the following versions encoded as strings: "10.2.9", "10.3.17". The proposed function would convert the strings to 1374456717312 and 1374490337280 in decimal. By comparing the numbers it is clear that $1374456717312 < 1374490337280$, resulting in "10.2.9" < "10.3.17".

4.11 Modelling

In this Section we discuss the design from the modelling perspective of view. Mostly Unified Modelling Language (UML) is used. All diagrams were created using Visual Paradigm software, version 16.2. [3] The modelling does not reflect the actual implementa-

[2] rc stands for release candidate
[3] More information on the official website: https://www.visual-paradigm.com/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

tion precisely. Otherwise, the diagrams would be too cluttered and filled with unnecessary details. By simplifying the models, we want to make it clear and simple to the reader what is the core design of the software solution. The modelling also covers the RQ2 from 1.3.

4.11.1 Use Case Diagram

The Figure 4.3 shows Use Case Diagram modelled in UML. It reflects mostly the user stories defined in Section 4.4.

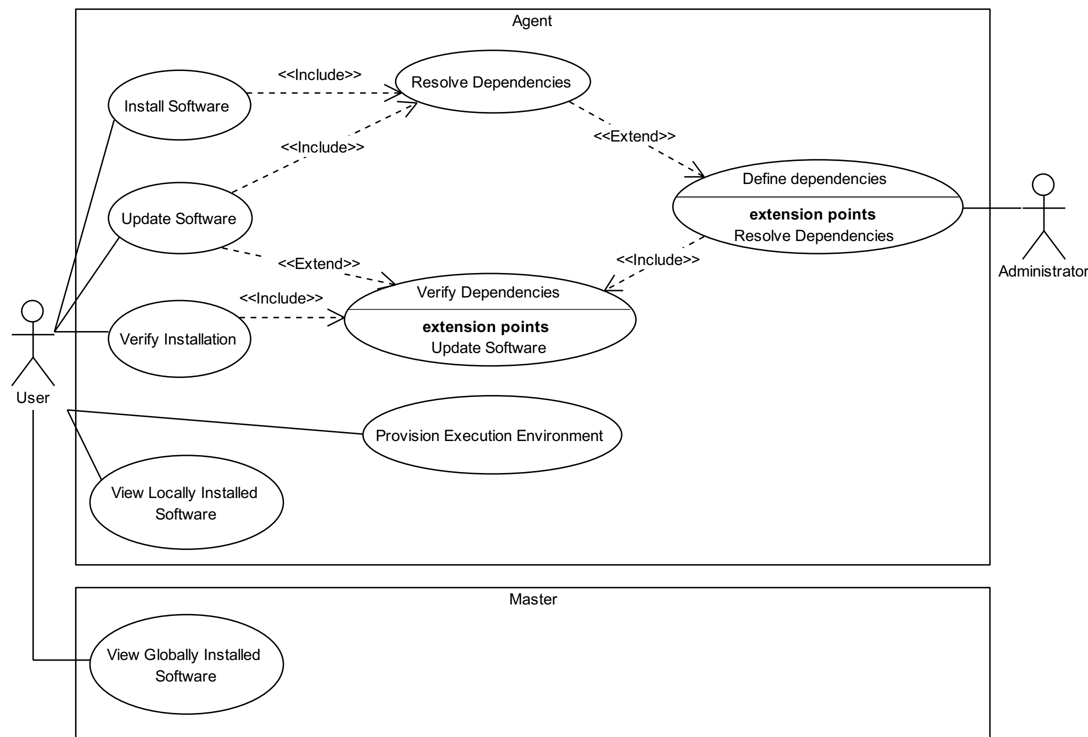

Figure 4.3: UML Use Case diagram

Two main types of users are outlined: user and administrator. A user can be essentially anyone who is interested in performing the stated tasks. Thus, it could be, for example, an inspector, the Data Flow Management Officer or a PTS member. An administrator could be, for example, a software engineer or a person having enough expertise to define software dependency relations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The diagram shows both agent and master, as described in Section 4.8. Both of them share a similar use case: "View Locally/Globally Installed Software". For an agent, all installed software components are in mind at the location(s) the agent is assigned to. [4] Respectively, a master keeps track of all installed software artefacts across all locations.

The main and most important use case is "Update Software", and it is self-explanatory. "Install Software" resembles "Update Software", however it focuses on a brand-new installation. "Resolve Dependencies" and "Verify Dependencies" are indirect use cases that support almost every other use case. "Define Dependencies" refers to the process of defining dependency relations/constraints between software components. "Verify Installation" is an activity of ensuring software components of correct versions are installed across the nodes.

"Provision Execution Environment" use case refers to the process of setting up a new node. For instance, it could be adding its metadata to the system, ensuring it has the needed execution environment(s), store the node's IP address if possible/necessary, perform other similar tasks.

4.11.2 Class Diagram

The Figure 4.4 shows the class diagram of the most important entities that serve as the basis for the application's business logic. Further, we shall explain each of the classes and talk about their roles. The Figure does not depict all properties and methods on purpose, but only the most important ones at this stage: this way we want to avoid cluttering and potential distractions for the reader.

As mentioned previously in Section 4.2, one agent serves multiple locations, so there exists a separate class Location. A Location can have multiple installations (Installation class). An Installation always refers to one software system (SoftwareSystem class), but a software system can have multiple installations. A software system is an aggregation of multiple components (Component class), since components cannot exist independently and are integral parts of a Software System. For example, a typical modern Web application consists of frontend (F), backend (B) and database (D). A frontend is typically the graphical user interface consisting of HTML, CSS and JavaScript files. Backend implements most of the application's business logic and can be implemented used a wide variety of technologies and programming languages. Let us assume there exist an additional native mobile application (M). All these four elements are denoted as components, instances of class Component, and aggregate a software system, an instance of class SoftwareSystem. For convenience purposes, let us say the components F, B and D are interconnected and must be deployed together. Thus, we can add them to one group, an instance of ComponentGroup class. A component does not have to belong to any group, if there is no point in doing so. M is a good example of it. It is additional

[4] As "software component" can be interpreted in many ways, please refer to the beginning of Section 4.6 for an explanation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

software, essentially another GUI, which is installed completely separately. Thus it can be left without a group assignment at all.

To handle different versions of a component, the class ComponenVersion is created. For example, let us assume component F has three versions released: 1.0.0, 1.1.2 and 1.3.1. So an instance of Component F shall be associated with three instances of ComponentVersion, each with a respected version value assigned. To express dependency relations between components, CompDepConstr class is of help. Each instance of this class is associated with two different components. Also, each of the component version strings and rule types must be defined. The rule types are built based on the classifications of version constraints specified earlier in Table 3.1. Essentially, such a rule results in a Boolean expression $(X \implies Y)$, where X denotes satisfaction of the first criterion and Y of the second one. For instance, we want to express the dependency of F on B, such that F with a fixed version of "1.3.1" depends on B of var-patch (var-micro) version of "1.3.*". [5] This dependency rule can be expressed as follows: $(F_{ver} = "1.3.1") \implies (B_{ver} \geq "1.3.0" \land B_{ver} \leq "1.3.n") \mid n = MAX_PATCH_VERSION$. The instances of ConstraintType class act, therefore, like enumerations and are not meant to be changed by the users.

Let us deviate from the abstract entities of the diagram and talk about the more realistic ones i.e., entities that are directly related to execution. As just discussed, a component can have multiple versions, instances of ComponentVersion class. Each component version has at least one or more executable artefacts (ExecutableArtefact class). ExecutableArtefact is an abstract class representing a file – archive, image, etc. – which can be executed in the end. Since our system works primarily with Docker containers, we included DockerImage class which extends ExecutableArtefact on the diagram. The multiplicity is 1 to many, since theoretically, the same component version can have multiple artefacts. Think of a Docker image and a war archive for a Java application. Another example would be a Java jar file with all dependencies included and a jar file without any dependencies.

Every executable artefact is associated with none, one or multiple execution environments, instances of ExecutionEnvironment class. ExecutionEnvironment is also an abstract class. It represents an environment where an artefact can be executed. Multiple other classes have an "is-a" relationship to this class, such as ExecutionHost and VirtualizableExecEnv. The classes VirtualMachine and DockerEngine extend it indirectly. An instance of an ExecutionHost can be associated with multiple instances of VirtualizableExecEnv, meaning a server can host one/many virtual machine(s), a docker engine, etc. As shown on the diagram: DockerImage can be executed by DockerEngine, and Android apk file can be executed by Android OS. Other examples would be a Java jar file as an artefact and Java VM as an environment. Similarly, a dependency constraint can be defined between an execution environment and an executable artefact. To avoid code duplication, ExecEnvDepConstr class which extends DependencyConstraint class is used. The dependency constraints are needed to specify what version of the execution

[5] Please refer to the Table 3.1 for the definition of fixed and var-patch (var-micro) constraints.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

environment is expected to be provided. For example, there is an Android apk executable artefact of version "2.0.1" which suppose to run on Android OS of at-least version "8.0.0". Or the users can specify what version of Docker engine a particular Docker image was tested on and suppose to work with. Similarly, if it is known, some version of some artefact car be incompatible with some specific version of an execution environment. This can be specified using "not" constraint type and denoting a conflict. Even though specifying execution environment constraints is very handy, the main focus lies on managing dependency relations/constraints between components.

4.11.3 Object Diagram

Further, we discuss the object diagram shown on Figure 4.5 which is based on the class diagram from the previous Section 4.11.2. We included an object diagram to help the reader better understand the chosen design depicted on the class diagram.

The diagram shows two installations of the same system named GIMO at the same location named HQ. The installation names are QNAP_room_1102 and onlineTraining. Logically, they refer either to the physical location or to the purpose of the installation. The installation at room 1102 has an execution host named CentOSServer, which hosts a virtual machine named UbuntuVM which hosts a Docker engine. This Docker engine executes two executable artefacts. One of the artefacts has a constraint that if the artefact's version is "1.3.2", the Docker engine's major version has to be at least "18.*.*". One of the executables is associated with component version "1.3.2" of component GIMORABackend (B). There exists a constraint named FrontDepOnBack (F) which expresses the following: if the frontend component is of version "1.3.2" (F_v) exactly, then the backend component must be of at least version "1.3.2" (B_v), but the patch version stays variable. Formally it can be expressed like this: $(F_v = 1.3.2) \implies (B_v \geq 1.3.2 \wedge B_v \leq 1.3.n \mid n = MAX_PATCH_VERSION)$. FrontDepOnBack2 instance expresses a very similar constraint, but with different values. When the dependency resolution of a component is performed, all the associated instances of CompDepConstr are considered.

The other installation named onlineTraining has similarly to the first one an execution host named UbuntuServer which hosts the Docker engine. The Docker engine is associated with again two executable artefacts. One of them is shared with another Docker engine of another installation. Here backend component of version "1.3.2" is in mind. The frontend component differs in version. Since the constraints allow mixing versions like shown on the diagram, this would be a valid state. Additionally, frontend and backend components are parts of one component group. This means during a deployment, they will be considered as one entity.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.11.4 Activity Diagrams

Update Component

The diagram on Figure 4.6 shows activities involved in performing the use case Update Software. It does not try to model all possible outcomes and deviations, but rather to present a rigid plan on how to break down the actions needed to accomplish the task.

At first, the user selects a component to update. Based on the example from Section 4.5, let us assume the user wants to update component A of group F named Field App (FA). The latest version is selected. In accordance with the global knowledge, dependencies are resolved. Next, all needed artefacts are collected (preparation of the "shipment" term from Section 4.6) and, if needed, are prepared to be shipped. Afterwards, a plan on what nodes must be updated/visited is shown. Then the agent disconnects from the internet or any other network and connects to each node from the list. The node then provides its local knowledge. As discussed in Section 4.5, at this point of time the agent possesses true local knowledge and should again resolve dependencies. If there are no unsolvable issues, the update is pushed and performed in the scope of the execution environment. The processes are iterated until the end of the list is reached. Again, as discussed before, the dependencies may reach an unresolvable state. In this case, the problem is reported to the user, possibly advising on how to solve it.

Provision Execution Environment

The activity diagram 4.7 describes the use case Provisioning Execution Environment. Essentially it covers adding a new node, for example, a server with an installed Docker Engine. Let us describe it in more details.

At first, depending on the type of Execution Environment (EE), the user defines such information as IP address, SSH credentials like public/private key pair, etc. Then a secure connection to the EE is established. Afterwards, metadata is retrieved. For example, if it is an EE of type Docker, the returned metadata should contain the version of the installed Docker engine. It can also contain some information about the host like the distribution name, kernel version, etc. Afterwards, this data is verified. If the EE is expected to be of type Docker, logically, the Docker engine must be installed. Additionally, a constraint on its version can be put. After a successful verification, the EE is added to the system as an active one. It means later on it can be used for software provisioning or software updating.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

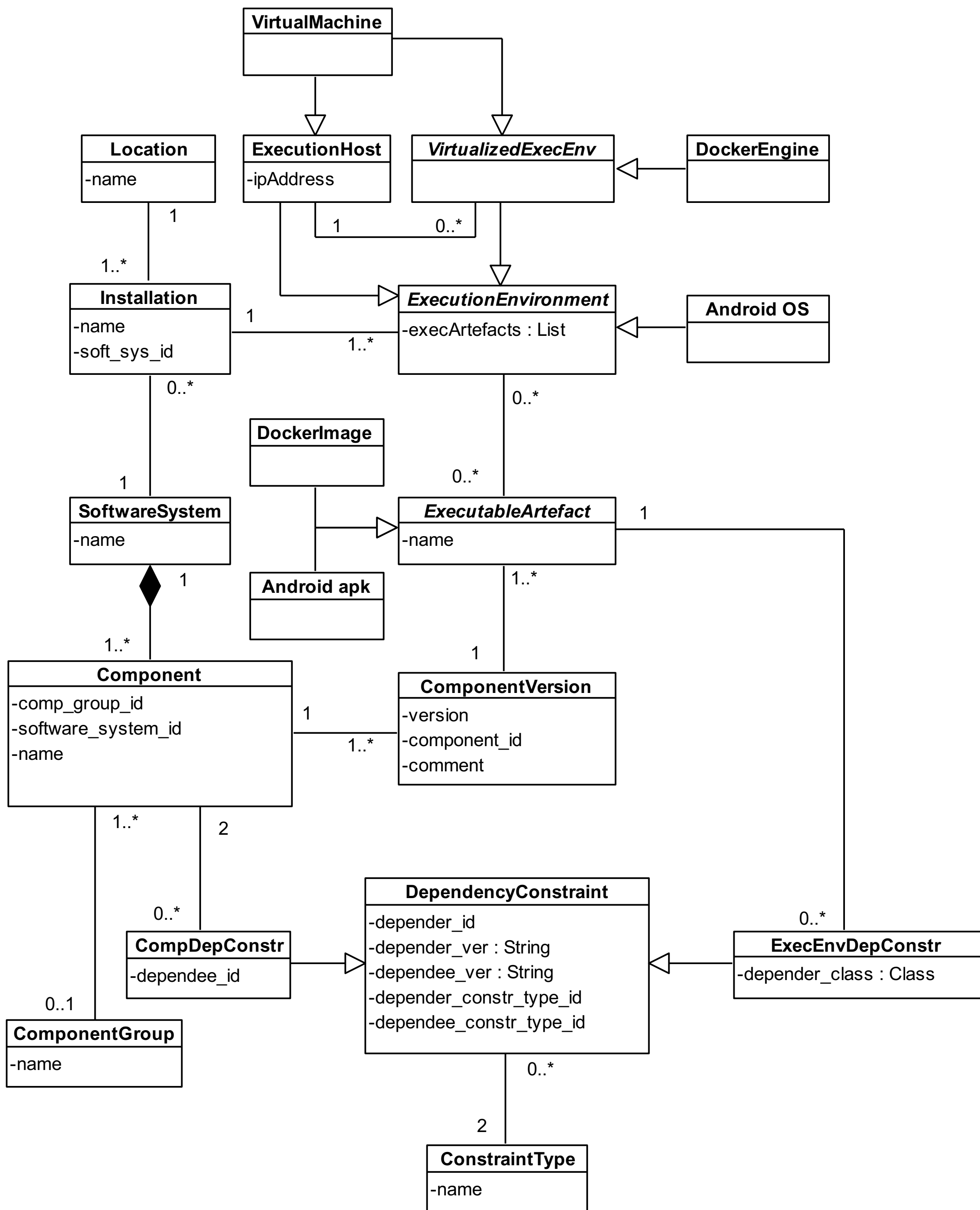

Figure 4.4: UML Class diagram - domain

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

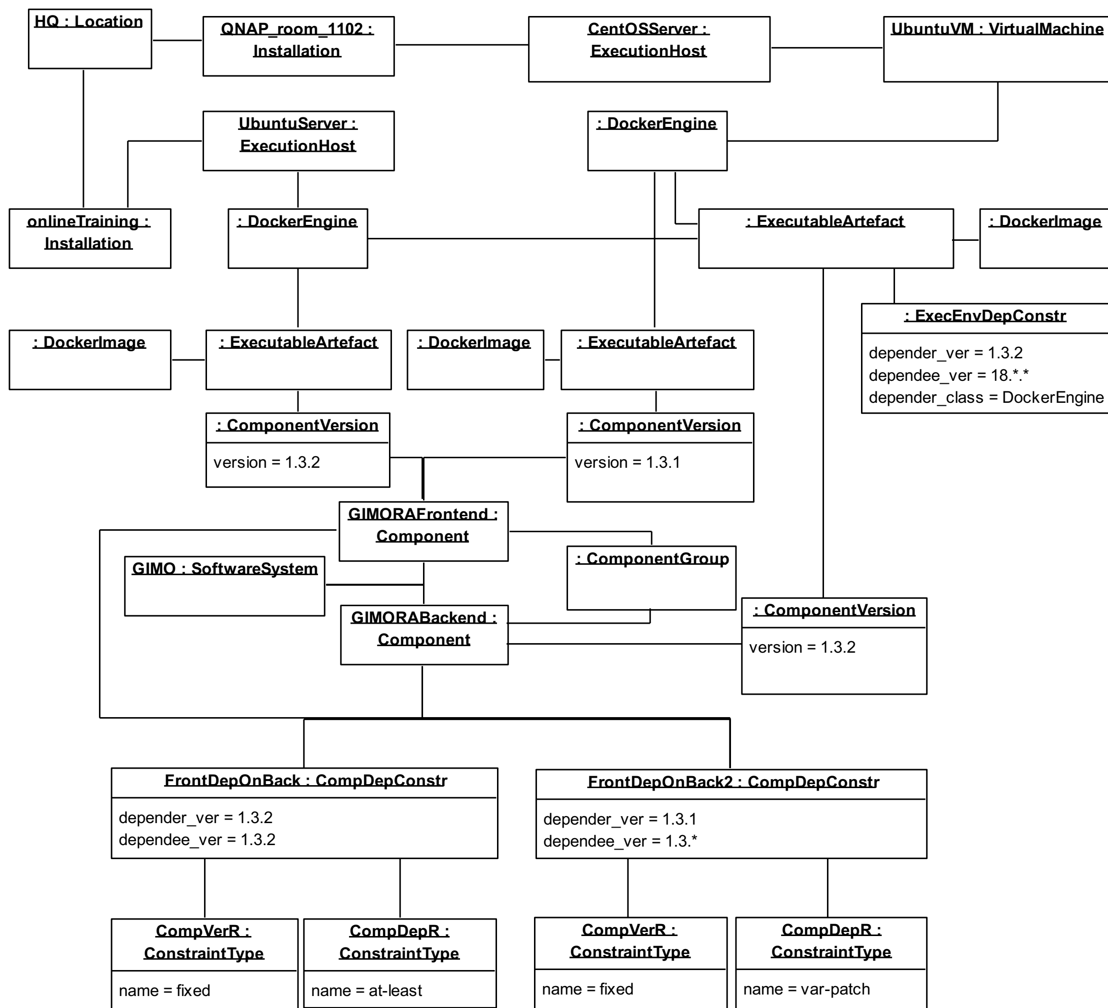

Figure 4.5: UML Object diagram - domain

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

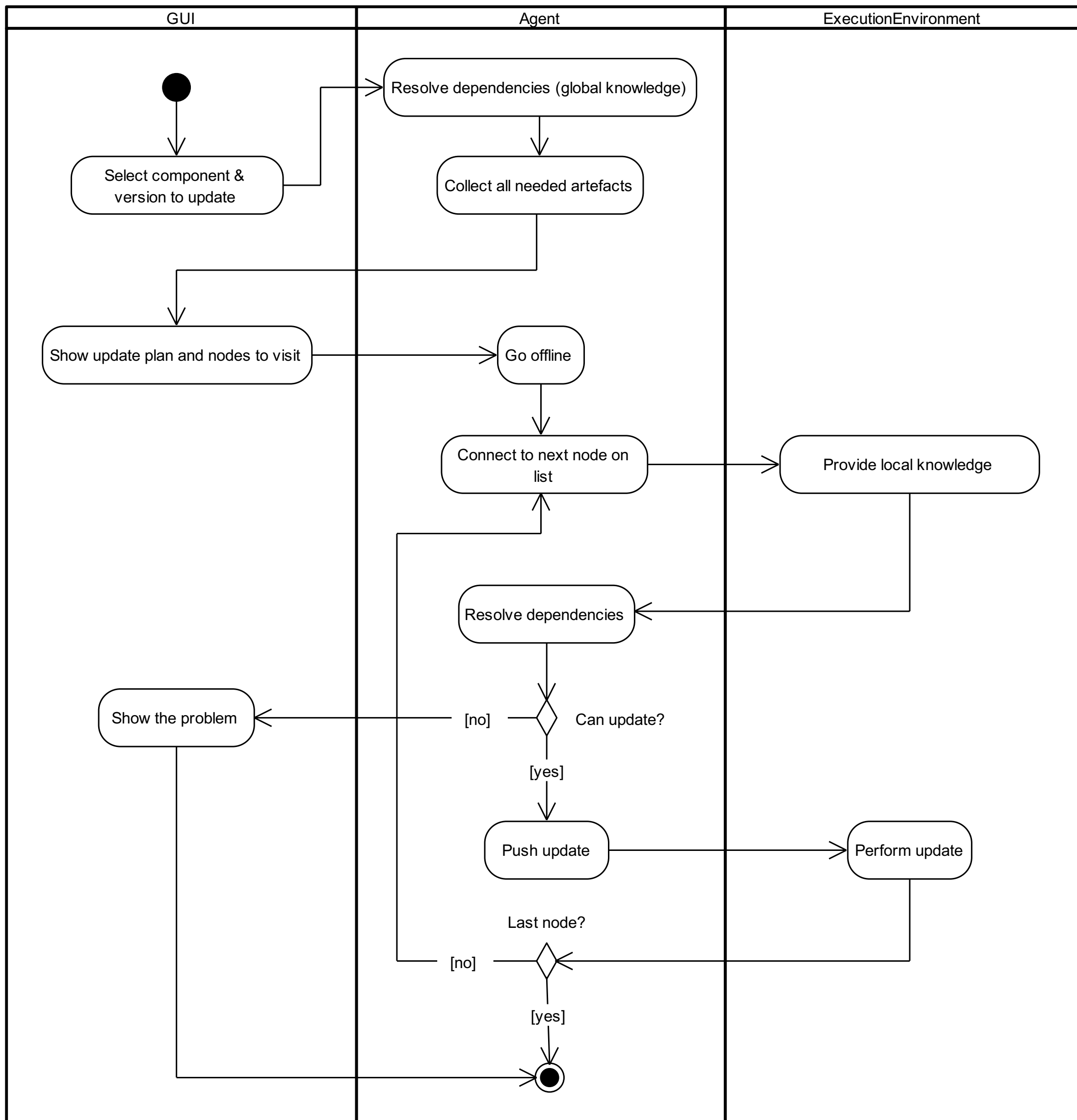

Figure 4.6: UML Activity diagram - update procedure

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

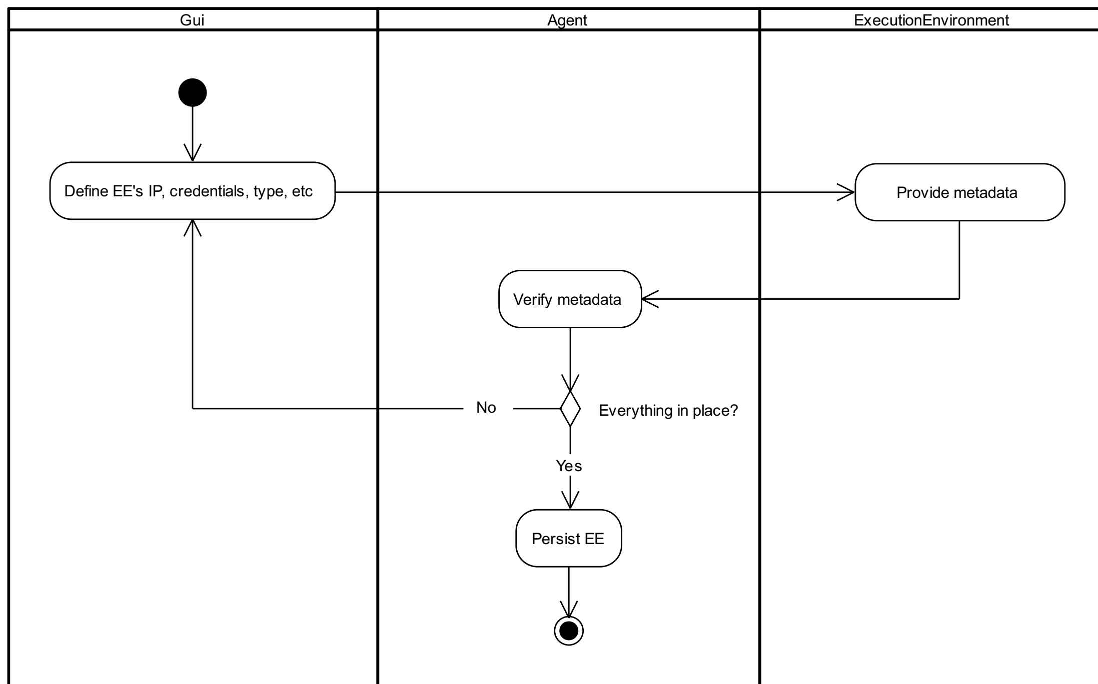

Figure 4.7: UML Activity diagram - provision execution environment

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Dependency Resolution with SMT

As previously illustrated in Sections 4.11.1 and 4.4, updating a component is one of the core functionalities expected to be implemented. Let us assume that the application (component group) PRA from the example system (see Section 4.5) must be updated to a newer version. Even though the Figure 4.1 shows mainly dependencies between component groups, within a group, there could be dependency relations between components as well. In 4.6 we argue global knowledge is unreliable, because local knowledge can be altered anytime without us knowing. To combat this problem, we introduce "c" which stands for completeness. Based on this, we constructed function "f" that resolves the dependencies and returns a set of suitable component versions. In this work, however, we focus first on dependency resolution using global knowledge as input. A proper implementation of f, which considers the previously discussed probability and completeness, requires development of a dedicated and sophisticated algorithm, which is out of the scope of the thesis.

This chapter discusses how to model the required dependency resolution procedure using SMT by elaborating on the problem first and coming up with multiple solution strategies next. Further, we describe core algorithms using pseudocode to decouple them from a concrete implementation. Finally, we shed light on the actual implementation with PySMT in details.

5.1 Problem Formulation with SMT

5.1.1 SMT Basics

Satisfiability Modulo Theories (SMT) allows to combine multiple theories and helps to solve Boolean Satisfiability (SAT) like problems. This makes it possible to take advantage

of different useful operations a classical SAT-solver does not provide. For example, one can compare integers, solve equations or perform similar algebraic operations. This versatile extension comes in handy when variables are to be constrained to certain values or ranges of values. In Section 4.10 we discuss how version strings can be converted to integers. This conversion can be utilized when comparing/constraining versions with SMT. Let us remind you, a constraint can be expressed using implication. For example, component A of version "1.0.3" requires component B of version "1.1.8". Since it is an exact match, it can be encoded as: $A = toInt(\text{"1.0.3"}) \implies B = toInt(\text{"1.1.8"})$. The toInt function intuitively returns the numerical representation of the passed version string. Later we explain how a simple implication can be applied with a more advanced dependency relation to constrain the values to a certain range.

5.1.2 Problem Statement

Let us further describe the problem statement concerning how to express the SMT formula to implement the expected dependency resolution functionality.

Given (1) a set of installed components of certain versions, (2) a component of a newer version we want to update to and (3) a set of constraints describing dependency relations between the components, is such a constellation satisfiable? If it is, no further actions are needed, as the component can be safely updated. If it is not satisfiable, additional information is required to find new optimal dependencies. This includes sets of available versions per component. So essentially, the information at hand can be divided into **Domain Constraints** and **Facts**. Domain Constraints are all those rules which define component dependency relations. For example, component A of patch-variable version "18.2.*" requires component B of fixed version "1.2.5". Domain Constraints must always be satisfied and can never be left out. **Facts** can be subdivided into: **must-be**, **exactly-one**, and **is-variable**. The Table 5.1 describes them in more details.

Let us narrow down and summarize the problem statement:

Given must-be fact(s) (version(s) user wants to update to), exactly-one fact(s) (what versions do exist), is-variable facts (what version are already installed), and domain constraints (version dependency relations), get a set of components and versions that satisfies all facts and constraints.

5.1.3 Solution Approaches

In Section 3.2.4 we mentioned dependency resolution problem is NP-Complete. One of the major difficulties is that depending on the number of component versions, how restrictive the constraints are, etc., the resolution can easily explode and take too much time to get processed. As a countermeasure, we present an optimization strategy, since it is necessary. This strategy (called MAX-VER) iteratively downsizes the problem by determining only the maximum satisfiable versions. Moreover, we cover another less optimized dependency resolution approach called ALL-VER. Further, the mentioned approaches are described in more details.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Fact Type	Description
must-be (upgrade-versions)	One or more versions of different components must be installed. Usually, a must-be fact reflects a version of a component the user wants to update to. For example, the user would like to upgrade component A to version X and component B to version Y, because of some newly introduced feature in the first one and a recently fixed bug in the second one.
exactly-one (available-versions)	When a must-be version of a component is defined, it may result in a mandatory update/downgrade of another component due to some domain constraint. However, it would be computationally-wise inefficient to look for all possible values without bounding them. Let us say, component A with patch-var constraint "7.5.*" depends on component B with minor-var constraint "10.*.*". Even if component A was bound to one version, the number of all satisfiable versions of component B would be as high as $2^{24}-2$. Thus, bounding this type of facts to a much smaller set would be beneficial. For example, the set could contain all available versions of component B: "9.8.9", "9.9.0", "10.0.1", "10.0.3", "10.1.0". The number of versions is therefore drastically downsized. Because exactly one and only version must be chosen, we call this fact type exactly-one.
is-variable (installed-versions)	When an update is performed, components of specific versions are already installed. It is possible dependency resolution results in a mandatory update for another component. If a must-be fact does not bind this component's version, we denote it as is-variable. For example, there are three components A (1.0.2), B (2.5.1) and C (0.1.3). For an update to succeed, A must be updated to version 1.1.0 (must-be fact), but B and C are unbound (is-variable facts). Dependency resolution showed that B must be updated to 2.5.2 to be compatible with A (1.1.0), but C can keep the version 0.1.3.

Table 5.1: Types of Facts (dependency resolution)

Finding Maximum Versions - MAX-VER

The MAX-VER aims to reduce the calculation time by finding only the maximum component versions. In the proposed procedure, we apply a commonly-known divide and conquer paradigm.

Imagine we try to solve a dependency resolution problem with Domain Constraints and Facts in place, as per our definition. Let us suppose there are n components (is-variable facts) with m versions per component (exactly-one facts). For each component all valid versions v are determined using an SMT solver, so it holds that $1 \leq v \leq m$. To increase the solver's efficiency, after a valid version is found, it is excluded from the SMT model. When all valid versions of the component are determined, the maximum one is selected.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

This maximum version is added to the must-be facts, which narrows down the problem. The process is repeated until the maximum version for each of the components are resolved.

Alternative Approach - ALL-VER

Apart from introducing optimization on how to constrain the problem more and thus reduce the computational time, we would also like to cover an alternative approach that finds all viable versions (ALL-VER). Because SMT/SAT solvers have become much more efficient within the last half of the decade [ES03, MMZ+01], we believe finding all feasible solutions is still a valid option, although should be used wisely. We fully understand this approach does not scale in all scenarios. However our tests as described in Section 7.3 based on the example system from Section 4.5 showed that dependency resolution for a system with around 20 components and around 1000 versions (50 per component) takes still a reasonable amount of time. Since it is advantageous to get all feasible solutions in certain cases, we would like to leave this option as a secondary solution.

5.2 Dependency Resolution

5.2.1 First-order logic formula

We construct a first-order logic (FOL) formula to describe MAX-VER and ALL-VER formally. At first, we describe the similarities and then cover the differences. For readability purposes "constraintHolds" subroutine and its inner subroutine "constraint" were created. The formula is described in reversed order, meaning the complete formula is presented in the end. Please refer to Table 5.2 for further explanation of the used notation. You can find a textual explanation of each of the formulas afterwards.

constraint(T, v)

$(v.type == fixed \wedge T == v) \vee (v.type == patch_var \wedge T >= v \wedge T <= max_patch) \vee (v.type == minor_var \wedge T >= v \wedge T <= max_minor)$

This subroutine bounds the allowed version range of a target component version (T) depending on the constraint's type. Fixed, patch_var, minor_var are enums; max_patch and max_minor are constants. V in lower case (v) signalises only one version.

constraintHolds(T, C)

$constraint(T, C.dependerVer) \implies constraint(T, C.dependeeVer)$

If and only if the constraint of depender holds, the constraint of dependee must hold as well.

Complete formula

There exist a precondition to the formula: the explicitly set target version(s) of component(s) has/have to hold. For example, a component A must be of version X, because the user defined it. Following is the complete formula for **ALL-VER**.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Notation	Explanation
T	Target component version variable. The SMT solver determines it, and it represents a suitable version per component.
TS	Target component version set. The SMT solver determines it, and it represents multiple suitable versions per component.
C	Constraint, which describes dependency relation. For example: if component A is of version X, then component B must be of version Y.
V	Set of available versions per component. Example: there exist versions X and Y of component A. Essentially, its main goal is to provide options for the solver when it resolves a version for a component.
v	Lower case v denotes just one available version per component.
==, >=, <=	Comparing operators. Operator "==" returns true if both left-hand and right-hand operands are equal. Operators ">=" and "<=" are commonly well-known for combining "greater" and "less" checks with equality comparison.
.	The dot (.) notation serves as access operator to retrieve another associated value. It is similar to commonly used operator in most of the object-oriented programming languages. For example, "C.dependerVer" means to retrieve the depender version of the constraint (C). "v.type" means to retrieve the type of the version (v).
fixed, patch_var, minor_var	These can be described as enumeration types in analogy to most of programming languages.
max_patch, max_minor	These are constants representing a specific value each, depending on the maximum supported version for patch or minor parts.

Table 5.2: FOL notation

$$\forall T \forall C \forall V : constraintHolds(T, C) \land \exists v \in V : v \in T$$

For all target component versions (T), which are also the output, for all constraints (C) and for all set of versions per component (V), each constraint should hold for each target component version, and all the rest of the components should be assigned to one of the versions available per component.

MAX-VER difference to ALL-VER

The MAX-VER approach is generally very similar to ALL-VER. The major difference is that after each iteration, the maximum version of the delivered resulting set is returned. For example, the ALL-VER routine returns the following set of versions for component A: "5.0.2", "4.9.5", "5.1.0", "4.6.4". MAX-VER would return only "5.1.0" as it is the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

maximum or the most recent one. So the complete formula for MAX-VER deviates only slightly from the ALL-VER one.

$$\forall t \forall c \forall V : constraintHolds(t, c) \wedge \bigoplus_{v \in V} v == t$$

Time limit

Even though the tests conducted in Chapter 7 with a realistically high number of elements show acceptable results in terms of computational time, this approach cannot scale. Thus a time limit or budget can be introduced. At first, the idea is to calculate an arbitrary feasible result, a valid combination of component versions, and within the rest of the given time get as many feasible results as possible. Once the time budget is depleted, stop and return the results. This way, the resulting set always contains at least one feasible combination and delivers all the rest of the valid combinations using the best-effort approach.

5.2.2 Pseudo code description

This Section describes the algorithms used to implement solution approaches to the SMT modelled problem as outlined in the previous Section. The intent is to present the solutions in pseudocode to decouple them from the actual implementation, which shall be discussed in the next section.

The Algorithm 5.1 shows how the core SMT problem is composed. It shows the problem composition without diving into the details. The Algorithm 5.2 depicts the construction of a constraint depending on the type. The last Algorithm 5.3 shows how the exactly-one facts are assembled.

Algorithm 5.1: Construction of the problem

input : Component to update $compToUpd$, version to update to $verToUpd$, dependency constraints as array $depConstraints$, rest of the components as array $restComps$, all component versions as map $compVers$

output : Problem composed of facts and domain constraints.

1 domain ← `toConstraint`($depConstraints$, `union`($compToUpd, restComps$))
2 facts ← `equals`($compToUpd, verToUpd$)
3 facts ← `and`(facts, `disjoin`($restComps, compVers$))
4 problem ← `and`(facts, domain)
5 **return** problem

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.2: Construction of the constraints

```
1  Function toConstraint(depConstraints, components) is
       input  : Dependency constraints as array depConstraints, components as
                array components
       output : Array of constraints.
2      constraintSet ← new List;
3      foreach constraint in depConstraints do
4          Repeat for dependee and depender of the constraint as constrVer for
            related component
5          if constrVer.type is FIXED then
6              equals(component, constrVer)
7          else
8              if constrVer.type is PATCH_VAR then
9                  and(GE(component, constrVer), LE(component,
                    toHighestPatch(constrVer)))
10             else
11                 if constrVer.type is MINOR_VAR then
12                     and(GE(component, constrVer), LE(component,
                        toHighestMinor(constrVer)))
13                 end
14             end
15         end
16         constraintSet.add(implies(dependeeRule, dependerRule))
17     end
18     return constraintSet
19 end
```

5.3 PySMT Implementation

PySMT is an open-source Python high-level library that simplifies working with SMT. It serves as an intermediary between the SMT-LIB and solvers API. [1] The main advantage of this approach is the freedom of choosing the underlying concrete solver implementation.

PySMT uses a notion of symbols that are commonly known in SAT problems as literals. A symbol is a variable of different types: integer, number with floating point, string, etc. In our implementation, we use symbols to find viable versions of components.

As PySMT does not provide out-of-the-box support for strings comparison, which is essential for comparing versions, we came up with a version-to-integer converting procedure described in Section 4.10. Further, it is frequently used and is hidden behind the "get_numerical()" method of a component version object.

[1] For more information, refer to https://pysmt.readthedocs.io and https://smtlib.cs.uiowa.edu

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm 5.3: Construction of the exactly-one facts

```
1  Function disjoin(components, compVersions) : int is
       input  : Components as array components, version of all components as
                map compVersions
       output : List of rules for expressing the exactly-one facts
2      andSet ← new List;
3      foreach component in components do
4          orSet ← new List;
5          foreach version in component.versions do
6              orSet.add(or(equals(component, version)));
7          end
8          andSet.add(and(orSet));
9      end
10     return andSet
11 end
```

5.3.1 Domain Constraints

The Listing 5.1 shows how a domain constraint, e.i. dependency relation, is encoded with PySMT. The function accepts a constraint and the symbols. Each symbol represents a component. The symbols are used to find satisfiable versions of the respected component, which is specific to PySMT. The constraint consists of two parts describing a range of valid values for the depender (which requires) and the dependee (which is required). Each of the parts can be constrained differently. The dependee constraint should be considered only if the depender constraint is true.

Each of the constraint parts can be of the types: FIXED, PATCH_VAR or MINOR_VAR. In case of the FIXED type, it is just implied to be equal to the numerical value. [2] With PATCH_VAR or MINOR_VAR the clause must be expanded to this form: $symbol \geq starting_version \wedge symbol \leq max_version$. The max_version is either the maximum patch or minor version. Since the versions are converted to their numerical representations, it is possible to constrain the values to a range.

Listing 5.1: Encoding domain constraints with PySMT

```
def __to_implies(self, constraint:
  ↪ ComponentDependencyConstraint, symbols):
    def convert(comp_ver: TypedComponentVersion):
        symbol = symbols.get(comp_ver.component_id())
        if comp_ver.type == ConstraintType.FIXED:
            return Equals(symbol, Int(comp_ver.get_numerical()))
```

[2] Here and further on numerical value representation of the version is meant.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
        elif comp_ver.type == ConstraintType.PATCH_VAR:
            return And(GE(symbol, Int(comp_ver.get_numerical())),
                    LE(symbol, Int(ComponentVersion.
                        ↪ get_highest_patch(comp_ver.get_numerical
                        ↪ ())))))
        elif comp_ver.type == ConstraintType.MINOR_VAR:
            return And(GE(symbol, Int(comp_ver.get_numerical())),
                    LE(symbol, Int(ComponentVersion.
                        ↪ get_highest_minor(comp_ver.get_numerical
                        ↪ ())))))
    return Implies(convert(constraint.depender_comp_ver),
        ↪ convert(constraint.dependee_comp_ver))
```

5.3.2 Facts

The Listing 5.2 shows how the must-be and exactly-one facts are applied. The must-be facts are handled straightforwardly. The symbol must equal the value according to the must-be fact. Theoretically, multiple must-be facts can be defined, but the implementation is focused on just one so far.

The exactly-one facts are added by disjuncting all possible versions per component, meaning a component can be either of the versions. Each of the disjunction is then conjuncted, like any other fact.

Listing 5.2: Encoding facts with PySMT

```
def __disjoint(self, all_comps: [Component], symbols_dict):
    return And(
        [Or([Equals(symbols_dict.get(comp.id()), Int(v.
            ↪ get_numerical())) for v in comp.comp_versions()])
            ↪ for
         comp in all_comps])

def __get_facts_for_fixed_versions(self, symbols_dict, comps: [
    ↪ Component]):
    if comps:
        return And([
            And(Equals(symbols_dict.get(comp.id()), Int(comp.
                ↪ comp_versions()[0].get_numerical()))) for comp in
            comps]
        )
    else:
        return And()
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
# Add hard facts -- the versions must be satisfied
facts = self.__get_facts_for_fixed_versions(symbols_dict, (
   ↪ comps_with_ver_to_lock + installed_comp_vers))

# Limit to what versions are there
facts = And(facts, self.__disjoint(rest_comps, symbols_dict))
```

5.3.3 ALL-VER

The Listing 5.3 shows the encoding of the ALL-VER resolution strategy described in Section 5.1.3. Most of the used functions were already introduced previously. The new function named "___all_smt" is defined at the beginning of the listing. This function depicts the solving process in its essence. The formula (problem) is passed to the solver, and then iteratively the partial solutions are retrieved, stored in the dependency tree and lastly excluded for further search. This process is repeated until all partial solutions are not found. A partial solution is a combination of the PySMT literals representing a feasible combination of component versions.

The function "apply_all_sat" accepts "comps_with_ver_to_lock" argument which contains: component version(s) the user wants to update to or component version(s) that are/is currently installed (global knowledge). In either case, these component versions must be locked. In certain cases, the installed component versions may require an update as well, thus they may be omitted. By excluding this information, the computation time is expected to increase, but it provides more flexibility. Since ALL-VER delivers all feasible component versions anyway, a needed upgrade for an already installed component can be easily derived afterwards.

As pointed out in Section 5.1.3, we are fully aware this approach is computationally expensive. As argued in certain situations, it can be still attractive to have all feasible solution. Moreover, the performance tests (see Section 7.3) showed good results for problems of reasonable sizes.

Listing 5.3: ALL-SMT resolution strategy (PySMT)

```
def __all_smt(self, formula, symbols_dict, comp_root: Component,
   ↪  rest_comps: [Component]) -> DependencyTree:
   tree = DependencyTree(comp_root)
   tree.append_components(comp_root, rest_comps)

   target_logic = get_logic(formula)
   for k in symbols_dict.keys():
      with Solver(logic=target_logic) as solver:
         solver.add_assertion(formula)
         while solver.solve():
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
            partial_model = [EqualsOrIff(symbols_dict.get(k),
                ↪ solver.get_value(symbols_dict.get(k)))]
            for pm in partial_model:
                ver_str = ComponentVersion.to_str(int(str(pm.
                    ↪ args()[1])))
                comp_id = int(str(pm.args()[0].symbol_name()))
                tree.append_version(Resolver.
                    ↪ find_comp_ver_in_comps([comp_root] +
                    ↪ rest_comps, comp_id, ver_str))
            solver.add_assertion(Not(And(partial_model)))
    return tree


def apply_all_sat(self, comps_with_ver_to_lock: [Component],
    ↪ dep_constraints: [ComponentDependencyConstraint],
    ↪ rest_comps: [Component]) -> DependencyTree:

    symbols_dict = self.__to_symbols_dict(comps_with_ver_to_lock
        ↪  + rest_comps)
    # Add all constraints
    domain = And([self.__to_implies(c, symbols_dict) for c in
        ↪ dep_constraints])

    # Add hard facts -- the versions must be satisfied
    facts = self.__get_facts_for_fixed_versions(symbols_dict,
        ↪ comps_with_ver_to_lock)

    # Limit to what versions are there
    facts = And(facts, self.__disjoint(rest_comps, symbols_dict)
        ↪ )

    problem = And(domain, facts)
    self.__assert_problem_is_sat(problem)

    return self.__all_smt(problem, symbols_dict,
        ↪ comps_with_ver_to_lock[0], rest_comps)
```

5.3.4 MAX-VER

The Listing 5.4 shows the encoding of the MAX-VER dependency resolution approach. It differs from ALL-VER mostly in the last part. Instead of finding all solutions for all literals (all satisfiable versions per component), this approach iteratively finds only the most recent versions. Let us further describe an iteration in more details. For one of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

components, all feasible solutions (versions) are determined. Afterwards, the maximum one is selected and is added to the hard (must-be) facts. As a result, during the next iteration for the next component, the problem will be more constrained and take less time to get solved. The process is repeated until all maximum version are found.

The performance tests in the Section 7.3) show prominent results when this strategy is applied.

Listing 5.4: MAX-SMT resolution strategy (PySMT)

```
def __all_smt_one_symbol(self, formula, symbol) -> [str]:
    result_comp_vers: [str] = []
    with Solver(logic=get_logic(formula)) as solver:
        solver.add_assertion(formula)
        while solver.solve():
            partial_model = [EqualsOrIff(symbol, solver.get_value(
                ↪ symbol))]
            for pm in partial_model:
                ver_str = ComponentVersion.to_str(int(str(pm.args()
                    ↪ [1])))
                result_comp_vers.append(ver_str)
            solver.add_assertion(Not(And(partial_model)))
    return result_comp_vers

def apply_max_sat(self, comps: [Component], dep_constraints: [
    ↪ ComponentDependencyConstraint],
                  comps_with_ver_to_lock: [Component] = None) -> [
                      ↪ ComponentVersion]:
    symbols_dict = self.__to_symbols_dict(comps)
    # Add all constraints
    domain = And([self.__to_implies(c, symbols_dict) for c in
        ↪ dep_constraints])

    # Add hard facts -- the versions must be satisfied
    facts = self.__get_facts_for_fixed_versions(symbols_dict,
        ↪ comps_with_ver_to_lock)

    # Limit to what versions are there
    facts = And(facts, self.__disjoint(comps, symbols_dict))
    self.__assert_problem_is_sat(And(domain, facts))

    result_vers: [ComponentVersion] = []
    # Walk through each component, get all feasible versions,
        ↪ get max of these versions, constraint problem to it
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
    for comp in comps:
        all_versions_strs = self.__all_smt_one_symbol(And(domain,
            ↪  facts), symbols_dict.get(comp.id()))
        # find max
        max_version_numerical = max(ComponentVersion.to_numerical
            ↪ (ver_str) for ver_str in all_versions_strs)
        max_version_str = ComponentVersion.to_str(
            ↪ max_version_numerical)
        comp_ver = Resolver.find_comp_ver_in_comp(comp,
            ↪ max_version_str)
        # Add hard fact -- the max version has to be satisfied
        facts = And(facts,
                    Equals(symbols_dict.get(comp.id()), Int(
                        ↪ max_version_numerical)))
        result_vers.append(comp_ver)
    return result_vers
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 6

Software Architecture for Air-Gapped Updates

Before we initiate our discussion about the solution architecture, let us cover one of the major workflows related to **software provisioning**. It should remind the reader what functionality is expected. To this point, the workflow is implementation-agnostic, but it strongly relies on the previously defined class and activity diagrams 4.11.2, 4.11.4. The workflow is broken down to the preparation and deployment phases. The preparation phase consists of:

1. The user selects a Software System.
2. The user selects at least one Component and a Component Version of it to be installed.
3. The user selects strategy on how to calculate dependencies. By default, the newest version per Component is returned (MAX-VER).
4. The system resolves dependencies and returns a list of Component Versions to install. Due to possible further dependencies, the system can add more Component Versions of new Components which have to be installed too.
5. The user selects a location and gets all available Execution Environments.
6. The user assigns each Component Version through an Executable Artefact to an Execution Environment which can run it.
7. Deployment Plan is scheduled. User gets information which Execution Environments must be visited.

The deployment phases consists of:

1. The user connects to an Execution Environment from the list.
2. The user changes settings of the new installation if needed and initiates deployment. The user iterates this process until all components are updated.

6.1 Security aspects

Every software strives to achieve high-security standards, however, the unique constraints applicable to this software solution make it to absolute necessity. Authentication and secure communication are the most prevalent security-related points of interest which require further elaboration.

- Authentication

  The login to the software itself follows the current best practices commonly applied for web-based technologies. User authentication is based on a secure password. The communication with the frontend and the backend, embraces usage of TLS/HTTPS protocols, which implies certificate-based authentication, making passing the user's password secure.

- Communication with Execution Environments

  The software is expected to securely communicate with different kinds of Execution Environments such as Docker Engine, UNIX/Windows/Android OS, etc. Exposing these systems to outside connections may introduce a loophole for data breach despite an air gap in place, so it must be approached with special attention.

  Secure Shell Protocol (SSH) is one of the most popular choices when a secure connection to a system is needed; however, its level of security strongly depends on the type of authentication. Password-based authentication has proven to be the most vulnerable one, as the system can be exposed to dictionary attacks; moreover, humans tend to use easy-to-remember insecure passwords [Elf06, SSS09]. Thus, a password-less authentication using public/private key pair is a better choice that we embrace as well. A passphrase can be used to protect the private key from unauthorized use. Docker daemon offers by default only connections through a local unix socket, so no external access is allowed. By enabling TCP connections, the daemon listens to a port that can be exposed to outside clients; however, setting up such access securely requires maintaining additional certificates for the daemons and their clients. To reduce complexity, existing password-less SSH connections are utilized. Communication with Docker daemon is done via port-forwarding, so there is no need to expose it externally and maintain additional certificates.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.2 Artefacts Storage

The system must cache executable artefacts locally and therefore suitable artefact storage should be selected. Since Docker is our main target platform, the main focus is on it. Two options for storing Docker images are presented; either of those can be chosen and the choice does not affect our software solution.

6.2.1 Docker Basics

Docker images are typically pushed to and pulled from a so-called registry. A registry is a storage and content delivery system, holding named Docker images, available in different tagged versions [doca].

In such a registry, Docker images are stored following certain conventions. The image's name consists of "distribution/registry" parts and can be supplied with tags that typically represent the versions. Usually, a publicly available registry named Docker Hub is used, [1] because of its zero-maintenance, high availability and other advantages. Docker Hub is always used implicitly, for example, "docker pull something" command will be translated to "docker pull docker.io/library/something", pointing to the Docker Hub registry. In contrast, command "docker pull somedomain:port/x/y" will force Docker to connect the registry located at somedomain:port to download the image x/y.

Along with the publicly available registry such as Docker Hub, there is a possibility to deploy and operate a private registry. Docker creators claim that running a private registry is a great addition for a CI/CD pipeline and it is the best way to distribute images inside an isolated network [doca]. The latter applies to our scenario.

6.2.2 Docker Private Registry

Docker private registry can be very conveniently started using the publicly-available "registry" image. It means Docker itself runs the private registry. The following command shows how the registry can be started:

```
$ docker run
  -d -p 5000:5000 --restart=always --name registry
  registry:2
```

The command would download the image "registry" with tag "2" from Docker Hub if the image is not available locally. Then it starts a container named "registry" in detached mode, the container's port 5000 is going to be mapped to port 5000 of the running host, and the container will always be restarted if it fails for some reason.

If the registry is exposed to Docker daemons from outside, it should be secured using TLS protocol, and thus a certificate is needed.

[1] More information: https://hub.docker.com/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Image mirroring

Images available publicly can be easily mirrored to a private registry. Let us walk through an example based on a well-known "alpine" image. Our example is inspired by one of the official tutorials [doca].

Download "alpine:3.12" image from Docker Hub to the local Docker storage if it has not been cached before.

```
$ docker pull alpine:3.12
```

The "alpine:3.12" image already available locally gets an additional tag "localhost:5000/alpine:3.12". Notice the prepended hostname and port, which indicates the location of an alternative to the Docker Hub registry.

```
$ docker tag alpine:3.12 localhost:5000/alpine:3.12
```

The image then gets pushed to the registry running on the same IP as the host, using port 5000.

```
$ docker push localhost:5000/alpine:3.12
```

This command returns all locally available images. We present a compact output showing only such information as repository, tag and image id. Now the same image was pulled or tried to be pulled from two different repositories: alpine, which refers to the public one on Docker Hub, and localhost:5000/alpine, which clearly signifies the registry's origin at localhost, port 5000. Nevertheless, both images are identified as the the same with id d6e46aa2470d.

```
$ docker images
REPOSITORY TAG IMAGE ID
localhost:5000/alpine 3.12.1 d6e46aa2470d
alpine 3.12.1 d6e46aa2470d
```

6.2.3 Sonatype's Nexus repository

The Nexus Repository OSS is arguably one of the most popular open-source solutions on the market for managing artefacts offered by the privately owned company Sonatype. [2] This software serves as a single stop when storing all popular binary and build files, covering such platforms as Maven/Java, npm, NuGet, Helm, Docker, P2, OBR, APT, GO, R, Conan and many more. [3] Nexus Repository can run one or even multiple Docker private registries, therefore it does not compete with it.

[2]For more information refer to https://www.sonatype.com/company
[3]For more information refer to https://www.sonatype.com/nexus/repository-oss

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.3 Docker in Production

Docker containers are well suited to be run in production environments; however, doing so without an appropriate tool may deliver unsatisfactory results. For example, a container may fail and get stopped – most likely, the desired behaviour is to restart it after a failure. Similarly, an application may need to be scaled up to handle more workload than originally planned. To provide this kind of functionality, an orchestration tool should be used. Currently, there are multiple solutions available, and arguably the most popular ones are Kubernetes developed by Google and Docker's own so-called Swarm mode. [4]

6.3.1 Docker Swarm

In this section we discuss what Docker Swarm is and how we can take advantage of it based on our needs.

Docker Swarm is a container orchestration solution shipped together with the Docker Engine to facilitate Docker-based deployments in production. Docker creators call a cluster of Docker Engines a swarm. A swarm can be managed by using the same Docker CLI as for managing containers. Docker Swarm is a set of nodes, and each node can be a manager, a worker or both. A node is an instance of the Docker engine participating in the swarm [docb] which runs services and performs certain tasks.

Swarm Service

A swarm service is the definition of the tasks that should be executed on the nodes. Typically, there is one service per image, for example, a RESTful microservice would be deployed using one image as a service. On this level, the following can be defined: the port that the swarm makes available to the outside world, an overlay network to connect to other services, CPU and memory limits and reservations, a rolling update policy and the number of replicas of the image to run in the swarm [docc]. For example, the mentioned RESTful microservice can be scaled by assigning the image to a service and defining the desired state, such as how many replicas are needed. The service will take care of load-balancing between the replicas by automatically splitting tasks among the running containers.

On top of this, Docker ensures that tasks are isolated, meaning one task runs only on one container as defined by the scheduler. The authors describe a task as a "one-directional mechanism", and it progresses through a series of states: assigned, prepared, running, etc. If the task fails, the orchestrator removes the task and then creates a new one as a replacement. In the end, the orchestrator should reach the desired state specified by the service [docc].

A service may run in a so-called "pending" state, if there is no node that can run the service's tasks. This can happen if no node satisfies the constraints, such as CPU/memory limitations, etc.

[4] For more information refer to https://kubernetes.io/ and https://docs.docker.com/engine/swarm/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Apart from being replicated, a service can be global. A replicated service creates n replicas of the running instances based on the provided image. Global service runs one task on every node. So, if another node joins the swarm, the orchestrator will assign the task defined by the global service to it.

Swarm Features

The following features offered by the swarm mode are noteworthy - as described in the official Docker documentation: [docb]

- Cluster management integrated with Docker Engine

  There is no need for any additional installation. Docker Swarm is immediately available.

- Decentralized design

  Docker Engine handles the differences in node types, workers and managers, on its own. A single disk image can be used to build an entire cluster.

- Declarative service model

  A declarative approach is employed to define services. It means a desired state of the services is defined, such as what image should be used, how many replicas must be started, etc.. It is the responsibility of the Docker Engine to reach and maintain this state.

- Scaling It is possible to scale the services up or down automatically. Swarm manager does it depending on how the desired state is defined/changed.

- Desired state reconciliation

  The swarm manager keeps monitoring the cluster state to reconcile any deviations between the actual and the desired state. For example, if one of the worker nodes becomes unavailable, the replicas that ran on that node to reach the desired state will be replaced by creating new replicas on one or multiple workers.

- Multi-host networking

  Once an overlay network is defined for the services, the swarm manager automatically assigns addresses to the containers.

- Service discovery

  Swarm manager assigns a unique DNS name automatically to each service and load balances running containers. The embedded DNS server can translate the service names to the respected addresses.

- Load balancing

  It is possible to expose the ports for services to an external load balancer.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

- Secure by default

  By default, each node authenticates itself using TLS, and the communication between nodes is encrypted.

- Rolling updates

  Updates can be applied to nodes incrementally. The swarm manager lets the user control the delay between service deployment. In case an update is not successful, a rollback can be done.

Swarm Initialization

The following command creates and initializes a new swarm. The –advertise-addr flag tells the manager node to publish its address as <MANAGER-IP>.

```
$ docker swarm init --advertise-addr <MANAGER-IP>
```

A swarm can also be created on just a single node. It means the node is going to be both manager and worker. To do this, the same command must be executed; however the –advertise-addr flag is omitted.

```
$ docker swarm init
```

6.3.2 Docker Configuration Considerations

In the case of a Docker-based deployment, the Execution Plan guides the users on what Docker hosts it needs to visit. If the Docker artefacts belong to a Group, all images are grouped as well, forming a **swarm stack**. The stack's settings can be described with one or multiple docker-compose files. [5] Alternatively, all settings can be applied directly without any files using the Docker Daemon REST API. Thus, two configuration approaches can be embraced: manage configuration information either on our own (custom approach), or rely on the docker-compose files. To better understand which option is more suitable, we list below the major requirements which must be met.

- A fresh installation of the components should be working out-of-the-box. This means there must always be a configuration state that would satisfy the chosen versions' current combination. For instance, for service A starting from version v_n an additional environmental variable must be defined. It means the configuration state related to this version should be aware of this environmental variable.

- The user still should be able to override any settings as it may be required.

The Table 6.1 explains the advantages and disadvantages for both of them.

[5] For more information, refer to https://docs.docker.com/compose/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Configuration type	Advantages	Disadvantages
Custom	Full flexibility, can be easily defined per Image version.	The new specification should always be explicitly added. Also, it is challenging to override the settings of a different service. For example, a new version of service A exposes a different port, so service B needs to get this information since B consumes A.
Docker-compose files	No need to store similar representation internally. In case a new option is needed, it is immediately available.	It is difficult to validate a docker-compose file created by the user to override some settings.

Table 6.1: Docker Services configuration types. Advantages and disadvantages.

Based on the requirements and the relative merits of both approaches, it has been decided to use docker-compose files. The main reason is that all settings implemented in future releases will be automatically available. Also, it is possible to combine multiple docker-compose files. This means each Docker Image will maintain its own configuration files. In the end, all of them are going to be automatically merged. Thus it satisfies the requirements discussed previously.

As a result, the administrator should always provide a base docker-compose file per service (Docker Image) that defines basic configuration like the name of the service, what network(s) it should be connected to, mapped volume(s) or port(s), and so on. A user should never change these settings. For a later Docker Image version, in case the settings change, the administrator defines a different docker-compose file that can override the base settings. Meaning, for example, if an environmental variable must change its value for a certain Docker Image version, it can be defined in another docker-compose file. When loading all the files, it should be loaded after the rest of the files to override the same settings.

Following our discussion, we present an example that shows the deployment of a group of only two components – backend and frontend services. As stated earlier, base configuration files must always be defined. The user may choose to override any settings but it must be done in a separate additional file. The Listing C.1 shows the base configuration file of the backend service. It uses an environmental variable to inject values for such attributes as image and labels dynamically. It is needed because the version depends on what has been resolved. As a result, it affects the referenced registry, repository, image and tag. A drawback of such an approach is that the environmental variables must follow certain naming convention that the user/administrator must adopt. For instance, the image value's environmental variable must be named as the service itself plus "_image" postfix.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

The "at.ac.tuwien.dsg.spunie.image.id" label defined in the example is used to assign internal database id to the deployed image. It is required to understand which exact image version is deployed on a node.

Other properties are set according to the docker-compose tool conventions. For example, a custom network named "www" is defined, and the service is connected to it. The internal port 8080 of the service is mapped to external port 8080 of the host. The deploy section describes different deployment-related properties, such as how many replicas should be created, what action should be applied on a failure, what is the restart policy, etc.

The Listing C.2 shows the base frontend configuration file. It is very much similar to the backend configuration file. Again, the environmental variables must be defined following the same naming conventions.

The Listing C.3 shows how the user can override or add new properties. It may be needed because a newer version requires a different configuration or there may be another reason to do so. The example configuration file adds another port mapping – 80 internal port to 80 external one. Unfortunately, it is not possible to remove the old port mapping, however, it should not be necessary in the first place. The base configuration files should only contain the essential configuration. All the files defined later are not going to be applied each time but only those currently needed. It means that if the port mapping is not defined in the base file, it will be completely overridden with another user-defined file. The other deployment-related property overrides the previously defined value directly. Once this configuration is applied, Docker Swarm will downscale the service from two replicas to just one.

New deployment

Let us describe which steps the user needs to perform for a new installation.

1. User connects to a node according to the Scheduled Plan.
2. Docker-compose files with standard settings are automatically applied. The user can optionally override any of the settings or defined new by creating additional docker-compose files.
3. The user initiates deployment of the artefacts.

Update deployment

To update an existing deployment, essentially the same information is needed as in case of a new deployment. The orchestrator, whatever implementation is supplied, will take care of the update procedure.

1. The user connects to a node according to the Scheduled Plan.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2. Docker-compose files are used, as in case of a new deployment, then appropriate image reference(s) is/are going to be injected.

3. User initiates deployment of the artefacts.

6.4 Plugin support

As described previously, integrating an orchestration tool for Docker containers would be valuable during the deployment and operations phases. However, there is a risk of vendor lock-in, if such a tool is directly integrated into the application. Also, the situation on the market can change and another tool may gain popularity or the currently popular ones may lose popularity. Therefore, supporting the integration of orchestration tools with the help of plugin architecture is the way to proceed. Additionally, it facilitates clear separation of concerns by forcing the usage of clearly defined interfaces.

To support the desired plugin-based architecture in the software solution, a framework is used. It has been decided in favour of PF4J first of all due to its simplicity. [6] Being lightweight (around 100 KB) and easy to use, it still offers great flexibility and support of multiple further frameworks like Spring, meaning both plugin container and plugin itself can be Spring-based projects. The core concept of this framework is to provide an API which is essentially an interface that inherits another interface org.pf4j.ExtentionPoint provided by the framework. Then a plugin implements this API and annotates the implementation with org.pf4j.Extension annotation. On top of this, to make use of Spring Dependency Injection, the plugin class loader can be set to the application context making the plugin implementations available via Dependency Injection. To package a plugin assembly, metadata must be set such as which class should be loaded, plugin version, provider, id, etc. After packaging the plugin, either a zip file or the whole folder containing compiled classes must be placed into a directory according to the settings of the plugin-container. Using org.pf4j.SpringPluginManager, the plugin-container loads all available extensions for the API. At this point, it may be enforced to use only exactly one implementation or still allow to make use of multiple ones. The Figure 6.1 shows the described plugin architecture based on a very simplified example using a UML class diagram. Please note the diagram does not fully depict the real implementation, but omits some information to preserve simplicity.

6.5 Final architecture description

Our software solution embraces the microservice-based architecture, as the intention is to create flexible and future-oriented software which should be easily modifiable and scalable. Another reason to do so is to mix multiple programming languages in the backend, such as Java and Python. As shown on the Figure 6.2, the software solution comprises these main blocks: Gateway, Services, Logs Monitoring, Service Registry, and others.

[6] Visit the official homepage for more information: https://pf4j.org/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

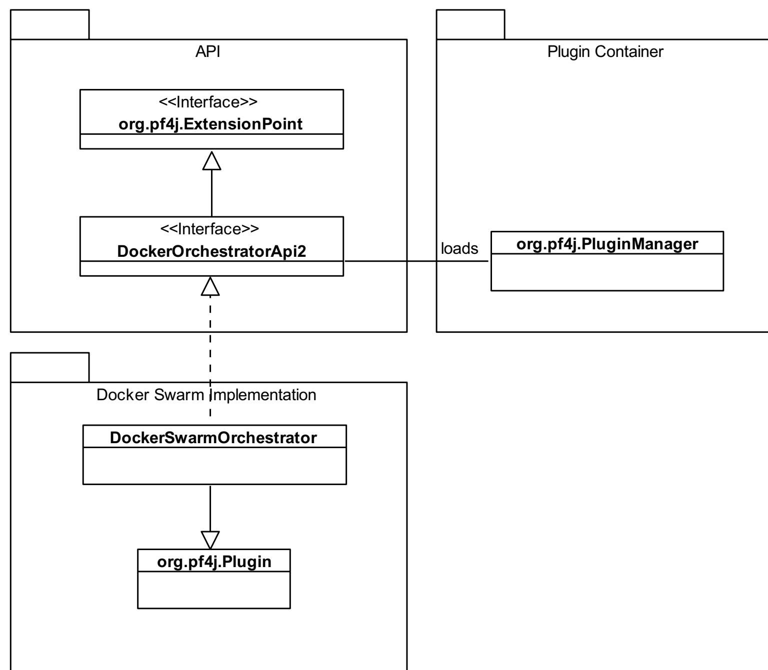

Figure 6.1: UML Class diagram - plugin-based approach

The Gateway and Core services were initially generated using JHipster framework, [7] but later significant changes were made. For example, since some domain entities are shared between multiple modules, these were externalized to a separate module using the Maven building and packaging tool.

The **Gateway** serves as the single entry point to the application. It provides a Web UI built with Angular framework. [8] The Gateway uses the Service Registry to redirect user requests to services, as needed. The Gateway itself holds neither any business logic nor domain entities, except for user access management. Due to this reason and to be able to route a user's request, the Gateway does have a server-side backend implemented using Spring Boot.

The **Core** service contains most of the business logic, such as CRUD operations for most of the entities, logic related to composing a Deployment Plan, etc. It is implemented using Spring Boot, and it does not offer directly any UI.

The **Dependency Resolver** service, as the name implies, is responsible for dependency resolution. The solution's core functionality lies within this module, which leverages the PySMT library. We provided in Section 5.3 a thorough explanation of why precisely this library has been chosen. To recapitulate, it is a versatile Python library that makes it possible to interchangeably use various SMT solvers, since it operates on a higher level of

[7]More information on https://www.jhipster.tech/
[8]More information on https://angular.io/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

abstraction. Python is an appropriate choice for this kind of programming tasks, as it is dynamically typed. The stand-alone module exposes a RESTful interface using OpenAPI specification (more details to follow).

The rightmost part of Fig. 6.2 illustrates functional components outside the core architecture, namely interaction with different execution environments the system may employ – Docker is assumed to be the main service containerization technology, but mobile application containers or images may be also included. In practice, Docker Swarm implements an interface to ensure loose coupling and avoid vendor lock-in. The depicted artefact repositories provide, for example, container images for offline usage. Finally, logging and monitoring facilities address traceability and auditability. Concrete technological choices for implementation of the functional components are depicted in Fig. 6.2 in gray; those represent contemporary technologies that can be adopted.

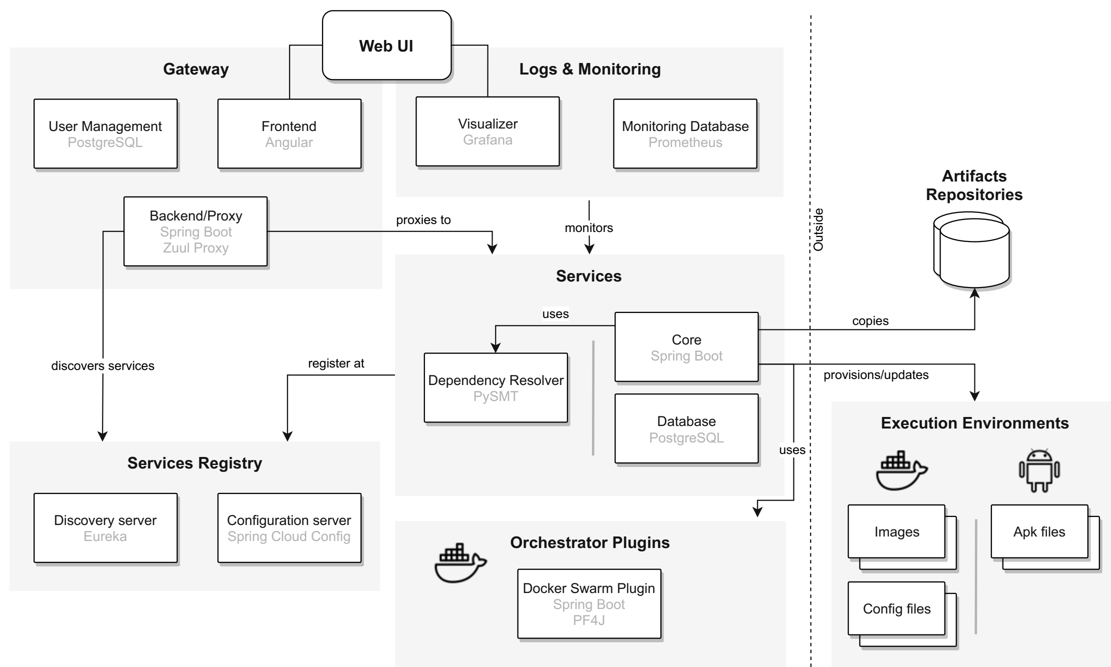

Figure 6.2: High-level solution's architecture diagram

We appreciate that a microservice-based architecture may not be the best choice for software meant to be run on a single workstation, as described in Section 4.2. Nevertheless, it provides a great degree of flexibility to the users. For example, the SMT module can be easily scaled up, and replicas can be deployed to external servers. Since the dependency resolution process occurs when the system is connected to external networks or the internet anyway, it can significantly reduce the processing time by utilizing the additional computational resources.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6.5.1 Communication between services

Currently, there are different options to establish communication between services. These are, but not limited to, REST, RPC, inter-process communication or utilization of queuing mechanisms. A RESTful API seems the most suitable choice due to such criteria as high adoption, scalability and language/technology decoupling. Additionally, it has been decided to use OpenAPI Specification, [9] as it allows to define RESTful APIs in a standard language-agnostic way. It also facilitates the application of best practice as well as clear documentation. Last but not least, client and server parts can be automatically generated for dozens of programming languages and frameworks.

6.5.2 Proof of concept

The developed software solution is a proof of concept. It does not provide all pieces of the functionality described in this work. Nevertheless, much of the basic functionality is in place. Most importantly, the domain model discussed in Section 4.11 was fully implemented as well as the core module for the dependency resolution. Additionally, we partially implemented the graphical user interface, which is presented in the next section.

6.6 Graphical User Interface

The graphical user interface is implemented as a single-page web application using the Angular framework. The biggest advantage of a web interface as opposed to a traditional desktop one is that it can be used on essentially any operating system and reached via a network. Further, we show screenshots and briefly describe them.

SPUNIE is the working title and stands for Software Provisioner and Updater in Network-Isolated Environments. While the user is not yet logged in, he/she can only change the language and manage his/her own account. After the login is performed, more functionality becomes available, as depicted on Figure B.1. As a result, the toolbar provides access to such additional drop-down menus as "entities" and "administration".

The "entities" menu, shown on Figure B.2, allows the user to manage domain entities. In most cases, only basic CRUD operations are provided. Figure B.3 shows an example of a standard create/edit view. The example illustrates the creation of a new Unix Execution Environment. A standard page listing all objects of an entity is shown on Figure B.4. Through the "entities" menu also more sophisticated views are available, which we will cover later.

Figure B.5 shows the "administration" menu. It allows the user to access administration related information, such as what services are available, managed users, view application metrics, etc. Figures B.6 and B.7 two of the pages accessible via the "administration" menu. The first one allows the user to check the application's health status, and the second one to monitor currently available services.

[9] More information here: https://swagger.io/specification/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Next, a drop-down menu for language selection is in place. The application offers English and German, but other languages can be easily added. Additional translation of all fields must be then provided manually.

The last toolbar menu to describe is named "account". The user can use it to change own account's settings, change the password or sing out.

Further, let us walk the reader through the process of software provisioning consisting of four steps. The first step is depicted in Figure B.8 where the user selects one of the predefined Software Systems. In the next step, the user is required to pick components that are to be installed. Figure B.9 shows step 2. On the Figure B.10 step 3 is presented. In this step the user chooses component versions and checks if the combination is valid by sending a request to the dependency resolution module behind the scene. If it is, the user gets notified with a success message, as shown on Figure B.11. In case the combination is invalid, the user sees an error message. Such a situation is shown on the Figure B.12. Finally, the user chooses the location where the software should be installed and assigns components to Execution Environments, as shown on Figure B.13.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Evaluation

We open this chapter by presenting a typical scenario and apply our solution to it. Then the original requirements are revisited. Next, we discuss some performance metrics of the dependency resolution module. The chapter is concluded with a lessons learned section.

7.1 Applicability

In this section a typical scenario is presented. It describes a situation where our solution can be applied and serves to verify the solution's applicability, as well as answers the RQ1 from Section1.3. The procedure is comprised of three key steps:

1. Development

   Based on interviews and observations, a typical scenario is identified and developed.

2. Application

   The solution is applied to the scenario.

3. Evaluation

   Results are evaluated based on the applicability of the solution. The advantages and shortcomings are discussed.

7.1.1 Scenario Development

During the development of the typical scenario, besides conducting interviews with selected key stakeholders, the diverging operating modes of the organization's On-Site Inspection division are to be taken into account. One of these modes is usual day-to-day work at the organization's headquarters when air-gapped software products are used/tested only occasionally. A second mode relates to training activities, including

the extensive preparation, when the software products are heavily used; however, minor disruption of service is acceptable. These training events may take place either at the headquarters or in a remote, distant location. For example, from 3 November to 9 December 2014, an Integrated Field Exercise took place in Jordan to simulate an almost entire on-site inspection. Last but not least, is real On-Site Inspection when the software is expected to be as robust as possible. Thus, the typical scenario should ideally cover all operating modes.

For further development of the scenario, four key users/stakeholders have been interviewed by asking these questions:

- What is the most common task you must perform?
- What is the most challenging task you struggle with?

Most of the interviewees stated without any hesitation that updating one or multiple systems is the most frequent task to be performed, because the majority of the software products are in constant development. According to them, this task is very challenging due to the air-gapped nature of the systems. Provisioning of new systems is of high interest as well, however it does not happen very often. Most of the users/stakeholders expressed their desire to improve both processes and make them well organized. Based on the previously defined use cases in Section 4.11.1, updating software is one of them, thus it is a good candidate for the scenario to be derived from.

7.1.2 Scenario description

By engaging in further conversation with the stakeholders, the scenario could be shaped and outlined. It fully embraces the example system used throughout the paper from Section 4.5. The scenario is purely hypothetical; however, it has been elicited using a real-world setting and is highly representative.

Imagine that an OSI has been requested, at which point preparations are made during the Launch Phase to deploy approved equipment, including all computing equipment, from headquarters to the inspection area. During this period all software systems are carefully prepared, configured and provisioned. All systems are then dispatched to the inspection area. The software setup is comprised of particular versions, as shown on the Figure 7.1.

In accordance with paragraph 59 of the Protocol to the CTBT [ctb96], during the second week of the inspection, the inspection team requests the inspected State Party (ISP) to provide a piece of equipment to conduct radionuclide measurements in the mobile laboratory following the discovery of a fault in one of the deployed detectors. The ISP detector and accompanying software meet the specifications set out in the list of approved equipment but the software is not identical and is not directly compatible with the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

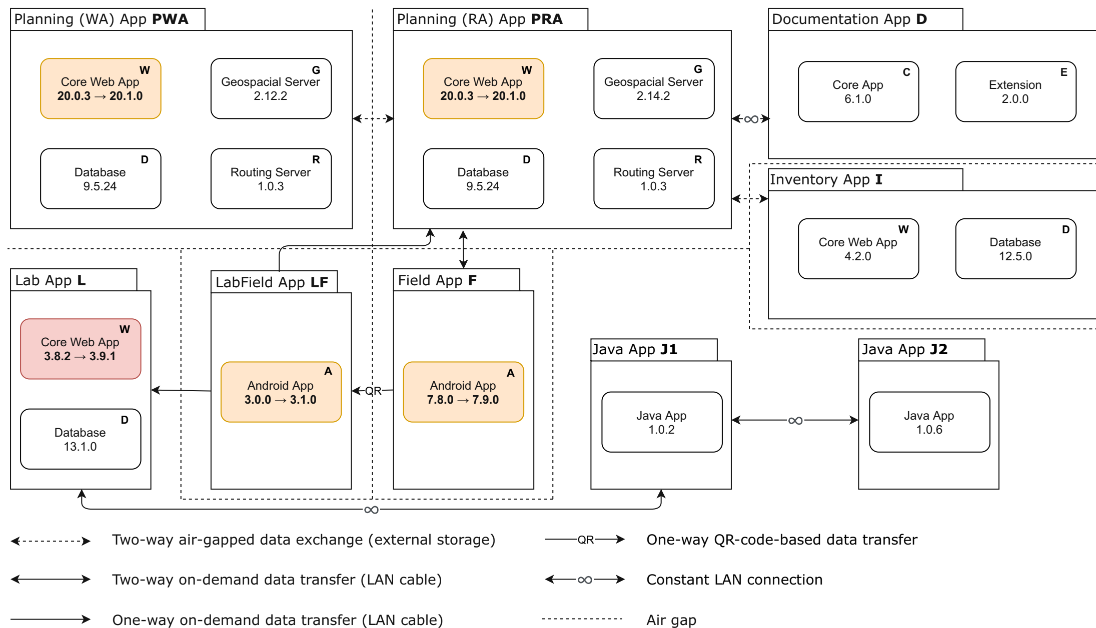

Figure 7.1: Scenario: software setup highlighting components to update

Laboratory Application. The inspection team DFO assesses the differences between the software and delegates the work to integrate the newly provided detector.

After the needed software update is delivered, it turns out it affects multiple applications along with the directly related one. The Figure 7.1 highlights in red the application which uses the equipment directly, and it highlights in orange four other components which must now incorporate changes as well. The diagram also shows how communication between the components is implemented. Another Figure 7.2 shows an airborne photograph of a simulated OSI. The picture denotes the places where the applications/components to be updated are used. None of the software products is connected to the internet nor to each other. Most importantly, there is a strict air gap policy between the working area (WA) and receiving area (RA) applications indicated by the red dotted line.

The scenario is now described with series of actions without yet taking advantage of our solution.

1. Multiple components receive an update

   Due to the previously described change request related to the Laboratory Application – its component named W should be updated to version 3.9.1. Because of this, new versions of other applications have been released too: Field Application (A component), LabField Application (A component), Planning (WA) App PWA (W component), Lab App (W component). Thus an update of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek, Your knowledge hub

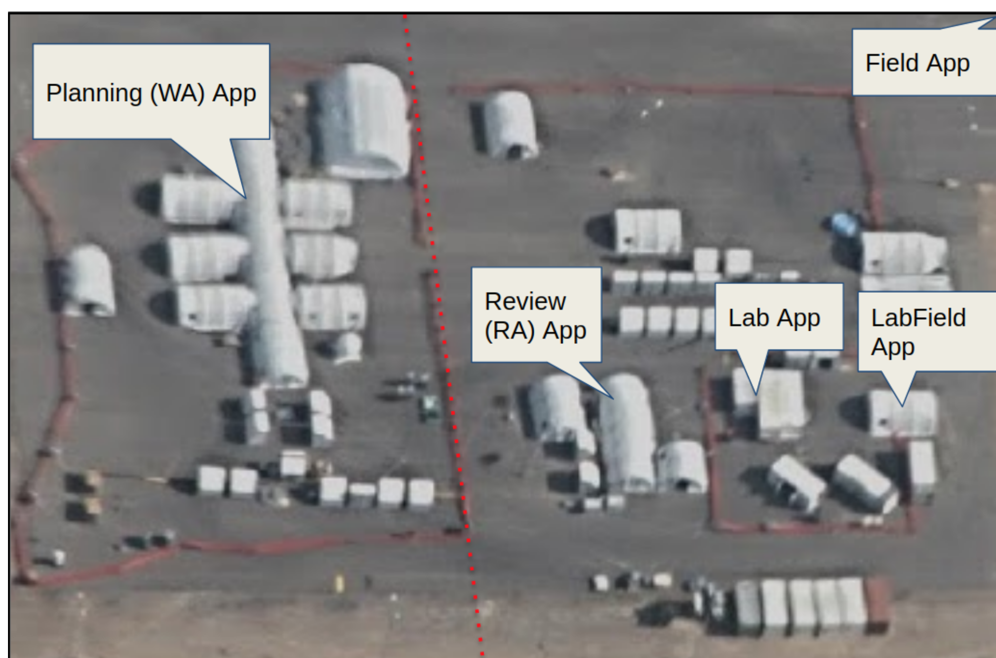

Figure 7.2: Scenario: airborne photograph of a simulated OSI from a training event showing deployed applications

Lab Application would require updating four other applications. Figure 7.1 illustrates the applications comprised of components, with versions (before and after) in bold.

2. Prepare all needed artefacts

   Within this step, all artefacts of needed versions must be prepared for offline deployment. The PRA, PWA and L applications are containerized, which means exporting the right Docker images to files. For the Android application LF and F it would mean the preparation of apk files.

3. Deploy the artefacts at each node

   Each of the nodes, either server or table PC, must be physically visited. The files are then transferred using a secure connection and deployed. The Docker images are imported and the applications are restarted. Potentially the configuration must be adapted. The apk files are installed on the tablet PCs.

Fulfilment of such a scenario normally requires performing multiple manual steps which is error-prone. Individual artefacts of the right versions must be collected beforehand. Different target execution environments require different artefacts: a container image for Docker-enabled environment and an apk file for the Android operating system.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

There is no overview of what components of what versions are running and how potentially a dependency may be left unsatisfied. These are only a few limitations to the manual non-systematic approach.

7.1.3 Applying proposed solution to the scenario

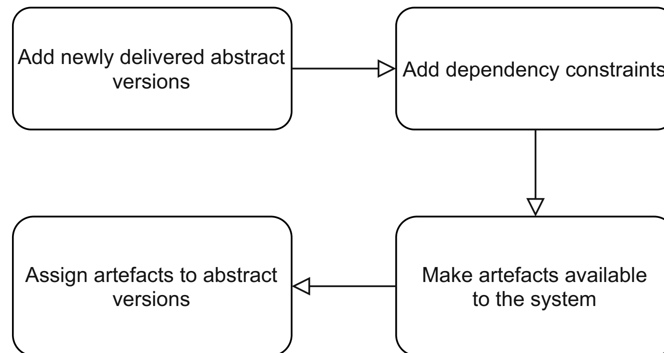

Figure 7.3: Scenario: add new versions, dependency constraints and artefacts (simplified)

This section covers the process of applying the software solution (further referred to as just "system") to the above-described scenario. Essentially it consists of the following steps:

1. Add new abstract versions, dependency constraints and artefacts (precondition).
2. Schedule Deployment Plan to perform needed updates.
3. Perform updates by visiting each node and close the transaction.

Further, we discuss each of the steps in greater details. The Figures 7.3, 7.4 and 7.5 show simplified versions of the procedures. Each Figure describes one following subsection in the same order of appearance.

Add new versions, dependency constraints and artefacts

The precondition for using the system is that the new version(s) and dependency constraint(s) must be defined beforehand by the experts who delivered the needed update(s). Usually, these experts are software engineers. At first, the newly delivered abstract versions are added, [1] as shown on the Figure 7.1:

[1] We say "abstract" versions to emphasize they are not associated with concrete executable artefacts yet.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1. Lab App W "3.9.1"
2. LabField App A "3.1.0"
3. Field App A "7.9.0"
4. Planning (RA) App W "20.1.0"
5. Planning (WA) App PWA "20.1.0"

Afterwards, the following dependency constraints are added:

1. Lab App W "3.9.*" [2] requires LabField App A "3.1.0"
2. Lab App W "3.9.*" requires Field App A "7.9.*"
3. Lab App W "3.9.*" requires Planning (RA) App W "20.1.*"
4. Lab App W "3.9.*" requires Planning (WA) App PWA "20.1.*"

The software engineers decided to restrict the dependee version to a fixed one only of LabField App A; in the rest of the cases, only the major and minor version parts are fixed, the patch one is open.

Another precondition that should be set potentially by the software engineers as well as to provide the new artefacts and link them with the new versions. According to the scenario, Docker and Android platforms are the target execution environments. This is why the following artefacts are added:

1. Docker image "docker-registry.local/lab/w:3.9.1" to Lab App W "3.9.1"
2. Android apk file named "lab_field_3.1.0.apk" to LabField App A "3.1.0"
3. Android apk file named "field_7.9.0.apk" Field App A "7.9.0"
4. Docker image "docker-registry.local/pra/w:20.1.0" to Planning (RA) App W "20.1.0"
5. Docker image "docker-registry.local/pwa/w:20.1.0" to Planning (WA) App PWA "20.1.0"

The system makes the above-listed artefacts offline-available by automatically downloading all Docker images from docker.ctbto.org, an external Docker registry, to a local one, and by copying the apk files from an SFTP server fs01.ctbto.org to Nexus Repository OSS. [3] To do so, exceptional permission is granted by the ISP. As a result, a VPN

[2] Meaning the Lab App software component W of version 3.9.* with variable last part named patch
[3] The locally used storage solutions are described in more details in 6.2

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

connection to the Vienna headquarters is established with a highly restrictive firewall in place. The completeness of shipped products outlined in 4.7 covers in this case, only the bare minimum (c_{min}) plus other artefacts that were made previously offline-available as well (c_{old}). All together the completeness equals $c_{min} + c_{old}$.

The target execution environments are automatically matched based on the artefacts: Docker Engine for Docker images and Android OS for apk files.

Schedule Deployment Plan

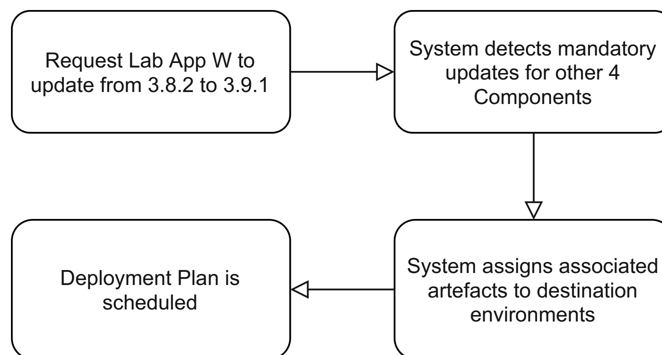

Figure 7.4: Schedule Deployment Plan (simplified)

After the prerequisite information has been added, the DFO initiates the update by scheduling a Deployment Plan:

1. DFO selects the target Software System, which is "OSI Suite" in this case.
2. DFO selects the Component which was meant to be updated in the first place –> Lab App W.
3. DFO sees current version "3.8.2" which is supposedly installed. He/she selects a new version "3.9.1" to update to.
4. System resolves dependencies and returns a list of mandatory updates for other components: LabField App A "3.1.0", Field App A "7.9.0", Planning (RA) App W "20.1.0", Planning (WA) App PWA "20.1.0".
5. DFO confirms the initiation and system schedules a Deployment Plan. In this case the available artefacts match the target platforms, so the system automatically assigns them for installation without involving the user.

Visiting each node

The DFO studies the Deployment Plan to understand which nodes must be visited. He or she follows it by performing the following steps:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

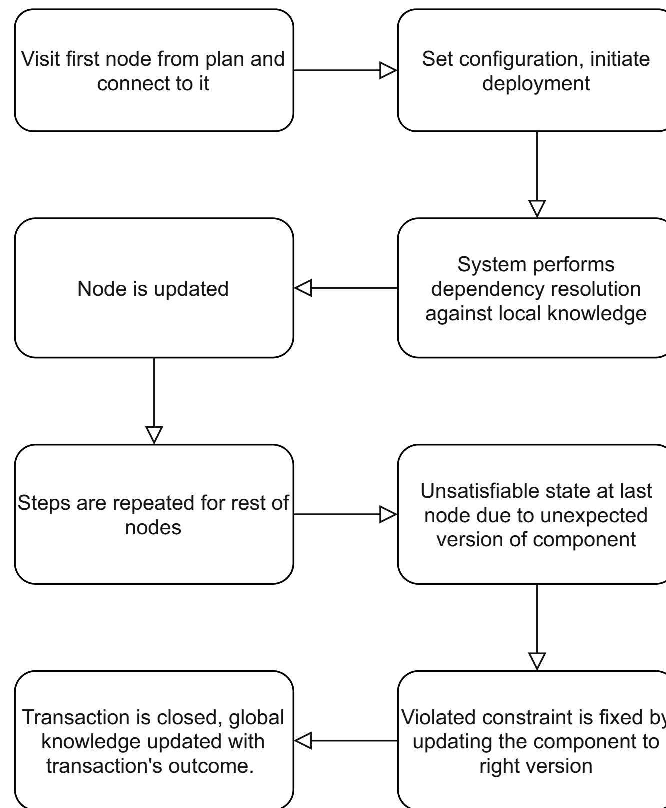

Figure 7.5: Scenario: visiting each node (simplified)

1. At first the DFO visits the node which hosts Lab App W and connects to the server by a network cable.

2. The DFO sets two environmental variables in the configuration yml file to define the credentials of a recently added database user. Then the DFO initiates the deployment. The software solution locates the server by the predefined IP address (172.27.59.93) and can authenticate itself via SSH automatically using a private key.

3. Before initiating the deployment process, the system verifies if the local knowledge matches the global one and performs dependency resolution to make sure the new installation will not break the compatibility of the components and their versions. No inconsistencies were spotted.

4. The DFO repeats the same steps for all the rest of the components from the Deployment Plan: LabField App A, Field App A, Planning (RA) App W and Planning (WA) App PWA.

5. During the update of Planning (RA) App W, local knowledge does not match the global one, leading to an unsatisfiable state. Due to unknown reasons, the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Geospatial server was downgraded from 2.14.2 to 2.12.1, which had not been communicated to the system. This violates the following constraint: Planning (RA) App W "20.*.*" requires Planning (RA) App G of at least "2.13.0". Therefore the Deployment Plan is extended to one more installation of Planning (RA) App G "2.14.2". Luckily enough, the system cached the needed artefact for offline use before, so there is no need to make it offline-available.

6. After the successful installation of all the components, the transaction is closed, and the internal representation of the global knowledge is updated with the recent changes.

7.2 Revisiting original requirements

During the requirements engineering phase, we elicited and identified multiple requirements following the Volere template [RR00]. In this section we revisit these requirements. Firstly, we identify what requirements (in the form of user stories) could be fulfilled and by what means. Secondly, we discuss what could not be achieved and the reason why.

7.2.1 Fulfilled user stories

The most important user stories related to updating software are defined in 4.4.1, 4.4.3 and 4.4.2. All of them share one common goal – to update the software to a different version, but they differ in the location where it takes place. The scenario from 7.1 first of all clearly shows the fulfillment of the user story 4.4.3. Since all three of them are very similar, we can say the other two are fulfilled as well.

Along the way, the scenario demonstrates the fulfilment of a few more user stories. For example, in the scenario, dependency resolution is used, thus the DEFINE-DEPENDENCIES story 4.4.4 can be marked as fulfilled. The same applies to OVERVIEW 4.4.6, because the user can get an overview of installation across locations based on global knowledge, as discussed in 4.6.

7.2.2 Unfulfilled user stories

The user story NO-DATA-COLLECTION defined in 4.4.5 has not been fulfilled. It is about demonstrating to the ISP that the software solution does not acquire data while it performs its work. In other words, the ISP might claim the software solution collects data that should no be collected and the question is how to prove it does not. Even though the user story is of high importance, we argue it is out of the scope of the thesis and should be included into future work.

The user story VERIFY defined in 4.4.7, which covers verification that all needed software components are in place, has not been directly fulfilled, however indirectly, it has. Let us remind the reader that during provisioning or updating each of the steps, the dependency resolution module is verified. So the unreliable global knowledge is constantly being

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

matched against the local one. On the one hand, this does provide the required verification and partially eliminates the need in additional verification in the first place. On the other hand, there is still the possibility that one of the components has been changed, and the system is in an inconsistent state, so a separate verification would be required.

7.3 Performance

The core component of the software solution, responsible for dependency resolution, is the most crucial bottleneck in terms of performance. As it tackles an NP-Complete problem using SMT solvers, we present various performance tests of this component. Further performance evaluation helps us to answer RQ5 from Section 1.3.

In further tests, we, first of all, inspect the dependability between the size of the problem and the execution time, while considering the memory footprint. The tests were performed on a laptop with the following specifications: Intel® Core™ i7-6820HQ CPU @ 2.70GHz (4 cores, 8 threads), 32 GB of DDR3 RAM, 1 TB of SSD storage. The tests were executed in a container using Docker version 20.10.3, build 48d30b5. We also utilized the following software: Ubuntu Desktop 20.04, Python 3.8.5, PySMT 0.9.0 and MathSAT5 solver 5.6.1.

We conducted a short experiment by limiting the number of the CPU cores available for the Docker container to one versus not limiting them at all. As a result, there was a negligible difference in performance results, which means the SMT solving does not benefit from parallelism, just as was expected.

Last but not least, when we refer to dependency resolution, we do not count in external dependencies, such as those managed by different packaging managers and build tools like Maven, npm, etc.

7.3.1 Testing framework

To generate and collect the results in an organized way, a simple testing framework has been developed. Besides the need to automate the process of collecting performance results, we must consider the deviations between the test runs. This requires, even more, to develop an automated testing solution, as we need to perform n iterations to calculate the mean values and the confidence interval with a confidence level of 95%. We argue 30 iterations would be sufficient.

The Listing C.4 shows the main function which controls the execution of tests. It fully embraces the concept of polymorphism – a well-known pillar of object-oriented programming. The Listings C.5 and C.6 portray how abstract classes for data generation and test execution are constructed along with selected implementations. Let us now walk through the main function we just mentioned. After defining the test runners, which essentially represent different dependency resolution strategies (ALL-VER or MAX-VER), the number of cycles or test iterations is defined. The default value is 30 which should be sufficient to get meaningful results. Afterwards, a test data generator with

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

five components and a variable number of versions per components (see array named "iter_array") are defined. Finally, the first batch of tests are conducted using both MAX-VER and ALL-VER runners. We will discuss the internal work of the "conduct_test" function shortly after. The next tests run uses another data generator, which provides 15 components instead of 5. The versions growth stays the same as defined in the array named "iter_array". The last two test runs follow the same principle. They utilize the same data generators which increase the number of components; however, one generates 50 versions per component and another one 125 versions. The number of components is in the range from 2 to 20.

The Listing C.8 depicts the internal work of the previously mentioned "conduct_test" method. At first, an instance of dependency resolver is passed to both test runners A and B, mimicking a dependency injection. The iterations depend on the values defined in "iter_array, " which can increase the number of versions or increase the number of components per iteration. The inner loop then iterates over the number of cycles (30 by default). Inside both loops before running the test with either of the runners, the current timestamp and application's memory allocation size are captured. After the test has been completed, the elapsed time and the memory consumption are stored. [4] The same applies to test runner B. After each iteration of the inner loop, the results are stored in arrays. Finally, after the inner loop finishes its work, the results are processed by calculating the mean values and confidence intervals. The Listing C.9 shows how it is achieved.

Last but not least, the data generation procedure is displayed on Listing C.7. It is partially static. For example, the dependency constraints which describe the relations between components, never change. However, the amount of versions per component is dynamically generated. We omit other data generation approaches on purpose, as the differences are mostly insignificant.

7.3.2 Execution time vs size of the problem

The size of the problem can be quantified using multiple dimensions, such as several components and number of versions per component. These are also affected by the constraints which limit the selection pool for an SMT solver. When measuring performance, we take into account only the two firstly mentioned dimensions. The decision mainly is driven by lowering the complexity of synthesizing the tests.

To summarize, in our tests, the problem size grows by:

(a) fixed number of components and increasing number of versions per component

(b) fixed number of versions per component and increasing number of components

[4] We capture memory peaks before and after a test run, so by subtracting those, we can derive the memory consumption.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

a. Increasing number of versions per component

The Figure 7.6 shows execution time comparison between different strategies, ALL-VER and MAX-VER described in Section 5.1.

The x-axis shows the size of the problem, more specifically, the number of versions per component processed by the algorithm. For example, if there were only two components, A and B, and x equals 60, it means components A and B have 60 versions each, which results in the total size of the problem $60 * 2 = 120$. The size of the problem also strictly depends on the number of components, so we differentiate between a system with 5 components and a system with 15 components. For a system with 5 components, it would sum up to $60 * 5 = 300$ total number of versions, and for a system with 15 components $60 * 15 = 900$.

The y axis shows the execution time in seconds and denotes the algorithm's time to calculate the dependencies.

Last but not least, the vertical line named "real lim" shows the realistic limit of the size of the problem. It means in practice, the number of versions per component is improbable to surpass this limit. The problem could also be downsized to keep the number of versions within the stated limit by selecting only the top n versions beginning from the newest one.

b. Increasing number of components

Similarly to option (a), option (b) inspects the dependability between execution time and size of the problem, while applying different strategies, ALL-VER and MAX-VER. The only difference lies in the way the size of the problem grows. The Figure 7.7 depicts the results.

The x-axis shows the number of components with a fixed number of versions per component. Two options for fixed versions per component are presented: 50 and 125. The number of 50 versions should reflect what to expect in practice. Even though it is unrealistic to deal with 125 versions per components in a real setting, we think it is interesting to know how the solution scales.

Intermediary results interpretation

Based on the provided results as per option (a), using MAX-VER strategy is clearly advantageous when the problem grows significantly. However, with only 5 components, the algorithm can handle a substantial input size in a reasonable amount of time regardless of the applied strategy.

Option (b) proves again that the application of MAX-VER in favour of ALL-VER is only beneficial when the size of the problem is significant. When the number of components reaches 10 and more, the execution time difference approximately doubles. Again, it mostly applies when the problem is of significant size which is an unlikely case.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

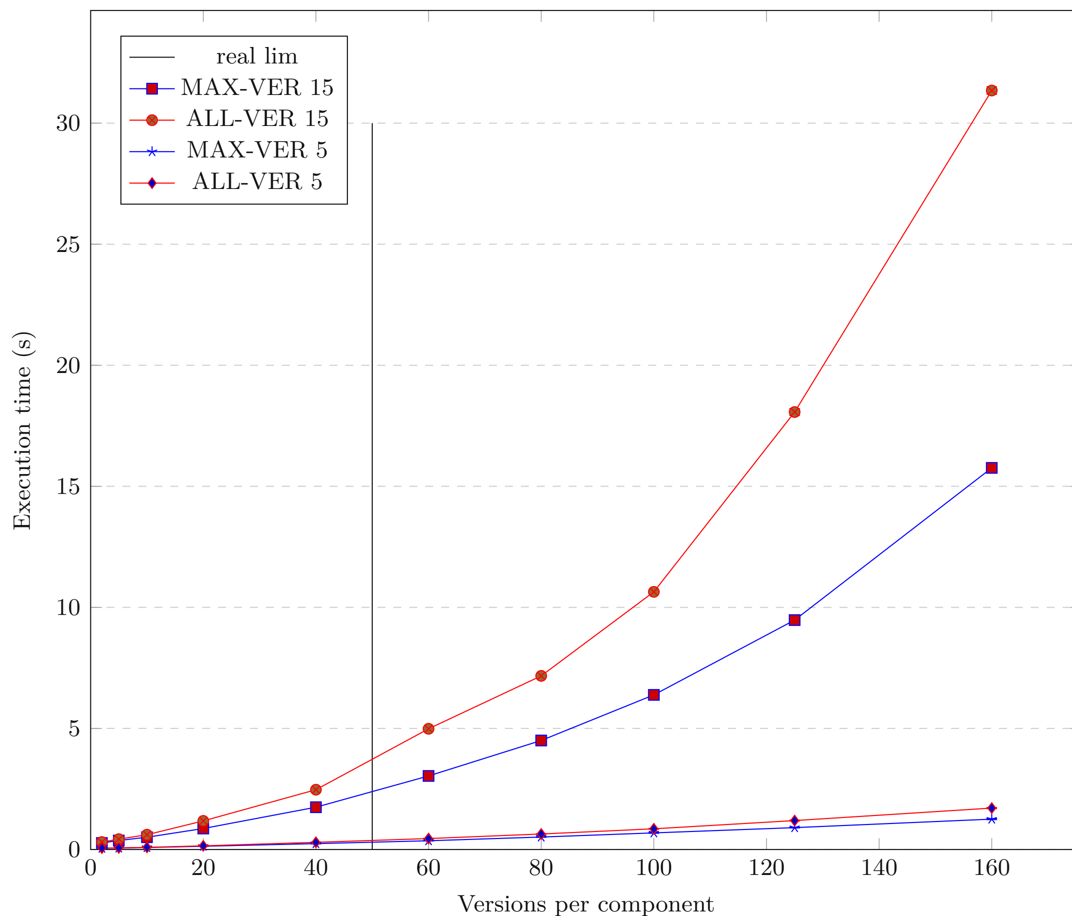

Figure 7.6: Dependency resolution performance (growing number of versions)

7.3.3 Memory consumption vs size of the problem

The two options (a) and (b) from the previous test case apply to this one as well. The only difference is that the y axis shows the memory consumption instead of execution time.

a. Increasing number of versions per component

The Figure 7.8 shows memory footprint comparison between different strategies, ALL-VER and MAX-VER described in Section 5.1.

The growth of the problem on the x-axis follows the same approach as described previously. The y axis shows the memory footprint/consumption in megabytes of the running algorithm. Again, the vertical line name "real lim" shows the realistic limit of the size of the problem.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

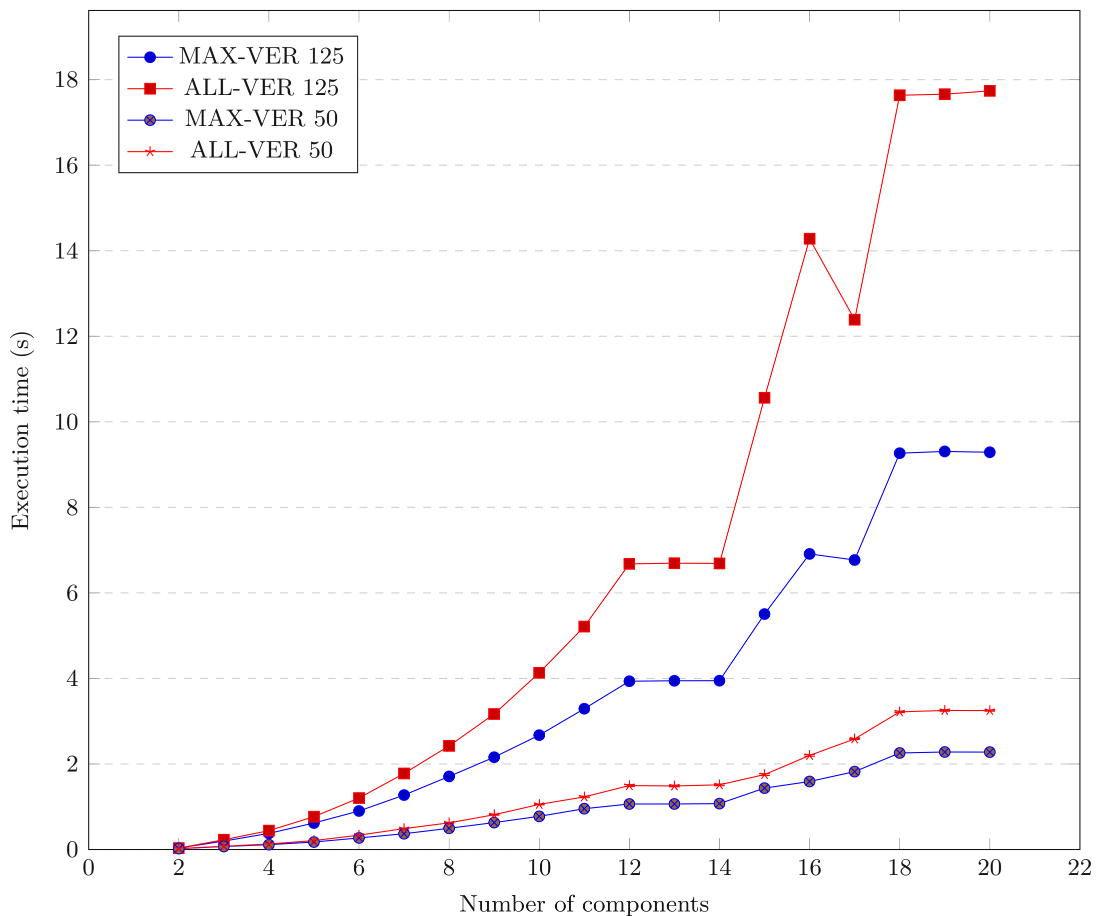

Figure 7.7: Dependency resolution performance (growing number of components)

b. Increasing number of components

The Figure 7.9 illustrates the memory footprint comparison between different strategies. The only difference to the previous figure lies in the x-axis. The rest applies to the description of option (a).

Intermediary interpretation

Based on the provided numbers, we can conclude that memory consumption is not directly dependant on the size of the problem. Moreover, even when the realistic limit is surpassed, the memory footprint stays within 12 MB which most likely can be neglected. Thus the algorithm does not greatly depend on memory.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

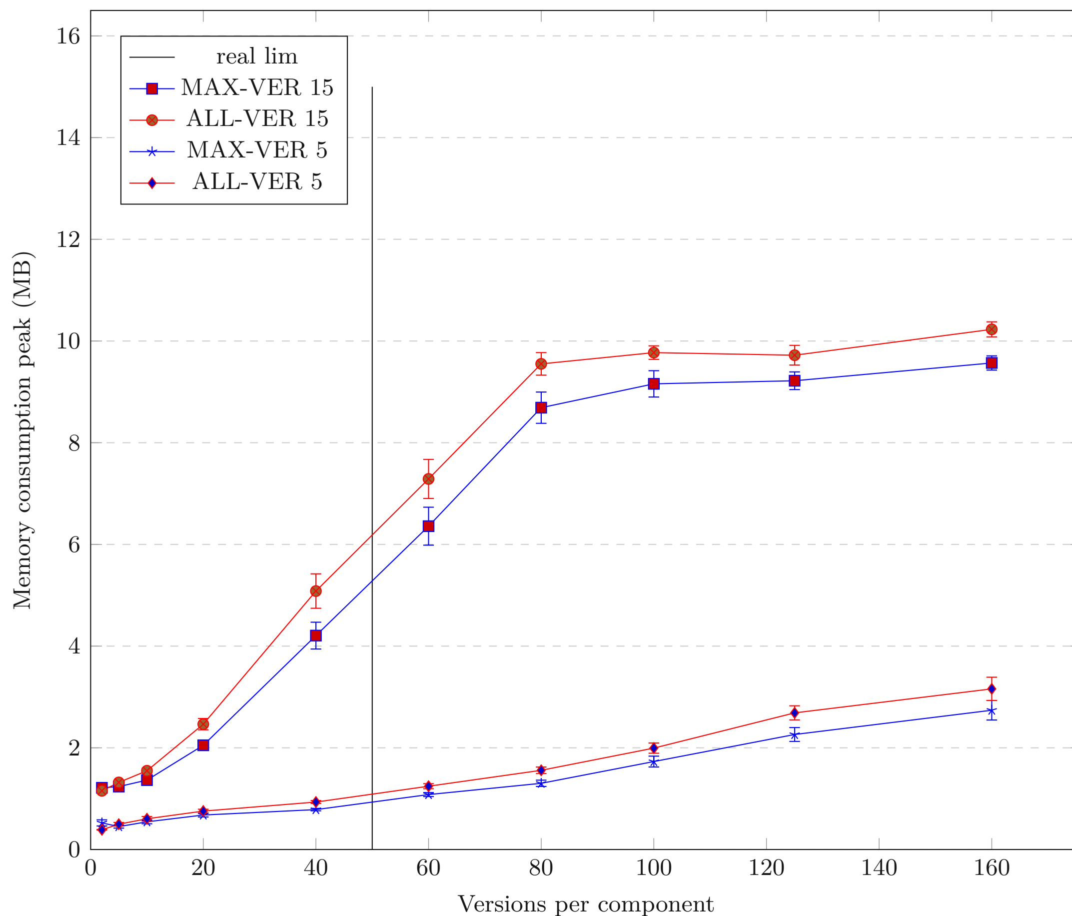

Figure 7.8: Dependency resolution memory footprint (growing number of versions)

7.4 Lessons Learned

In this section we discuss what lessons were learned based on the gained evaluation results.

7.4.1 Scenario applicability

Previously a typical scenario in Section 7.1 was presented, and its applicability of the system was discussed. Numerous **advantages** compared to the manual approach could be identified:

1. State overview

   The system provides an overview of a target software system (set of applications) based on global knowledge. Even though, as discussed earlier, the global knowledge is not completely reliable, which was also reflected while applying the Deployment

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

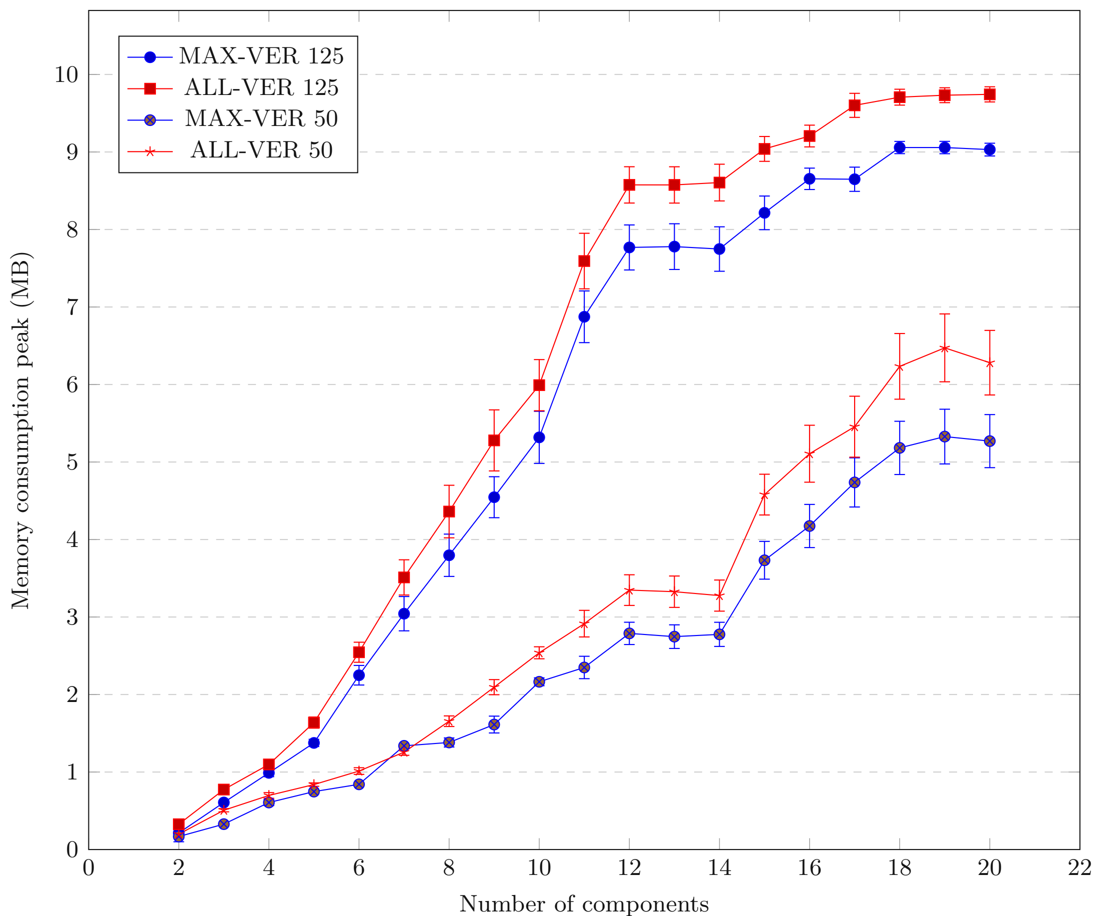

Figure 7.9: Dependency resolution memory footprint (growing number of components)

Plan, it still enhances the user experience by providing a great degree of overview. Let us remind the reader, when using the manual approach, there is no information on the installations at all.

2. Follow-a-plan approach

   The Deployment Plan portrays each of the actions or steps to complete the whole updating transaction which is structured and easy-to-follow. When an anomaly is detected, the Plan could be adjusted, so it is in a sense not static but a living document.

3. Satisfiability consistency

   One of the major advantages of using the proposed system is that each step of the whole procedure can be verified for satisfiability. It is not only useful to identify a satisfiable combination of dependencies when a Deployment Plan is created, but also while each of the nodes is visited. When we previously applied the solution to

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the scenario, it could determine a deviation in the global knowledge and suggest a change to bring the deployment back to a satisfiable state.

4. Automatic management of executable artefacts

   The artefacts are pushed to the target environment almost seamlessly and do not have to be manually copied, installed or moved around. It lets the users focus more on the ultimate goal rather than small details.

5. Eliminating miscellaneous steps

   Depending on the type of connection, the system eliminates the need to keep track of the connection information to the nodes, such as IP addresses, fully qualified domain names, etc. The same applies to the authentication information. When we applied the system to the scenario, a private key was used to authenticate the system via SSH. Altogether, it simplifies the whole procedure for the users, as they do not have to focus on the intermediary steps which do not contribute to achieving their goal.

6. Better management of startup configuration

   The system allows to transfer/load the base startup configuration needed for a single component or a group of components to operate properly. Due to the support of plugins as described in Section 6.4, different container orchestration tools can be integrated. It comes along with potentially introducing another way of handling the configuration information. The internal representation of configuration can be altered depending on the users' needs. In contrast, the previous practice of dealing with startup configuration would involve manual steps and no comprehensive approach to tailoring the configuration to user-specific needs.

7. Transactional approach

   The whole updating procedure is considered a transaction, meaning when successful, it is closed, and it can be rolled back on failure. Since all the steps are constantly tracked, the user can be guided in the backwards direction to undo the made changes. It allows bringing the software system, all its components, to the previous state known to be working.

The system clearly introduces numerous advantages compared to the manual approach, however it reveals a few **disadvantages** as well:

1. Limited freedom

   The system enables container-based deployment using Docker as well as offers a plugin architecture for container orchestration. On top of this, it allows defining user-specific configuration when a deployment is performed. Nevertheless, it does not provide a solution for special use cases instantly. For example, deploying to Windows or Apple's OS would require developing completely new software extensions. There

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

is also no out-of-the-box support for a more advanced configuration, which would use pre- and post-execution scripts or similar. The manual approach offers clearly a great degree of freedom which is constrained by technical limitations only.

2. False positives

   Humans set the dependency constraints defined between components and their versions without verifying whether the specified constraint really exists. So by mistake, a version can be mixed up with another, potentially leading to a false-positive result. In Section 3.2.3 we mentioned the automatic discovery of dependencies could mitigate this problem, however, it is out of the scope of this work.

   Besides that, we want to point out the manual approach also does not protect from delivering false positives, since it is prone to human errors as well.

3. Potential breach of sensitive data

   The system possesses comprehensive knowledge about execution environments, authentication points and software configuration, all of which can be theoretically exploited by a third person with malicious attempts. We argue this is only a semi-disadvantage, because no harm can be caused if the system's security is handled properly.

7.4.2 Performance

This Section summarizes the performance of the proposed solution's core module responsible for dependency resolution. The results were obtained by iterating the tests 30 times, then calculating mean values and confidence intervals with a confidence level of 95%.

Based on the four provided Figures (7.6, 7.8, 7.9, 7.7) we showed how the execution time grows depending on how does the size of the problem – by increasing the number of components with fixed versions per components (50 and 125), and vice versa, by increasing the number of versions with a fixed number of components (5 and 15). The selected input data reflect both the realistic size of the problem and intentionally overgrown one to demonstrate the over-provisioning capacities and scalability.

The tests show both strategies, MAX-VER and ALL-VER, can handle a realistic number of versions per component within an acceptable time budget. For a system with 15 components and 50 versions each, the execution time of calculating the dependencies results in around 2.5–4.0 seconds, depending on the applied strategy. The constellation of 15 components and 50 versions per component is the most realistic one when dealing with global knowledge. When local knowledge is considered, the studied example of 5 components and 50 version per component is representative, and the execution time is at around 0.4 seconds for both strategies.

The results show that the memory consumption does not grow linearly with the size of the problem. Overall, the memory allocation peaks at around 12 MB for a system with 15 components and reaches 3 MB for a system with 5 components. There is only a slight

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

difference in the results between different strategies. Generally, the memory footprint is arguably insignificant.

The tests were conducted on a mobile computer (laptop) of a workstation-class, meaning, it utilizes a higher performance CPU compared to other mobile computers. This testing platform is highly representative, as the system is expected to be hosted on a similar mobile workstation. In Section 4.8 we discussed the options and the reasons behind the decision making; however, because the SMT module is fully externalized and the system embraces the microservice architecture, it could be running on an external host via a network. In contrast, the core application is hosted on a low-performance device.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 8

Conclusion

A central activity within the lifecycle of service-based systems is management of their software updates. Although it is a problem that has been widely tackled by the community in the past, settings where security constraints impose compulsory network isolation call for specialized treatment. To this end, we adopted an architectural viewpoint and presented a technical framework for updating service-based systems in air-gapped environments. After describing the particularities of the domain, we provided suitable modelling notations for service versions, whereupon satisfiability is used for dependency resolution; an overall architecture was presented in an end-to-end solution. We evaluated the applicability of the framework over a realistic case study of an international organization, and assessed the performance of the dependency resolution procedures for practical problem sizes.

8.1 Discussion

Based on our evaluation results, we believe to have demonstrated that our framework facilitates the update process for air-gapped systems. A typical scenario was elicited and modelled in Section 7.1, demonstrating applicability; we successfully modelled a realistic scenario elicited from stakeholders without running into any conceptual issues with regard to our notions of service versions, dependency management and architecture materializing air-gapped updates. On a functional level, a satisfiable combination of versions is computed and a deployment plan is formed, to be installed via the physical visit where the service artifacts are pushed to the target host. Furthermore, the architecture illustrated in Figure 6.2 provides for an end-to-end solution, including configuration management, user authentication and container management. Since we followed versioning best practices (tailored for contemporary service-based systems) and employed satisfiability which is widely applied for version management, we believe internal threats to validity of our results to be minimal. However, we note that the case study, although realistic and catering to the needs of an international organization, implied certain type and number

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

of service components, as well as certain design choices in the overall service-based architecture. This is additionally relevant to the quantitative analysis of the dependency resolution; vastly different systems or with different update procedures would imply changes to the workflow and dependency resolution strategies. This would point that results of the case study may not apply to highly diverse cases, which is a threat to external validity. We believe identifying variation points in the architecture presented as a promising avenue of future work.

8.2 Summary

This work is based on the use case of an on-site inspection performed as part of the verification regime of the Comprehensive Nuclear-Test-Ban Treaty (CTBT). The Preparatory Commission for the Comprehensive Nuclear-Test-Ban Treaty Organization (CTBTO) was founded in 1996 and has a Relationship Agreement with the United Nations. The organization is comprised of three technical divisions: International Monitoring System, International Data Centre and On-Site Inspection. The organization's main objective is to prevent the world from testing nuclear weapons by monitoring air, water, seismic activities, etc. and inspecting on-site if in doubt a test has been indeed performed. Such an inspection implies stringent security mechanisms, including air-gapped networks. This imposes a unique problem, since distributed components of service-oriented software systems are scattered across multiple air-gapped servers/networks. In particular, we refer to the added complexity updating such distributed systems.

In Chapter 2 we review the related work to be aware of the current state of the art. Further in Chapter 3 we cover fundamentals to provide background information about the studied problem. It includes but is not limited to service-oriented architecture, distributed systems, dependencies, software versioning and containerization.

In the next Chapter 4, we discuss the requirements and design of the software solution, which aims to solve this work's problem. We elicit requirements by involving CTBTO experts and transform them into user stories as described in Section 4.4. Next, we introduce an example software system comprised of multiple services in Section 4.5. This system is an approximation of what applications the on-site inspection division of CTBTO uses. It helps us to back up the problem visually, as well as being convenient to refer to it at later points of time. An important contribution is introducing such terms as global and local knowledge (see 4.6) to point out the difference between reliable local information and unreliable global assumptions about what components and what versions of those components are installed. Additionally, this section addresses the RQ4. Based on predictability (see 4.7) we argue how the reliability of global knowledge can be improved. Next, we cover semantic versioning, a commonly-used versioning approach. We show how converting a version string to an integer can help us utilize SMT for dependency resolution. Section 4.11 answers RQ2 by performing modelling using various UML diagram types such as Use Case, Class, Object and Activity.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Chapter 5 unveils the main contribution of this work – an SMT-based solution for dependency resolution, which help us to approach the RQ3. After covering the SMT basics, we discuss the underlying problem, including constructing facts and domain constraints. We outline several solution approaches. Most importantly, we discuss the implemented MAX-VER and ALL-VER strategies. Section 5.2 refers to algorithmic ideation, where the proposed solution is presented first as first-order logic formula and then as a pseudocode to decouple the algorithm from its concrete implementation. Finally, the implementation in PySMT is outlined (see Section 5.3).

In Chapter 6 we outline the architecture of the proposed software solution. Since the scenario implies embracing high-security measures, the main considerations about those are discussed. Then we briefly discuss options for storing artefacts. Docker is the main platform targeted by our software solution. Therefore Section 6.3 describes in details how Docker can be best used in production. This includes Docker Swarm, an orchestration tool that is supplied with the Docker Engine by default. We discuss how configuration management of Docker Swarm services can be handled and the advantages and disadvantages of various approaches. The proposed solution strives to be highly flexible and modular. Thus, Section 6.4 explains how the embraced plugin-based architecture avoids vendor lock-in by allowing integrating alternative container orchestrators. We implemented one plugin that enables Docker Swarm. But, for example, Google's Kubernetes, currently a prevalent solution, can be integrated by implementing a new plugin.

The next Chapter 7 communicates the applied evaluation methods and their results. In Section 7.1 we present a realistic scenario elicited based on interviews with CTBTO experts. This scenario serves as a verification measure to prove the solution's applicability, while it also helps us to answer the RQ1. It covers a very concrete example and presents a detailed explanation of how the proposed solution would be applied. Further, the original requirements are revisited, and the fulfilled and unfulfilled user stories are identified in Section 7.2. The next Section 7.3 deals with quantitative metrics to measure the performance of the SMT dependency resolution module – this work's main contribution. We display multiple line charts showing such parameters as execution time, size of the problem, and memory consumption. Moreover, the differences between solving strategies are pointed out and alternating behaviour depending on the input data. The performance evaluation helps us to answer the RQ5. Last but not least, in Section 7.4 we communicate the results of the conducted evaluation, and discuss the benefits and shortcomings of the solution.

We begin the last Chapter 8 by summarising each of the chapters and pointing out the highlights. Afterwards, we present future possible work to further enhance the solution.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

8.3 Future Work

The high modularity and extensibility of the system foresee future modifications. The class diagram from Figure 4.11.2 shows how multiple Execution Environments can be supported with the help of an abstraction. So far, we have implemented support for Docker Execution Environments; however, no support for Android has been introduced as yet. For the future development of the system, environments such as Android and Windows could be of high interest.

The plugin support described in Section 6.4 avoids vendor lock-in to one specific container orchestration tool. Currently, only one plugin has been implemented enabling Docker Swarm, Docker's native orchestration solution. In the future, supporting Google's Kubernetes could be favourable.

The system is expected to be used in a setup where data security is of the highest priority. The CTBTO domain experts express concern that the ISP might reject establishing network connections to the air-gapped systems, making the solution impractical. He or she could argue that the system acquires classified data. This problem is reflected in one of the previously collected user stories (see 4.4.5). Future work can cover this user story by providing proof that the system does not collect data it should not be collecting.

In Section 4.2 we discussed master/agent architecture. In our work, we implemented a prototype of an agent, however, we did not engineer the master. Essentially, the master would require a very trivial implementation, as it just collects, aggregates and presents the data supplied by the agent(s). One simple approach would be to upload new data each time an agent can reach the master, meaning it is not in offline mode fulfilling a Deployment Plan.

As identified in Section 7.4.1, one of the system's drawbacks is that the dependency relations are defined manually by humans. This can lead to a false-positive result and subsequently to an error. To mitigate this issue, automatic discovery of dependencies can be applied, as discussed in 3.2.3. Even if an exact automatic discovery is most likely out of the question, at least the user can be guided with useful suggestions. For example, automatic analysis of a REST API and its consumer(s) can show one of the consumers tries to call a non-existing endpoint. This could be translated to a false dependency version and shown to the user as a suggestion.

In Section 4.7 we stated, because of the global knowledge's unreliability, increasing the number of shipped service/component versions increases the probability the update succeeds. The proposed function f, which depends on completeness, can improve the success rate of our software solution. Its implementation could be a great extension to the solution.

The case study showed a situation could emerge where the relevant parties do not trust each other. For example, the ISP may claim during an OSI, a collected picture has been altered while it was passed through the chain of applications. Also, currently, it is impossible to prove that a particular artefact was built using a particular source code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

However, by using a dedicated approach similar to hashing the authenticity of artefacts can be proven. The system could sign each of the artefacts it is dealing with, increasing transparency and trust between the relevant parties.

In the introduced case study, the solution addresses specific applications developed by the CTBTO to facilitate the progress of an inspection while respecting the required air gaps. As stated, the solution thus far does not address proprietary software, for example, ArcGIS geographic information system, SODIGAM gamma-ray analysis software and so on. Herein lines a potential major future work to expand the solution. An inspection team may use in excess of 30 different software packages to process data acquired by inspectors during field and laboratory activities. These software packages are typically inspection technique specific, in certain cases, node or dongle locked and not necessarily OS forward compatible. Ensuring the compatibility of these software packages with application PWA and others (see Figure 4.1) is fundamental to inspection team operations.

Last but not least, we presented two dependency resolution strategies, MAX-VER and ALL-VER. The tests in the Section 7.3 showed good results in terms of performance. Nevertheless, we can imagine other strategies tailored to some specific needs could be more efficient. Therefore, in the future, more dependency resolution strategies could be developed and adopted.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

Abbreviations

This Section lists all used abbreviations throughout the document.

UN: United Nations

CTBTO: Comprehensive Nuclear-Test-Ban Treaty Organization

(P)TS: (Preliminary) Technical Secretariat

CTBT: Comprehensive Nuclear-Test-Ban Treaty

ISP: inspected state party

OSI: on-site inspection

WA: working area

RA: receiving area

GUI: graphical user interface

UI: user interface

VPN: virtual private network

OS: operating system

SOA: service-oriented architecture

JVM: Java virtual machine

SMT: Satisfiability Modulo Theories

OOP: object-oriented programming

CBD: Component-Based Development

WSDL: Web Services Description Language

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

SOAP: Simple Object Access Protocol

REST: Representational state transfer

VM: virtual machine

DSL: domain specific language

FOSS: Free and Open Source Software

VPN: virtual private network

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub

APPENDIX B

Screenshots

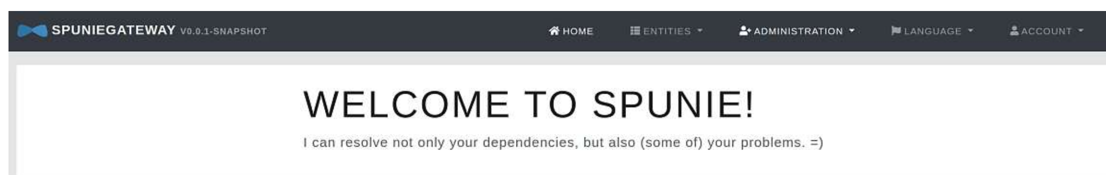

Figure B.1: GUI: landing page view when logged in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

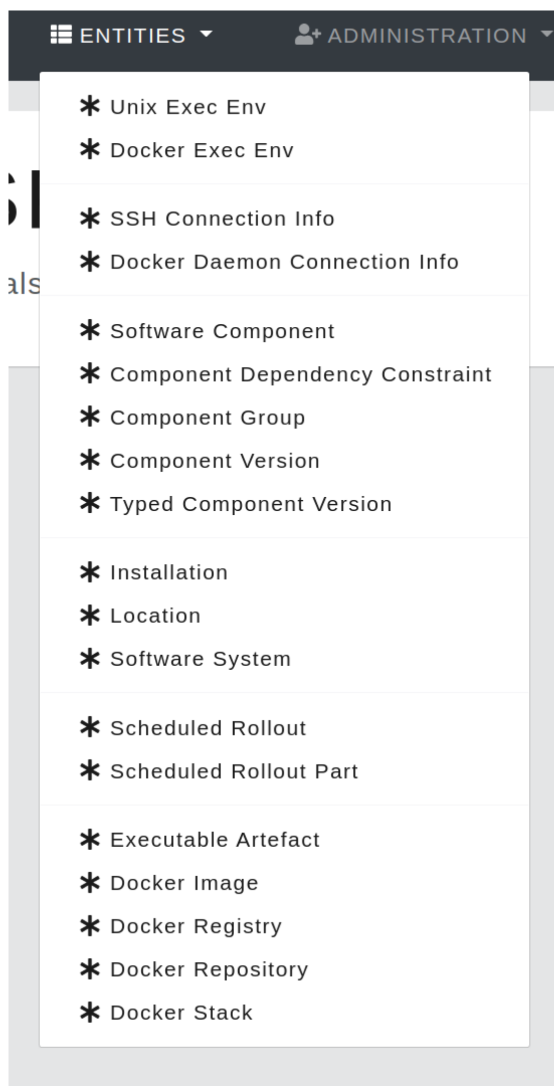

Figure B.2: GUI: entities dropdown menu

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

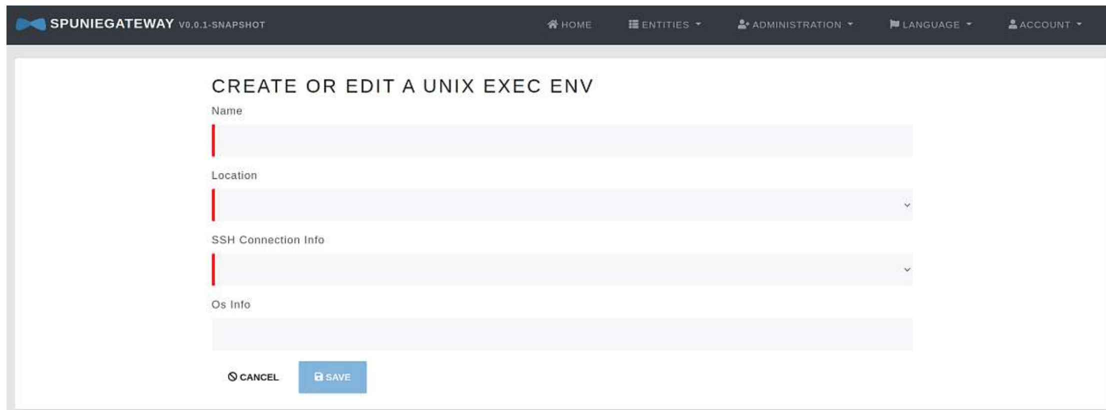

Figure B.3: GUI: create dialog of Unix Exec Env entity

Figure B.4: GUI: list of Software Component entities

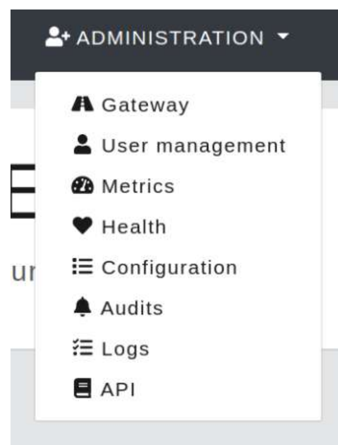

Figure B.5: GUI: admin dropdown menu

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

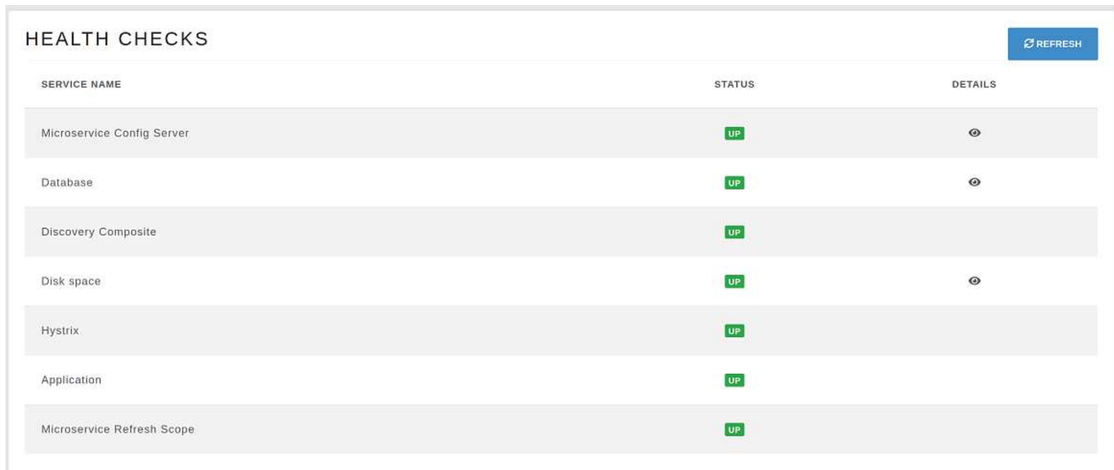

Figure B.6: GUI: health checks view

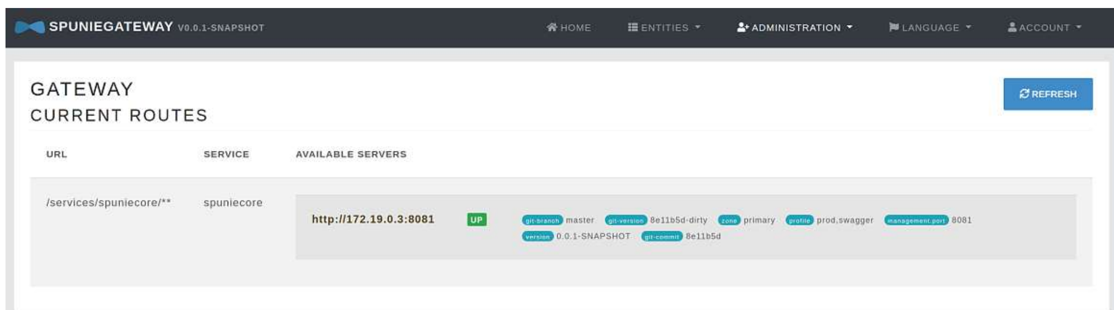

Figure B.7: GUI: list of gateway routes

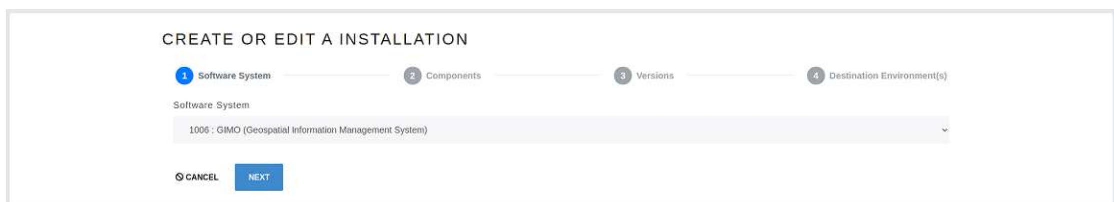

Figure B.8: GUI: provisioning new system - step 1

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

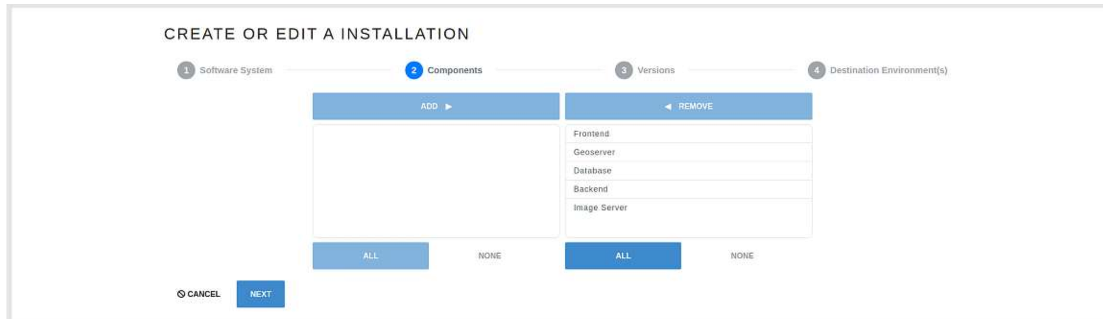

Figure B.9: GUI: provisioning new system - step 2

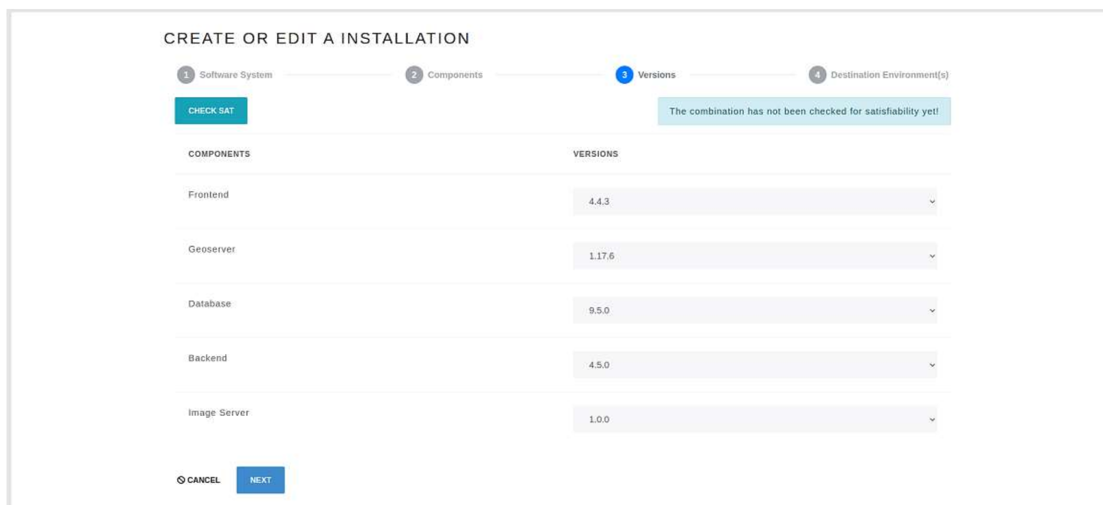

Figure B.10: GUI: provisioning new system - step 3

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

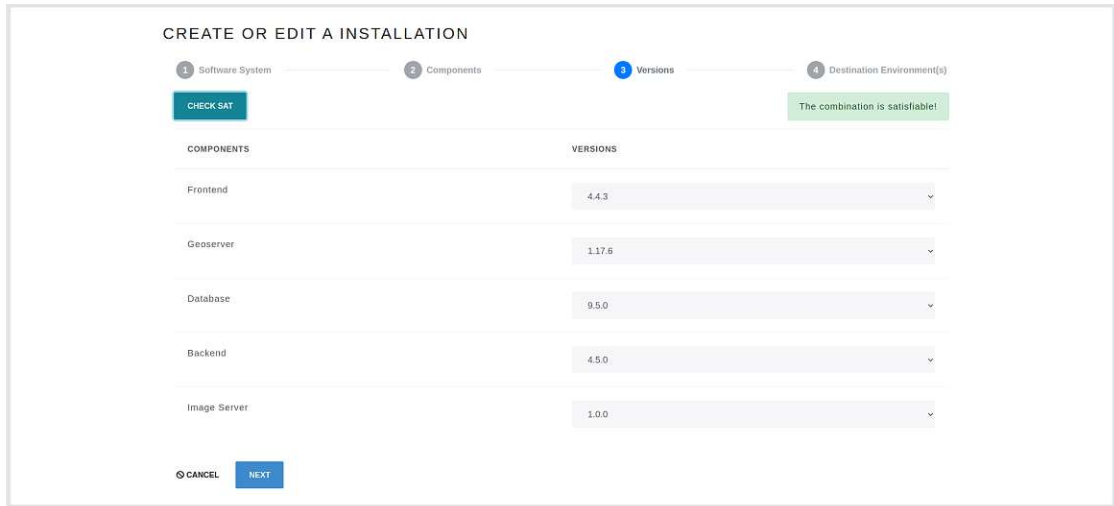

Figure B.11: GUI: provisioning new system - step 3 - satisfiable combination

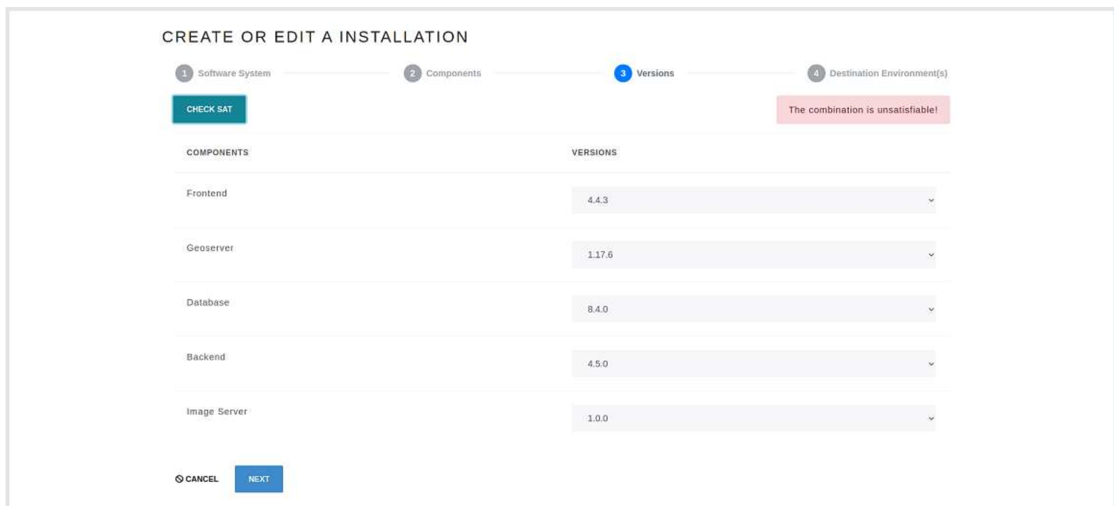

Figure B.12: GUI: provisioning new system - step 3 - unsatisfiable combination

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

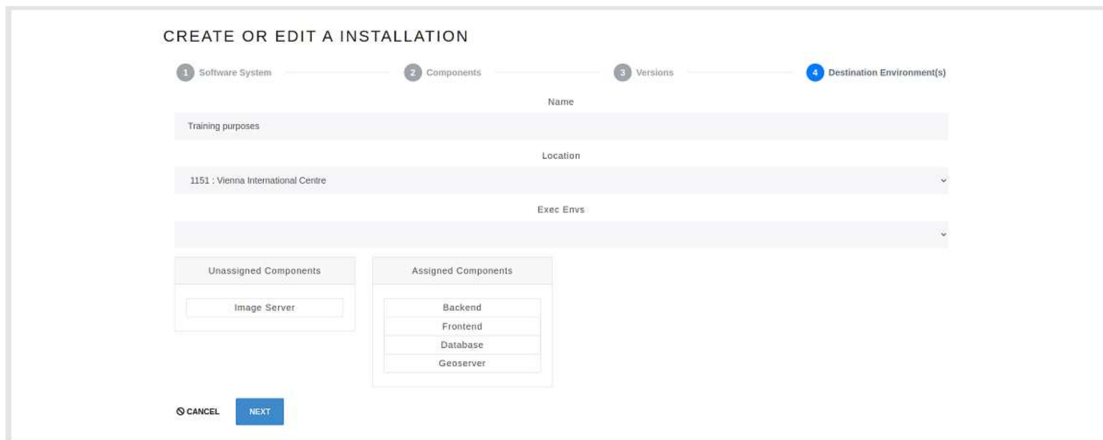

Figure B.13: GUI: provisioning new system - step 4

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX C

Listings

C.1 Docker swarm configuration

Listing C.1: Base backend configuration file

```
version: "3.2"

services:
  movieplex7:
    image: "${movieplex7_image}"
    ports:
      - "8080:8080"
    networks:
      - www
    deploy:
      labels:
        at.ac.tuwien.dsg.spunie.image.id: "${movieplex7_image_id
          ↪ }"
      replicas: 2
      update_config:
        parallelism: 2
        failure_action: rollback
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s

networks:
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
   www:
```

Listing C.2: Base frontend configuration file

```
version: "3.2"

services:
  react-client:
    image: "${react_client_image}"
    ports:
      - "80:80"
    networks:
      - www
    deploy:
      labels:
        at.ac.tuwien.dsg.spunie.image.id: "${
            ↪ react_client_image_id}"
      replicas: 2
      update_config:
        parallelism: 2
        failure_action: rollback
      restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
        window: 120s

networks:
    www:
```

Listing C.3: User-defined frontend configuration file

```
version: "3.2"

services:
  react-client:
    ports:
      - "80:80"
    deploy:
      replicas: 1
```

C.2 Performance testing

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing C.4: Running multiple cycles of tests

```
# global objects or data
max_sat_runner = MaxSatTestRunner()
all_sat_runner = AllSatTestRunner()
cycles = 30 # defines show many iterations are going to be made

# Tests with fixed 5 components and variable versions (see
    ↪ iter_array)
data_generator = FiveCompsDataGenerator()
file_path = f'{self.test_results_root_dir}/
    ↪ test_result_5_components.dat'
iter_array = [2, 5, 10, 20, 40, 60, 80, 100, 125, 160]

# run
self.conduct_test(iter_array, file_path, self.
    ↪ headers_table_tests_increase_nr_versions,
                cycles, data_generator, max_sat_runner,
                    ↪ all_sat_runner)

# Tests with fixed 15 components and variable versions (see
    ↪ iter_array)
data_generator = FifteenCompsDataGenerator()
file_path = f'{self.test_results_root_dir}/
    ↪ test_result_15_components.dat'

# run
self.conduct_test(iter_array, file_path, self.
    ↪ headers_table_tests_increase_nr_versions,
                cycles, data_generator, max_sat_runner,
                    ↪ all_sat_runner)

# ---------------------------------
# increasing number of components with 50 versions
data_generator = IncreaseComponentsDataGenerator(vers_per_comp
    ↪ =50)
file_path = f'{self.test_results_root_dir}/
    ↪ test_result_incr_comps_50_vers.dat'
iter_array = range(2, 16)

# run
self.conduct_test(iter_array, file_path, self.
    ↪ headers_table_tests_increase_nr_components,
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
              cycles, data_generator, max_sat_runner,
                 ↪ all_sat_runner)

# increasing number of components with 125 versions
data_generator = IncreaseComponentsDataGenerator(vers_per_comp
   ↪ =125)
file_path = f'{self.test_results_root_dir}/
   ↪ test_result_incr_comps_125_vers.dat'

# run
self.conduct_test(iter_array, file_path, self.
   ↪ headers_table_tests_increase_nr_components,
              cycles, data_generator, max_sat_runner,
                 ↪ all_sat_runner)
```

Listing C.5: Test data generators

```
class ITestDataGenerator:
   def get_test_data(self, iter_var: int) -> ([Component], [
      ↪ ComponentDependencyConstraint], Component):
      pass

class FifteenCompsDataGenerator(ITestDataGenerator):
   def get_test_data(self, multiplier: int) -> ([Component], [
      ↪ ComponentDependencyConstraint], Component):
      return generate_comp_versions_15_comps(multiplier)

class IncreaseComponentsDataGenerator(ITestDataGenerator):
   def __init__(self, vers_per_comp: int):
      self.__vers_per_comp = vers_per_comp
   def get_test_data(self, nr_components: int) -> ([Component],
      ↪  [ComponentDependencyConstraint], Component):
      return generate_comp_versions_var_comp_nr(self.
         ↪ __vers_per_comp, nr_components)
```

Listing C.6: Strategy test runner

```
class ITestRunner:
   def __init__(self):
      self.resolver = None
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
    def run(self, comps: [Component], dep_constraints: [
        ↪ ComponentDependencyConstraint], fixed_ver_comp:
        ↪ Component):
        pass


class MaxSatTestRunner(ITestRunner):
    def run(self, comps: [Component], constraints: [
        ↪ ComponentDependencyConstraint], fixed_ver_comp:
        ↪ Component):
        self.resolver.apply_max_sat(comps=comps + [fixed_ver_comp
            ↪ ], dep_constraints=constraints)


class AllSatTestRunner(ITestRunner):
    def run(self, comps: [Component], constraints: [
        ↪ ComponentDependencyConstraint], fixed_ver_comp:
        ↪ Component):
        self.resolver.apply_all_sat(rest_comps=comps,
            ↪ dep_constraints=constraints,
                                    comps_with_ver_to_lock=[
                                        ↪ fixed_ver_comp])
```

Listing C.7: Generate data with 5 components

```
def generate_comp_versions_5_comps(multiplier: int):
    id_gen = [0]

    def nextId(ids):
        ids[0] = ids[0] + 1
        return ids[0]

    # 5 Active components
    pra_w_id = nextId(id_gen)
    pra_d_id = nextId(id_gen)
    pra_g_id = nextId(id_gen)
    pra_r_id = nextId(id_gen)

    field_app_id = nextId(id_gen)
    field_data_id = nextId(id_gen)

    # Dependency constraints
    constr_fixed = ConstraintType.FIXED
    constr_patch_var = ConstraintType.PATCH_VAR
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
constr_minor_var = ConstraintType.MINOR_VAR

constraints = list()

# pra
# never applies
add_new_constr(constraints, "18.6.0", pra_w_id, constr_fixed,
    ↪  "9.6.0", pra_d_id, constr_fixed)
# applies
add_new_constr(constraints, "18.7.*", pra_w_id,
    ↪ constr_patch_var, "2.14.*", pra_g_id, constr_patch_var)
# applies
add_new_constr(constraints, "18.*.*", pra_w_id,
    ↪ constr_minor_var, "1.*.*", pra_r_id, constr_minor_var)
# applies
add_new_constr(constraints, "18.7.*", pra_w_id,
    ↪ constr_patch_var, "9.*.*", pra_d_id, constr_minor_var)

# field app
add_new_constr(constraints, "4.0.0", field_app_id,
    ↪ constr_fixed, "2.8.*", field_data_id, constr_patch_var)
add_new_constr(constraints, "3.*.*", field_app_id,
    ↪ constr_minor_var, "1.*.*", field_data_id,
    ↪ constr_minor_var)

# field app -> pra
add_new_constr(constraints, "4.*.*", field_app_id,
    ↪ constr_minor_var, "18.7.*", pra_w_id, constr_patch_var)
add_new_constr(constraints, "4.1.*", field_app_id,
    ↪ constr_patch_var, "18.8.0", pra_w_id, constr_fixed)
add_new_constr(constraints, "3.*.*", field_app_id,
    ↪ constr_minor_var, "17.*.*", pra_w_id, constr_minor_var)

# pra -> field app
add_new_constr(constraints, "18.7.*", pra_w_id,
    ↪ constr_patch_var, "4.0.*", field_app_id,
    ↪ constr_patch_var)
add_new_constr(constraints, "18.8.0", pra_w_id, constr_fixed,
    ↪  "4.1.*", field_app_id, constr_patch_var)
add_new_constr(constraints, "17.*.*", pra_w_id,
    ↪ constr_minor_var, "3.*.*", field_app_id,
    ↪ constr_minor_var)
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
# PRA

pra_w_versions = list()
add_versions(pra_w_versions, pra_w_id, ["18.6.0"]) # +
    ↪ gen_vers_between("1.8.0", "20.0.0", multiplier))

pra_d_versions = list()
add_versions(pra_d_versions, pra_d_id,
          gen_vers_between("9.0.0", "9.9.0", multiplier) +
          gen_vers_between("9.0.0", "9.9.0", 3))

pra_g_versions = list()
add_versions(pra_g_versions, pra_g_id,
          ["2.14.5", "2.13.59", "2.14.2", "2.14.1", "2.14.0"
             ↪ ] + gen_vers_between("0.1.0", "2.13.0",
             ↪ multiplier))

pra_r_versions = list()
add_versions(pra_r_versions, pra_r_id,
          ["1.9.0", "1.8.0", "2.0.1"] + gen_vers_between("
             ↪ 1.0.0", "10.7.0", multiplier))

# field app
field_app_versions = list()
add_versions(field_app_versions, field_app_id,
          ["4.0.0"] + gen_vers_between("1.8.0", "3.9.20",
             ↪ multiplier))

field_data_versions = list()
# add_versions(field_data_versions, field_data_id,
# gen_vers_between("2.8.0", "2.8.30", 1) + gen_vers_between
    ↪ ("1.5.0", "1.8.0", 3))
add_versions(field_data_versions, field_data_id,
          gen_vers_between("1.0.0", "2.8.90", multiplier) +
             ↪ gen_vers_between("1.5.0", "1.8.0", 3))

# ----------------------Components---------------------
# PRA
pra_web_core = Component(pra_w_id, "pra_web_core",
    ↪ pra_w_versions)
pra_database = Component(pra_d_id, "pra_database",
    ↪ pra_d_versions)
pra_geoserver = Component(pra_g_id, "pra_geoserver",
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
        ↪ pra_g_versions)
    pra_routing = Component(pra_r_id, "pra_routing",
        ↪ pra_r_versions)

    # Field App
    field_app = Component(field_app_id, "field_app",
        ↪ field_app_versions)
    field_app_database = Component(field_data_id, "
        ↪ field_app_database", field_data_versions)

    all_comps = [pra_database, pra_geoserver, pra_routing] + \
                [field_app, field_app_database]
    return all_comps, constraints, pra_web_core
```

Listing C.8: Conduction of a test

```
def conduct_test(self, iter_array: [int], file_path: str,
    ↪ file_headers: [str], cycles: int,
                generator: ITestDataGenerator, testRunnerA:
                    ↪ ITestRunner, testRunnerB: ITestRunner):
    resolver = Resolver()
    testRunnerA.resolver = resolver
    testRunnerB.resolver = resolver

    file = open(file_path, 'w')
    file.write('\t'.join(file_headers) + '\n')
    for iter_val in iter_array:
        arr_max_sat_time = []
        arr_max_sat_mem_peak_mb = []
        arr_all_sat_time = []
        arr_all_sat_mem_peak_mb = []
        for cycle in range(cycles):
            all_comps, constraints, fixed_ver_comp = generator.
                ↪ get_test_data(iter_val)

            tracemalloc.start()
            time_start = time.time()

            testRunnerA.run(all_comps, constraints, fixed_ver_comp
                ↪ )

            time_stop = time.time()
            max_sat_time = time_stop - time_start
            current_max_sat, peak_max_sat = tracemalloc.
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
            ↪ get_traced_memory()
        max_sat_mem_peak_mb = peak_max_sat / 10 ** 6
        arr_max_sat_time.append(max_sat_time)
        arr_max_sat_mem_peak_mb.append(max_sat_mem_peak_mb)

        #####################

        time_start = time.time()
        tracemalloc.stop()
        tracemalloc.start()

        testRunnerB.run(all_comps, constraints, fixed_ver_comp
            ↪ )

        time_stop = time.time()
        all_sat_time = time_stop - time_start
        current_all_sat, peak_all_sat = tracemalloc.
            ↪ get_traced_memory()
        all_sat_mem_peak_mb = peak_all_sat / 10 ** 6
        tracemalloc.stop()
        arr_all_sat_time.append(all_sat_time)
        arr_all_sat_mem_peak_mb.append(all_sat_mem_peak_mb)

    m_max_sat_time, ci_max_sat_time = self.
        ↪ calc_confidence_interval(arr_max_sat_time)
    m_max_sat_mem_peak_mb, ci_max_sat_mem_peak_mb = self.
        ↪ calc_confidence_interval(arr_max_sat_mem_peak_mb)
    m_all_sat_time, ci_all_sat_time = self.
        ↪ calc_confidence_interval(arr_all_sat_time)
    m_all_sat_mem_peak_mb, ci_all_sat_mem_peak_mb = self.
        ↪ calc_confidence_interval(arr_all_sat_mem_peak_mb)
    file.write('\t'.join([str(iter_val),
                    str(m_max_sat_time), str(ci_max_sat_time)
                        ↪ ,
                    str(m_max_sat_mem_peak_mb), str(
                        ↪ ci_max_sat_mem_peak_mb),
                    str(m_all_sat_time), str(ci_all_sat_time)
                        ↪ ,
                    str(m_all_sat_mem_peak_mb), str(
                        ↪ ci_all_sat_mem_peak_mb)
                    ]) + '\n')

file.close()
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Listing C.9: Calculating confidence interval

```
def calc_confidence_interval(self, arr, confidence=0.95):
    m_i = mean(arr)
    std_err = sem(arr)
    ci_i = std_err * t.ppf((1 + confidence) / 2, len(arr) - 1)
    return m_i, ci_i
```

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

List of Figures

1.1 Updating service-based air-gapped systems – an overview 3

4.1 Example software setup 28
4.2 Master/agent architecture 32
4.3 UML Use Case diagram 35
4.4 UML Class diagram - domain 40
4.5 UML Object diagram - domain 41
4.6 UML Activity diagram - update procedure 42
4.7 UML Activity diagram - provision execution environment 43

6.1 UML Class diagram - plugin-based approach 69
6.2 High-level solution's architecture diagram 70

7.1 Scenario: software setup highlighting components to update 75
7.2 Scenario: airborne photograph of a simulated OSI from a training event showing deployed applications 76
7.3 Scenario: add new versions, dependency constraints and artefacts (simplified) 77
7.4 Schedule Deployment Plan (simplified) 79
7.5 Scenario: visiting each node (simplified) 80
7.6 Dependency resolution performance (growing number of versions) 85
7.7 Dependency resolution performance (growing number of components) . . 86
7.8 Dependency resolution memory footprint (growing number of versions) . . 87
7.9 Dependency resolution memory footprint (growing number of components) 88

B.1 GUI: landing page view when logged in 101
B.2 GUI: entities dropdown menu 102
B.3 GUI: create dialogue of Unix Exec Env entity 103
B.4 GUI: list of Software Component entities 103
B.5 GUI: admin dropdown menu 103
B.6 GUI: health checks view 104
B.7 GUI: list of gateway routes 104
B.8 GUI: provisioning new system - step 1 104
B.9 GUI: provisioning new system - step 2 105
B.10 GUI: provisioning new system - step 3 105
B.11 GUI: provisioning new system - step 3 - satisfiable combination 106

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

B.12 GUI: provisioning new system - step 3 - unsatisfiable combination 106
B.13 GUI: provisioning new system - step 4 107

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

3.1 Different classifications of version constraints as according to [DPS+19] . 18

5.1 Types of Facts (dependency resolution) 47
5.2 FOL notation 49

6.1 Docker Services configuration types. Advantages and disadvantages. . . . 66

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

5.1 Construction of the problem 50
5.2 Construction of the constraints 51
5.3 Construction of the exactly-one facts 52

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[ACGZ20] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. Dependency solving is still hard, but we are getting better at it. In *Proc. 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER 2020)*, 2020.

[AD16] Abdulrahman Azab and Diana Domanska. Software provisioning inside a secure environment as docker containers using stroll file-system. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 674–683. IEEE, 2016.

[ADCBZ09] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. Strong dependencies between software components. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 89–99. IEEE, 2009.

[ADCTZ12] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software*, 85(10):2228–2240, 2012.

[ALS06] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In Dave Thomas, editor, *ECOOP 2006 – Object-Oriented Programming*, pages 452–476, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[Ars04] Ali Arsanjani. Service-oriented modeling and architecture. *IBM developer works*, 1:15, 2004.

[BDNL02] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Software update via mobile agent based programming. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 32–36, 2002.

[BGMLM16] Luciano Baresi, Carlo Ghezzi, Xiaoxing Ma, and Valerio Panzica La Manna. Efficient dynamic updates of distributed components through version consistency. *IEEE Transactions on Software Engineering*, 43(4):340–358, 2016.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Bro00] Alan W Brown. *Large-scale, component-based development*, volume 1. Prentice Hall PTR Englewood Cliffs, 2000.

[BT18] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.

[Byr13] Eric Byres. The air gap: Scada's enduring security myth. *Communications of the ACM*, 56(8):29–31, 2013.

[CLC05] Michel Chaudron, Stig Larsson, and Ivica Crnkovic. Component-based development process and component lifecycle. *Journal of Computing and Information Technology*, 13(4):321–327, 2005.

[ctb96] Comprehensive nuclear-test-ban treaty, 9 1996.

[doca] Docker registry. `https://docs.docker.com/registry`. Accessed: 08.10.2020.

[docb] Docker swarm. `https://docs.docker.com/engine/swarm/`. Accessed: 26.11.2020.

[docc] Docker swarm service. `https://docs.docker.com/engine/swarm/how-swarm-mode-works/services/`. Accessed: 27.11.2020.

[DPS+19] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. Dependency versioning in the wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 349–359. IEEE, 2019.

[EF14] Hani A Elgabou and Alan M Frisch. Encoding the lexicographic ordering constraint in sat modulo theories. In *Thirteenth International Workshop on Constraint Modelling and Reformulation (ModRef 2014)*, pages 85–96, 2014.

[Elf06] Patrick Elftmann. Secure alternatives to password-based authentication mechanisms. *Lab. for Dependable Distributed Systems, RWTH Aachen Univ*, 2006.

[ES03] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

[FL14] Martin Fowler and James Lewis. Microservices, 2014. *URL: http://martinfowler.com/articles/microservices.html*, 1(1):1–1, 2014.

[GG01] GC Gannodt and Barbara D Gannod. An investigation into the connectivity properties of source-header dependency graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 115–124. IEEE, 2001.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[GGBR07] Daniel M German, Jesus M Gonzalez-Barahona, and Gregorio Robles. A model to understand the building and running inter-dependencies of software. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 140–149. IEEE, 2007.

[GKKE14] Mordechai Guri, Gabi Kedma, Assaf Kachlon, and Yuval Elovici. Airhopper: Bridging the air-gap between isolated networks and mobile phones using radio frequencies. In *2014 9th International Conference on Malicious and Unwanted Software: The Americas (MALWARE)*, pages 58–67. IEEE, 2014.

[GNAK03] Manish Gupta, Anindya Neogi, Manoj K Agarwal, and Gautam Kar. Discovering dynamic dependencies in enterprise environments for problem determination. In *International Workshop on Distributed Systems: Operations and Management*, pages 221–233. Springer, 2003.

[gra] Gravity landing page. `https://goteleport.com/gravity`. Accessed: 24.03.2021.

[GZE19] Mordechai Guri, Boris Zadov, and Yuval Elovici. Odini: Escaping sensitive data from faraday-caged, air-gapped computers via magnetic fields. *IEEE Transactions on Information Forensics and Security*, 15:1190–1203, 2019.

[KC00] Fabio Kon and Roy H Campbell. Dependence management in component-based distributed systems. *IEEE concurrency*, 8(1):26–36, 2000.

[Lan98] Danny B Lange. Mobile objects and mobile agents: The future of distributed computing? In *European conference on object-oriented programming*, pages 1–12. Springer, 1998.

[LB10] Florian Lonsing and Armin Biere. Depqbf: A dependency-aware qbf solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):71–76, 2010.

[LBP08] Daniel Le Berre and Anne Parrain. On sat technologies for dependency management and beyond. 2008.

[Leh80] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[Mar19] Jordan Marin. Deploying applications into air gapped environments. `https://goteleport.com/blog/airgap-deployment`, 2019. Accessed: 24.03.2021.

[MBC+06] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. Managing the complexity of large free and open source package-based software distributions. In *Proc. 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, 2006.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[MMZ+01] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535, 2001.

[MYV18] Jose Andre Morales, Hasan Yasar, and Aaron Volkmann. Implementing devops practices in highly regulated environments. In *Proc. 19th International Conference on Agile Software Development (XP 2018), Companion*, 2018.

[NSS14] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[OBL10] Joel Ossher, Sushil Bajracharya, and Cristina Lopes. Automated dependency resolution for open source software. In *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*, pages 130–140. IEEE, 2010.

[PLM12] Valerio Panzica La Manna. Local dynamic update for component-based distributed systems. In *Proceedings of the 15th ACM SIGSOFT symposium on Component Based Software Engineering*, pages 167–176, 2012.

[PW19] Tom Preston-Werner. Semantic versioning 2.0.0. 2013. *Online: http://semver.org*, 2019. Accessed: 25.10.2020.

[RK15] Ratneshwer and Pawan Kumar. Dependency analysis of a soa-based system through petri nets and service algebra. *International Journal of Software Engineering, Technology and Applications*, 1(2-4):172–189, 2015.

[RR00] James Robertson and Suzanne Robertson. Volere requirements specification template. *Atlantic System Guild www. systemguild. com*, 2000.

[RR12] Suzanne Robertson and James Robertson. *Mastering the requirements process: Getting requirements right*. Addison-wesley, 2012.

[Rus16] Cox Russ. Version sat. `https://research.swtch.com/version-sat`, 2016. Accessed: 22.10.2020.

[RvDV17] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

[Sch14] Mathijs Jeroen Scheepers. Virtualization and containerization of application infrastructure: A comparison. In *21st twente student conference on IT*, volume 21, 2014.

[SSS09] Sandeep K Sood, Anil K Sarje, and Kuldip Singh. Cryptanalysis of password authentication schemes: Current status and key issues. In *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*, pages 1–7. IEEE, 2009.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[SW04] David Sprott and Lawrence Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17, 2004.

[TPGN18] Christos Tsigkanos, Liliana Pasquale, Carlo Ghezzi, and Bashar Nuseibeh. On the interplay between cyber and physical spaces for adaptive security. *IEEE Trans. Dependable Sec. Comput.*, 15(3):466–480, 2018.

[WW18] Sunny Wong and Anne Woepse. Software development challenges with air-gap isolation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, page 815–820, New York, NY, USA, 2018. Association for Computing Machinery.

[XJR+18] Quanqing Xu, Chao Jin, Mohamed Faruq Bin Mohamed Rasid, Bharadwaj Veeravalli, and Khin Mi Mi Aung. Blockchain-based decentralized content trust for docker images. *Multimedia Tools and Applications*, 77(14):18223–18248, 2018.

[Yas18] Robail Yasrab. Mitigating docker security issues. *arXiv preprint arXiv:1804.05039*, 2018.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN