Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Self-Containment Packager Framework for TOSCA Cloud Service Archives

Yaroslav Nalivayko

Course of Study:	Informatik
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Michael Zimmermann, M. Sc.
Commenced:	10. February 2017
Completed:	10. August 2017
CR-Classification:	D.2.11, D.4.9

Abstract

In recent years, Cloud Computing is gaining more and more popularity. But if someone will try to create a Cloud Application suitable to work with several different platforms, he will face a problem. The problem is that each platform provides its own **A**pplication **P**rogramming **I**nterface (API) to interact with Cloud Applications. Therefore it's difficult to create one unified application functioning on various platforms properly. **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplication (TOSCA) provides a solution for this problem. With the help of TOSCA, it's possible to define several models of interaction with many different APIs within one TOSCA Application. A TOSCA runtime environment is responsible to choose and process the right model and serves as a layer between external interfaces of a TOSCA application and an API of a platform. This allows to automate the migration of TOSCA applications between platforms which use completely different APIs. The description of a TOSCA Application is stored in a **C**loud **S**ervice **AR**chive (CSAR), which contains all components necessary for the application life-cycle.
Cloud Applications are often defined in such way that during their deployment some external packages, programs and files need to be downloaded via the Internet. These downloads can slow down the deployment and when the access to the Internet is limited, unstable or missing, they can prevent the installation at all. In addition, the download from external sources can compromise the security of applications. This document considers the development of the solution to this problem through the predownload of the necessary data. Different methods of encapsulation of CSARs will be defined. It will be described the architecture of the software solution which can recognize external dependencies in a CSAR, eliminate them, resupply the CSAR with all the data necessary for deployment and also change the internal structure of the CSAR to display the achieved self-containment. The prototype of the software will be implemented and validated.

Contents

List of Figures

List of Listings

List of Abbreviations

API **A**pplication **P**rogramming **I**nterface. 13

APT **A**dvanced **P**ackaging **T**ool. 24

BPEL **B**usiness **P**rocess **E**xecution **L**anguage. 22

BPMN **B**usiness **P**rocess **M**odel and **N**otation. 22

CSAR **C**loud **S**ervice **AR**chive. 20

IaaS **I**nfrastructure **a**s **a** **S**ervice. 18

NIST **N**ational **I**nstitute of **S**tandards and **T**echnology. 17

OASIS **O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards. 18

PaaS **P**latform **a**s **a** **S**ervice. 18

SaaS **S**oftware **a**s **a** **S**ervice. 17

TOSCA **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplications. 13

1 Introduction

Cloud Applications market is increasing with great speed. Global annual growth is about 15% [Sta16]. Furthermore one can observe the growth in the number of firms which are using Cloud Applications. And it concerns not only some big corporations but also many small companies [Bun14; Bun17].
One of the most important reasons for the development of Cloud Applications is the economy of resources. It is much easier and often cheaper to rent a part of another's big platform than to maintain one's own server. The growing popularity of Cloud Applications makes the automation and the ease of management increasingly important. Management is understood as deployment, administration, maintenance and the final roll-off of Cloud Applications [HP09].
The common problem of Cloud Applications is a *vendor lock-in* [Ank+15]. This means the problem when an application configured to work with one provider can't be easily migrated to another provider and therefore the owner of the application is locked to the first provider. Each provider defines it own **A**pplication **P**rogramming **I**nterface (API) according to his preferences and the sphere of activity. The migration of a Cloud Application configured to interact with the API of one provider to another provider with another API is a difficult but important task. The ability to move a Cloud Application to the more suitable provider quickly and simply is a key to the development of competition and reducing the cost of maintenance. When each consumer can easily choose a provider with the best price quality ratio, it will stimulate providers to reduce costs and improve the service.
Topology and **O**rchestration **S**pecification for **C**loud **A**pplications (TOSCA) [OAS13a] is a standard to solve this problem. TOSCA defines a meta-model to describe Cloud Application's structure and management portable and interoperable. The use of TOSCA allows to simplify and automate the administration and migration of Cloud Applications. According to the TOSCA standard, a structure and internal data of a Cloud Application are stored in a **C**loud **S**ervice **AR**chive (CSAR). These archives describe Cloud Applications, their interfaces, internal dependencies, and behavior and contain the data for the deployment and operation, for example an image of a file system or executable files.
OpenTOSCA [IAA13] is an open source ecosystem for the TOSCA standard developed by the University of Stuttgart. It provides an runtime environment which can process CSARs and perform the management operations. However, installation operations often contain references to external packages, programs and files which will be subsequently

downloaded via the Internet during the deployment of a Cloud Application. These references can add expenses to the time required to download packages, money spent on rent of an idle platform and the Internet traffic for pre-known data. But the main problems of external dependencies are security and stability. To ensure the security of internal data some firms restrict the access to the Internet. In such systems, a user is not allowed to download or upload data without additional authorization. In other networks access to the Internet can be extremely limited. For example, if a platform hasn't broadband access or if communication is carried out only by satellite at certain hours. An attempt to deploy a Cloud Application with external dependencies in such networks may not succeed or lead to a breach of security.
During this work, a concept of a software solution for this problem will be developed. This concept will describe the resolution of external references in CSARs with the help of encapsulation. Since it is impossible to predict and describe all possible types of external dependencies, the software must be developed in the form of a framework which means an easily expandable modular system. To encapsulate a CSAR it is necessary to analyze it, identify dependencies to external files and packages, resolve them by downloading the files and the necessary installation data for the packages as well as installation data for all depended packages. Then all downloaded data must be added into the CSAR. Thus, these changes minimize or even eliminate the required access to the Internet during an application life-cycle. A prototype resolving external dependencies in CSARs will be implemented and validated.

Structure

The work structure is as follows:

Chapter 2 – Basics. This chapter explains the basic terms and technologies used in this work, which include definitions and descriptions of Cloud Applications (section 2.1), TOSCA standard (section 2.2), OpenTOSCA environment (section 2.3), packet management (section 2.4) and configuration management (section 2.5).

Chapter 3 – Requirements. It clarifies requirements for the developed solution.

Chapter 4 – Concept and Architecture. The main concepts as well as architecture of the framework are explained and illustrated in chapter 4.

Chapter 5 – Implementation. This chapter describes the implementation of the prototype. It explains the design and development of individual components of the software.

Chapter 6 – Example Integration of new Module. A new package manager will be added into the framework to proof the ease of extensibility.

Chapter 7 – Example Usage of the Tool. In this chapter the output of the developed program will be presented and validated.

Chapter 8 – Conclusion and Future Work. The results of the work will be summarized in the last chapter. Possible directions for further work will be presented.

2 Basics

In this chapter, the fundamentals used in this work will be explained. These include definitions for Cloud Computing and Cloud Applications, description of TOSCA and OpenTOSCA. At the end, the overview of packages management, files download and tools for its automation will be presented.

2.1 Cloud Computing and Cloud Application

To understand problems of *Cloud Computing* we require a clear definition of this term. Unfortunately, a generally accepted definition of Cloud Computing that describes all possible situations doesn't exist. But in the scientific community, the definition put forward by **N**ational **I**nstitute of **S**tandards and **T**echnology (NIST) [Nat] is commonly accepted. This definition appropriately describes the concept of Cloud Computing used in this paper, and therefore this definition will be used:
Cloud Computing *is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [Pet11].*
Also the is no generally accepted definition of Cloud Application, but it will be defined for this work using the Cloud Computing term. A **Cloud Application** is an application that is executed according to the Cloud Computing model. In addition, a short meanings of a Cloud system, a provider, and a consumer will be defined too. A composite Cloud Application which consists of multiple small applications will be called a **Cloud system**. An owner of a physical platform, where Cloud Computing takes place will be called a **provider**, an owner of a Cloud Application renting a provider's platform will be called a **consumer** [Mic+10].
Providers grant access to a wide range of different services: software, hardware devices, etc. Some groups of services which follow the common rules and perform the similar function are described by service models. NIST distinguishes between three main types of such models:

- **S**oftware **a**s **a** **S**ervice (SaaS). The capability provided to the consumer is to use the provider's applications running on a Cloud infrastructure. The applications

are accessible from various client devices through either a thin client interface, such as a web browser (e.g., web-based email), or a program interface. The consumer does not manage or control the underlying Cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited userspecific application configuration settings [Pet11].

- **P**latform **a**s **a** **S**ervice (PaaS). The capability provided to the consumer is to deploy onto the Cloud infrastructure consumer-created or acquired applications created using programming languages, libraries, services, and tools supported by the provider. The consumer does not manage or control the underlying Cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly configuration settings for the application-hosting environment [Pet11].
- **I**nfrastructure **a**s **a** **S**ervice (IaaS). The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. The consumer does not manage or control the underlying Cloud infrastructure but has control over operating systems, storage, and deployed applications; and possibly limited control of select networking components (e.g., host firewalls) [Pet11].

Usage of Cloud Computations

Nowadays Cloud Computing and applications can be found everywhere, and their number constantly grows [Laz16]. They are used for test, development, big data analyses, file storage, etc. Cloud Computing allows to use resources effectively, to distribute the load to multi-server systems and to shift the maintenance to the providers. If a service uses a single physical server and this server will be disabled, then the entire service will be completely unavailable too. But if a Cloud Application uses a hundred of physical servers, then disabling of one will not carry such serious consequences. In addition, a consumer doesn't need to maintain a team of administrators for the event of various problems.
Consumers don't have direct access to the provider's infrastructure (servers and operating systems) when using the PaaS or SaaS service models. They can operate only with the provided Application Programming Interface (API). An API defines a set of methods to communicate with provider's infrastructure. Each provider prepares his own set of methods depending on his area of specialization. On the one hand, this specialization makes easier to work with the provider, but on the other hand, it becomes more difficult to migrate an application to another provider.

2.2 Topology and Orchestration Specification for Cloud Applications

The **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplications (TOSCA) standard developed by **O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards (OASIS) [OAS] provides an opportunity to enable portable automated deployment and management of Cloud Applications. Using TOSCA it is possible to describe the structure of an application as a topology containing components and relationships between them. TOSCA application is a Cloud Application defined according to the TOSCA standard. To process TOSCA applications a TOSCA runtime environment must be executed on a provider's platform. Such runtime environment can process applications and serves as a layer between TOSCA applications and the provider's API. That allows to implement a single application suitable for work with different providers.

Structure of TOSCA Applications

TOSCA provides a language to define components of TOSCA applications and relationships between them. In addition it describes management procedures which create or modify services. The definition of elements of a TOSCA structure used in this work is provided.
$Service\ Templates$ are the main components of a TOSCA structure. They define Cloud services provided by TOSCA applications as a combination of their structure (Topology Template) and process models (Plans). It must be at least one Service Template within one TOSCA application. The combination of topology and orchestration defines what is needed to be preserved across operation in different environments. Thus it guarantees the interoperable deployment and management of Cloud services throughout the complete life-cycle and is useful when they are ported to alternative providers [OAS13b].
$Plans$ provide capabilities to manage Cloud Applications, especially their creation and termination. These components combine management capacity to create high-level administering tasks which can then be executed for fully automated deployment, configuration and other operations of the application. Plans can be started by a user or fully automatically and call management operations of the nodes in the topology. A $Topology\ Template$ describes the topology of a Cloud Application, defining nodes (Node Templates) and relations between them (Relationship Templates). A $Node\ Template$ instantiates a Node Type as a component of a service. A $Node\ Type$ describes the properties of such a component and the operations available to manipulate it. A $Relationship\ Template$ instantiates a Relationship Type as a relationship between Node Templates in a Topology Template. The Relationship Template indicates that two nodes are connected and defines the type and direction of the connection. A $Relationship\ Type$ specifies

semantics and properties of the type of connection. Node Types and Relationship Types can be instantiated multiple times. These types are like abstract classes in high-level programming languages and Templates are objects of these classes.
A simple cloud Application for weather calculation can be considered to provide an example. The calculation is performed by a python script which requires a python environment that is hosted on an Ubuntu virtual server. $Node\ Types$ must be defined for the python script, python environment and Ubuntu server. These $Node\ Types$ will describe available operations for specified components. It will be the $compute$ operation for the python script, the $install$ operation for the python environments and the $deploy$ and $shutdown$ operations for the server. Additionally, one must define $Relationship\ Types$ for $requires$ and $hosted\ on$ dependencies. Then these types will be instantiated inside the $Topology\ Template$ named $weather\ calculator$. For each specified $Node\ Type$, the corresponding $Node\ Template$ with unique identifiers is created. These identifiers are used by $Relationship\ Templates$ to define the dependencies. Figure 2.1 presents the described application.
Artifact represents the necessary content such as executables (e.g. a script or an executable program), configuration files, data files, or something that might be needed for other executables (e.g. libraries or images of file system). TOSCA distinguishes two kinds of artifacts: $Implementation\ Artifacts$ and $Deployment\ Artifacts$. $Implementation\ Artifacts$ represent the implementation of an operation described by a Node Type. $Deployment\ Artifacts$ represent the content needed to materialize an instances of a Node Type. In the example with the weather calculation, the python script will be an $Implementation\ Artifact$ associated with the $compute$ operation and the image of the Ubuntu virtual server will be a $Deployment\ Artifact$. $Artifact\ Types$ describe types of artifact: python script, installation package, etc. $Artifact\ Templates$ are used to describe artifacts itself. Each $Artifact\ Template$ contains information about one artifact: location, type and other attended data. $Node\ Type\ Implementations$ combine information about artifacts implementing the corresponding Node Type. If a Node Type contains $deploy$ and $shutdown$ operations, then its Node Type Implementation can contain two Implementation Artifacts with executables implementing these operations and one Deployment Artifact with data necessary for the materialization of instances. Implementations are like final classes between Node Types and Node Templates, but in TOSCA standard, the Implementation will be chosen only during execution. Types, Templates, and Implementations describing a TOSCA application are stored in definition documents which have the XML format.

CSAR

To pack a TOSCA application a **C**loud **S**ervice **AR**chive (CSAR) is used. This is a ZIP-file with ".csar" extension that contains all the data needed for instantiation and management

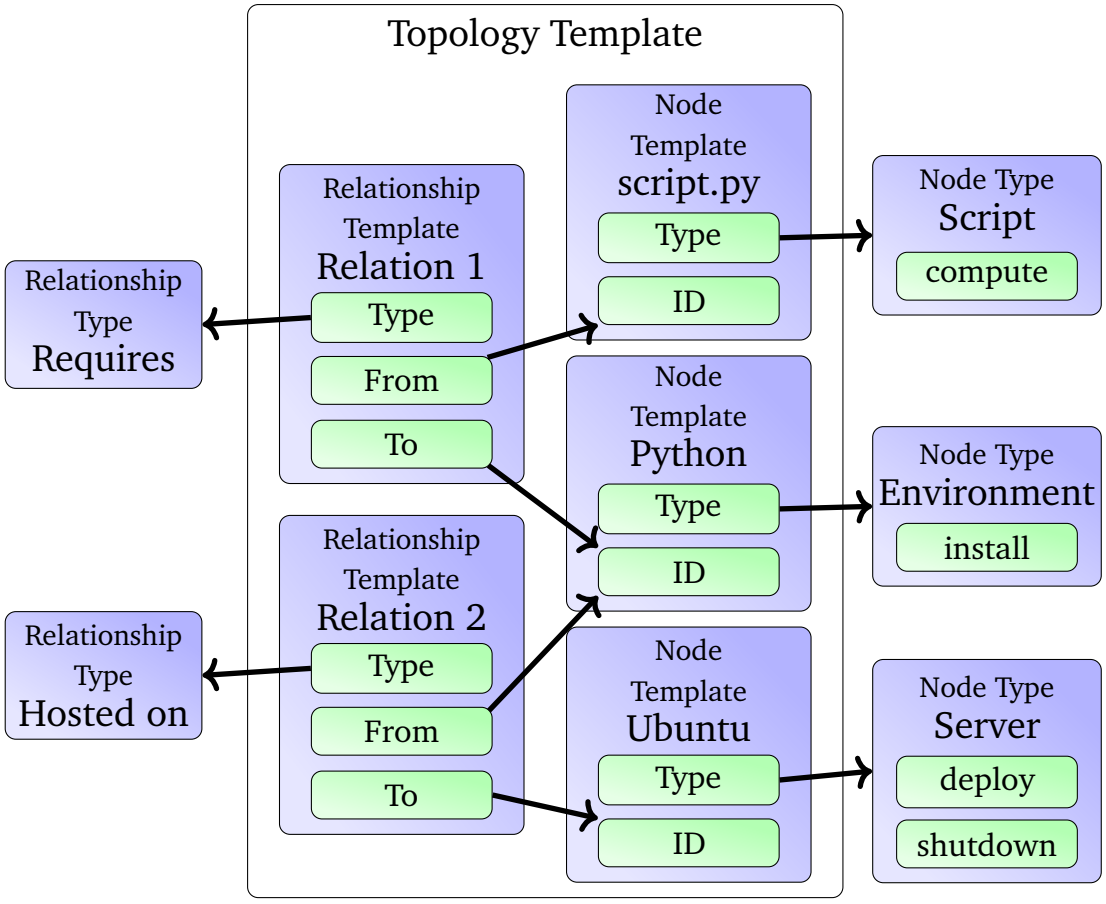

Figure 2.1: The example of the TOSCA application for weather calculation

of TOSCA application. They include definition documents, artifacts, etc. In this form, a TOSCA application can be processed by a TOSCA runtime environment.
The root folder of any CSAR must contain the "Definitions" and "TOSCA-Metadata" folders. The "Definitions" folder contains definition documents one of which must define a Service Template. The "TOSCA-Metadata" folder must contain TOSCA metadata in the form of a file with the "TOSCA.meta" name. This metafile consists of name/value pairs, one line for each pair. The first set of pairs describes the CSAR itself (TOSCA version, CSAR version, creator, etc). All other pairs represent metadata about files from the CSAR. The metadata is used by a TOSCA runtime environment to process the given files correctly.

2.3 OpenTOSCA

OpenTOSCA provides an open source ecosystem for TOSCA applications. This ecosystem consists of three parts: the TOSCA **runtime environment**, the graphical TOSCA modeling tool **Winery**, and the self-service portal for the applications available in the container **Vinothek** [IAA13]. Descriptions of the runtime environment and Winery will be provided in more detail in the following.

Runtime Environment

In this section, the OpenTOSCA runtime environment will be presented. The runtime environment serves as a container for CSARs and enables a fully automated plan-based deployment and management of TOSCA Applications presented by CSARs. It contains five main components: Controller, Implementation Artifact Engine, Plan Engine, Database and Container API. Container API provides an interface to communicate with Controller which starts plans, controls other components, tracks their progress, and interprets the TOSCA application. Implementation Artifact Engine is responsible to make implementation artifacts accessible for Plan Engine and execute them fully automatically. It has the plugin architecture which ensures extensibility. Implementation Artifacts are processed by the corresponding plugin which knows where and how to run this kind of artifact. The plugins deploy the respective artifacts for execution and save into Database their endpoints which can be called by Plans to start the artifacts. The deployment of Web Archives on Tomcat [Apa], Axis Archives on Apache Axis [Apa06] and many others is supported [Zim13]. The Plan Engine handles plans in the same manner. It is built according to a plugin architecture too and supports different workflow languages, e.g., **B**usiness **P**rocess **M**odel and **N**otation (BPMN) or **B**usiness **P**rocess **E**xecution **L**anguage (BPEL), and their runtime environments [BBH+13]. Plan Engine plugins deploy supported plans and save their endpoints into Database. Database stores all files Presented in a CSAR and endpoints of deployed artifacts and plans.
The processing of a CSAR is done in following manner. First, the CSAR is unpacked and its files are put into Database. Then, the TOSCA definitions documents are loaded, resolved, validated, and processed by Controller, which calls the Implementation Artifact Engine and the Plan Engine. The Implementation Artifact Engine deploys the referenced Implementation Artifacts and stores their endpoints in the Endpoints database. The Plan Engine binds and deploys the application's management plans. The endpoints of the management plans are stored into the Plans database too [BBH+13].

Winery

Winery provides a complete set of graphically functions for creation, edition and removal of elements of TOSCA topologies. It can operate with CSARs and consists of four parts: the type, template and artifact management, the topology modeler, the BPMN4TOSCA plan modeler [KBBL12], and the repository.
The type, template and artifact management component enables management of all TOSCA types, templates and related artifacts. This includes node types, relationship types, policy types, artifact types, artifact templates and artifacts. The topology modeler allows to create service templates which consist of node templates and relationship templates. They can be annotated with requirements and capabilities, properties, and policies. BPMN4TOSCA plan modeler offers web-based creation of BPMN models with the TOSCA extension. The modeler supports the BPMN elements and structures required by TOSCA plans and not the full set of BPMN. The Stardust project [Ecl] offers Browser Modeler, which covers all phases of the Business Process Lifecycle including modeling, simulation, execution and monitoring. In the context of Winery, this modeler was extended to support BPMN4TOSCA. The repository stores TOSCA models and allows managing their content. For instance, node types, policy types, and artifact templates are managed by the repository. The repository is also responsible for importing and exporting CSARs [Win]. An example of the TOSCA topology visualization is presented in Figure 2.2.

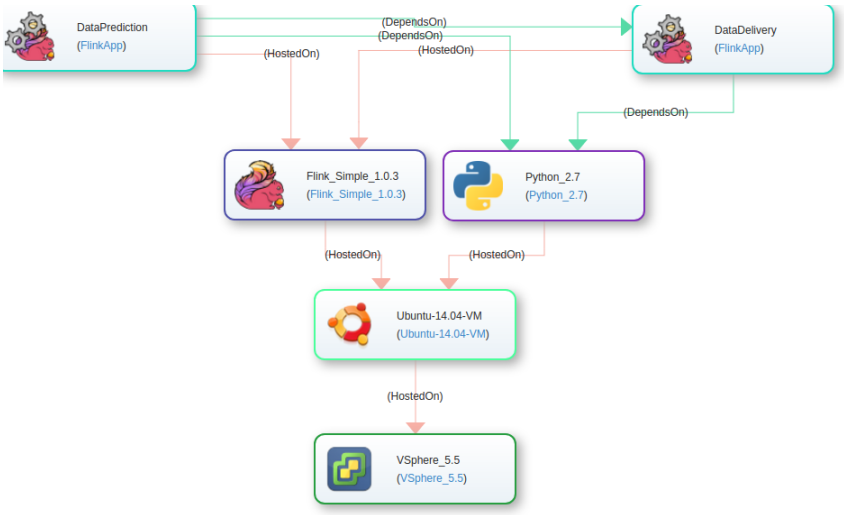

Figure 2.2: TOSCA Application visualized by $Winery$.

2.4 Package Management

The package management in Unix-like operating systems is described in this section. It handles the installation, update, configuration and remove of packages or in other word the management of packages. Packages are archive files containing both data for installation of the program components and their metadata like name, function, version, producer, and the list of dependencies to other packages [Chr07]. One package can present not only a complete program but also a certain component of a large application. Usually packages are stored in external web repositories which can be accessed by a user or a package manager to download the necessary packages. In this work, we are trying to eliminate usage of such external sources.
For a user, a package manager is a set of software tools which automate the management of packages. But from the operating system side, package managers are used to handle the database of packages, their dependencies and versions, and to prevent erroneous installation of programs and missing dependencies. This task is especially complex in computer systems relying on dynamic library linking. These systems share executable libraries of machine instructions across applications. Complex relationships between different packages requiring different versions of libraries result in a challenge colloquially known as "dependency hell" [Jan06]. Good package management is vital to these systems.
To give users more control over the kinds of programs that they allow to install on their systems, packages are often downloaded only from a number of software repositories. In Unix-like systems, package managers use official repositories appropriate for the operating system and the architecture of device where they operate, but it's possible to use additional repositories, like third-party repositories or repositories for another architecture.
Package managers distinguish between two types of dependencies: $required$ and $preRequired$. Dependency $package1$ $required$ $package2$ indicates that the $package2$ must be installed for a proper **operation** of the $package1$. Dependency $package1$ $preRequired$ $package2$ indicates that the $package2$ must be installed for a proper **installation** of the $package1$. In these examples, the $package2$ is needed for the $package1$, but the $package2$ itself can require additional packages. A structure describing all necessary packages and dependencies between them for the given root-package is called a dependency tree. The dependency type $required$ can lead to cycles in dependency trees and makes them different in comparison with the common tree graph structures.

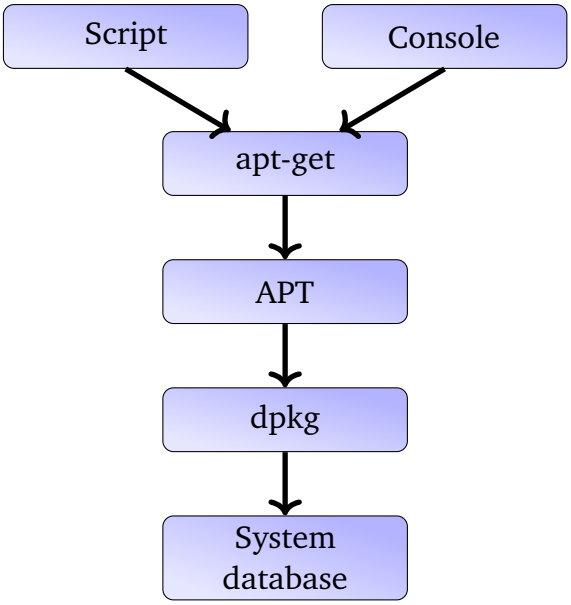

Figure 2.3: Scheme of apt-get execution.

Example Dependencies Handling

Its necessary to describe the principle of operation of common package managers to determine possible methods of encapsulation of their work. Therefore, the *apt-get* package manager will be considered to provide an example of execution. This application is part of **A**dvanced **P**ackaging **T**ool (APT) program which uses *dpkg* application to communicate with an operating system. The system keeps the database of packages and their conditions. These relations are presented in Figure 2.3.
apt-get has many functions: install, remove, update, autoremove, download, etc. We will consider the install, remove and autoremove operations to present the common algorithms of processing. When a package manager becomes a *package* installation command, it builds a dependencies tree for the *package* and checks the possibility to install the depended packages. For example, it must check the compatibility with previously installed packages. If the check was successful, the *apt-get* downloads and installs the packages from the dependencies tree. The *package* is marked in the database as manually installed and all the other packages are marked as automatically installed. It will be helpful during the autoremove operation when all automatically installed packages will be checked whether they are still needed. After installation of all packages from the dependencies tree, the *package* will be ready to work.
A *package* can be deleted by the *remove package* command. It happens only if there are no other packages depending on the *package*. If the deletion is very important, then these packages can be removed too to keep the consistency of the database. The packages necessary for the *package* itself will be deleted only by the autoremove command.

2.5 Configuration Management

In this section the $configuration\ management$ will be described. Configuration is the specific state of the operating system. It describes packages which must be installed, accounts that must be created, services that should be running, files which must be downloaded and other necessary characteristics. One can define all these setting in an configuration file which can be proceed by the corresponding configuration management tool to achieve specified state. During proceeding, configuration management tools can use package managers and data download utilities which handle external references. Data download utilities are used to get necessary data from remote sources. The data can be database images, other configuration files, etc. Common configuration management tools like Ansible, Chef and CFEngine will be presented. Additionally the Bash language and the Wget and cURL data download utilities will be described.

Ansible

$Ansible$ is an open-source automation engine. As with most configuration management software, Ansible has two types of servers: controlling machines and nodes. First, there is a single controlling machine which is where orchestration begins. Nodes are managed by a controlling machine over SSH. The controlling machine describes the location of nodes through its inventory. Ansible playbooks express configurations, deployment, and orchestration in Ansible. The playbook format is YAML. Each playbook maps a group of hosts to a set of roles. Each role is represented by calls to Ansible tasks [Ans16].

Chef

$Chef$ is a configuration management tool, which uses Ruby for writing system configuration files called "recipes". They describe how Chef manages applications and utilities and how they are to be configured. These recipes can be grouped together as a "cookbook" for easier management. Chef can run in client/server mode, or in a standalone configuration named "chef-solo". In client/server mode, the Chef client sends various attributes about the node to the Chef server. In solo mode the local system will be configured [Met15].

CFEngine

$CFEngine$ is an open source configuration management system. Its primary function is to provide automated configuration and maintenance of large-scale computer systems, including the unified management of servers, desktops, consumer and industrial devices, embedded networked devices, mobile smartphones, and tablet computers [Bur13]. Configurations are described by "policy" files, which are plain text-files with $.cf$ extension [CFE].

Bash

$Bash$ is a Unix command language written as a free software. It isn't a configuration management tool, but provides enough capabilities to be used as if it is. In addition Bash denotes a command processor that typically runs in a text window, where a user types commands that cause actions. Bash is examined because it is very popular since it is the default command line processor in Unix-like systems [RF17]. Executable files containing bash commands are called scripts. We are interesting in those scripts which are used to configure a system: to check environment, call package managers and data download utilities, etc.

Wget

GNU *Wget* is a utility designed for retrieving binary documents across the Web, through the use of HTTP and FTP. Wget is non-interactive, which means it can work in the background, while the user is not logged in [Xia99]. This tool is suitable for automatically tasks because it has built-in mechanism for the server response analyze. For example a user can setup it to repeat download a certain number of times in case of fault. Additionally Wget supports recursion mechanism which allows to download folders as simple as a single files.

cURL

$cURL$ is a command line tool for transfer the data. It is based on libcurl a free open source library which supports a big number of platforms and various protocols. Due to this portability, cURL is often used in small embedded devices [cUR97].

3 Requirements

In this chapter, requirements for the developed solution and its components resolving external references will be defined. Furthermore it will be described what we expect as a result of the work.
It's necessary to develop the concept of the new software which will resolve external references in CSARs and complement its topology to represent the changes made. A prototype must be implemented to validate the correctness of the concept. During this work, such terms as $input\ CSAR$ and $output\ CSAR$ will be used. The $input\ CSAR$ is the CSAR, which can contain external references and will be processed by the framework. The $output\ CSAR$ is the CSAR, which was processed by the framework and doesn't contain external references.
The application must proceed input CSARs and identify references to external packages and files. That includes a wide range of file download and package installation commands which use package managers to access package repositories and data download tools to get external files. To find such commands the software must be able to unpack the data and access all artifacts contained in input CSARs. Since one can't develop an application which support all possible types of external references, the structure of the application must ensure easy extensibility which allows to extend the software and handle any new types of references. Therefore the framework must be developed in the form of modular system so that each module will be able to identify and resolve the corresponding external references type. The three types of modules must be developed. One type of modules will be responsible for configuration management tools, the other type will process package managers and the last one will handle data download tools.
When an external reference is identified it must be resolved. In case of packages, that includes remove of package installation command and integration of the package installation data into the proceeded CSAR. To provide the encapsulation of the output CSAR, the package must be accompanied by all dependent packages which must be found, downloaded and added into the CSAR too. Then all downloaded packages must be integrated into the TOSCA topology. That means that one must define new nodes and relationships to map the dependencies from a package database into the topology. For file download commands, one must remove the command, download the file and integrate it into the CSAR structure. As a last step the processed TOSCA application must be archived back to get the output CSAR. The concept and architecture of the program

which satisfies the requirements will be described in chapter 4. The implementation of the prototype will be provided in chapter 5.

Result

As a result of the program's work, an output CSAR will be received. This CSAR must have the same functionality as the input CSAR, but all external references to additional packages or files must be resolved. The output CSAR must be able to be deployed properly without downloading these data via the Internet. In order to validate output CSARs, the TOSCA topology can be verified and the defined artifacts can be validated through their installations on a test machine.

Summarize

The requirements are summarized here. The developed framework must:

1. identify references to external packages and files in a input CSAR.
2. delete the external references.
3. integrate installation data and downloaded files into the CSAR to encapsulated it.
4. represent the changes into the TOSCA topology.
5. have an easy expandable structure.
6. generate an output CSAR with the same functionality as the input CSAR, but without external references.

4 Concept and Architecture

In this chapter, the concept and the architecture of the software which can satisfy the requirements, presented in chapter 3, will be explained and substantiated. The chosen solution is to build a modular framework, where each module will be responsible for identification and resolution of the specific external references type. During resolution, external data will be downloaded and integrated into the structure of the processed CSAR.

4.1 Concept

In this section, the main concept of this work are described. The general structure of the framework is visualized in Figure 4.1. Three types of modules will be introduces: $language\ modules$, $package\ manager\ modules$ and $download\ tool\ modules$. Language modules will handle configuration files written in the corresponding language. Package manager modules and download tool modules will proceed the package installation commands and data download commands respectively. They will identify and resolve external references and call the topology handler which will update the internal structure of TOSCA applications.
In section 4.1.1, it will be explained how to determine the Node Templates which use the given artifact. Then functionality of language modules, package manager modules and download tool modules will be presented. In section 4.1.5, it will be expressed how to create a new node for a TOSCA topology. After that, some methods to determine the architecture of the final platform will be presented.

4.1.1 Analysis of a TOSCA-Topology

To update the TOSCA topology properly, it is necessary to add references from the nodes where external references were to the newly created nodes which resolve the external references. For the given artifact with an external reference, one need to find all Node Templates which use this artifact, create for each of them new Node Templates resolving the reference and define the dependencies between them. The search can be done once for all artifacts during the preprocessing stage. Later one can use the results of the search

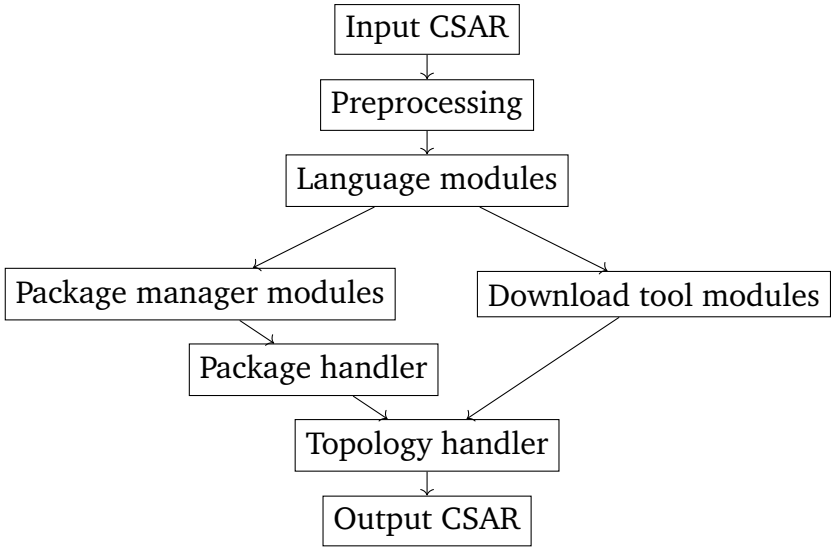

Figure 4.1: The general description of the software's work flow

to identify all Node Templates for the given artifact. Furthermore, Service Templates where these Node Templates are defined must be determined to create there new Node Templates and Relationship Templates. The pointers to artifacts are stored in Artifact Templates which are used by Node Type Implementations. Node Type Implementations implement Node Types wich are instantiated by Node Templates. By composing all the information the references chain can be built:
$Artifact \rightarrow Artifact\ Template \rightarrow Node\ Type\ Implementation \rightarrow Node\ Type \rightarrow Node\ Template \rightarrow Service\ Template$
Now consider the references in more detail.

- $Artifact \rightarrow Artifact\ Template$
  An Artifact can be referenced by several Artifact Templates. (Despite the fact that this is a bad practice.)
- $Artifact\ Template \rightarrow Node\ Type\ Implementation$
  The same way an Artifact Template can be used by several Node Type Implementations.
- $Node\ Type\ Implementation \rightarrow Node\ Type$
  A Node Type Implementation can describe an implementation of only one Node Type.
- $Node\ Type \rightarrow Node\ Template$
  Each Node Type can have any number of Node Templates.
- $Node\ Template \rightarrow Service\ Template$
  But each Node Template is defined only once.

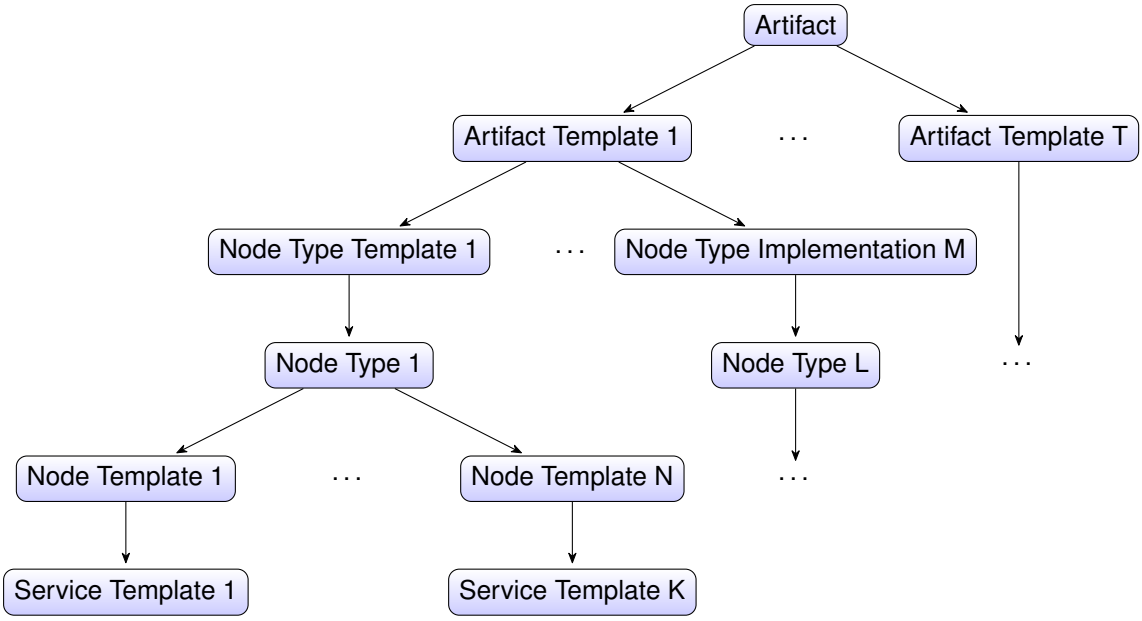

Figure 4.2: An example tree describing how to find Service Templates and Node Templates for the given script

This structure can be described as a tree with an Artifact as a root and Service Templates as leaves, and will be called the internal dependencies tree. The example is provided by Figure 4.2.
There is an additional problem in references between Node Types and Node Type Implementations. A Node Type can have several implementations, but which one will be used is determined only during the deployment. The chosen solution to this problem is to use each Node Type Implementation in the hope, that they will not conflict.
The following steps to build the internal dependencies tree can be executed during the preprocessing.

- Find all Artifact Templates to build references from Artifacts to Artifact Templates.
- Find all Node Type Implementations. Since they contain references both to the Node Type and to the Artifact Templates, the dependency from Artifacts to Node Types can be built.
- Find all Service Templates and all Node Templates they contain. Each Node Template refers to its Node Type what is useful for building a dependency from Artifact to Node Template.

In this way the required internal dependencies tree with references
$Artifact \rightarrow Node\ Template$ and $Artifact \rightarrow Service\ Template$ can be built.

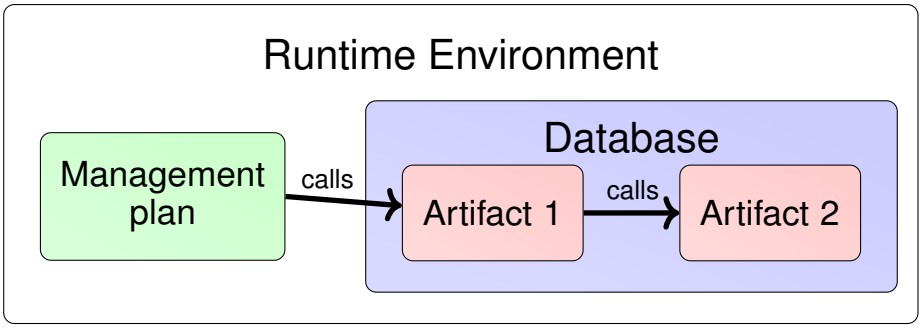

Figure 4.3: Bad artifacts call sequence

4.1.2 Search for Artifacts

Since artifacts may contain external references, we need to find all of them in order to resolve the references. The first simple solution is to analyze the structure of a TOSCA application and to identify all Artifact Templates and corresponding artifacts. But this method brings possibility to miss artifacts because some of them can be called from other artifacts and may be not presented in the TOSCA topology by an Artifact Template. This case is presented by Figure 4.3. In this example the "Artifact 1" was deployed into Database by Implementation Artifact Engine and is called by a management plan. The "Artifact 2" is an executable file without an Artifact Template. That is a bad practice, but we must consider that option too, to provide higher level of reliability. It is possible, that the "Artifact 2" will not be considered by the method described above. The found solution is to analyze all files presented in input CSARs and resolve external references in all of them. One needs to define methods to identify configuration files for each supported configuration management tool.

4.1.3 Modules and Extensibility

The framework should handle different configuration management tools, each of which can use various package managers and data download tools. It was decided to develop a modular system, where modules handle the above listed elements with the exception of configuration management tools. Since configuration files can have many forms, like Ansible playbook or Bash script, it will be created one unified abstract type of modules - language modules which will handle specific languages used to write configuration files. Each language module itself contain and support package managers modules and download tool modules which handle package managers and data download tools respectively. This principle can be illustrated by Figure 4.4. Language modules should filter out files not belonging to the language and the accepted files will be transmitted to the supported modules. Package manager modules resolve external references and

Listing 4.1 Unreadable bash script

```
#!/bin/bash
set line = abcdefgijklmnoprst
# The "line" contains a part of the alphabet
set word1 = ${line:0:1}${line:14:1}${line:17:1}
# The 1th, 15th and 18th letters of the "line" variable are stored into the "word1".
# "word1" will contain the "apt" string
set word2 = ${line:6:1}${line:4:1}${line:17:1}
# The 7th, 5th and 18th letters of the "line" variable are stored into the "word2".
# "word2" will contain the "get" string
$word1-$word2 install package
# This is the "apt-get install package" command,
#            but to determine that a good interpreter is needed.
```

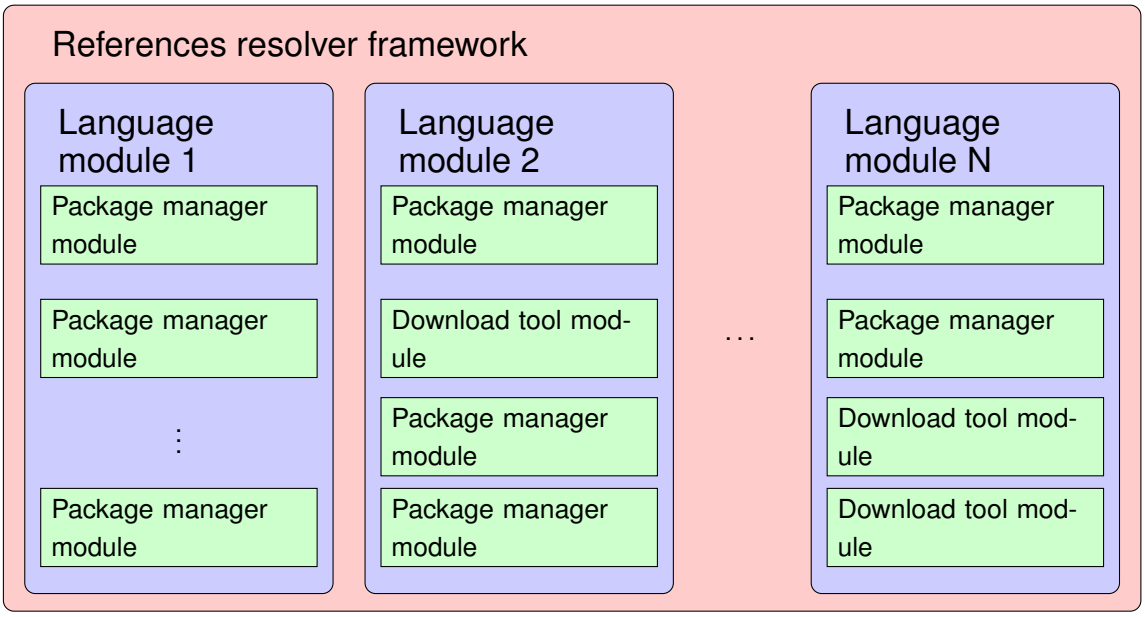

Figure 4.4: An example scheme representing several language modules containing package manager modules

transmits the names of packages from these references to a package handler described in section 4.2.3. Download tool modules identifies download commands, removes them, downloads the data and calls a topology handler defined in section 4.2.4.
It is impossible to identify all types of external references, even when only one language and one package manager are used. An example can be found in listing 4.1. Since this work is aimed at creating the easily expanded and supplemented tool, then only basic usage of package managers will be considered initially.

Encapsulation of CSARs with extarnal Packages

Here will be defined some methods of encapsulation of a CSAR containing external packages. At first, one will describe the $generate\ custom\ repositories$ and $generate\ shared\ repositories$ methods not representing packages in a TOSCA topology and then the $one\ node\ for\ one\ package$ and $sets\ of\ packages$ methods mirroring packages into the topology.

Generate Custom Repositories

It's possible to download all necessary packages and create one's own custom package repository for each operating system used in the application. Then one must rework any package installation commands or exchange system preferences to setup an access to the custom repository. This method introduces minimal changes into a structure of TOSCA application. The main problem is the creation of the custom repositories. When a TOSCA application consists of many operating systems executing on devices with limited capabilities, it may be difficult to start custom repositories on each of them.

Generate Shared Repository

Another opportunity is to create a single repository for all operating systems in a TOSCA application. It can be difficult to choose the right location for such a server, but since the applications often represents connected systems, this step can redistribute the load to a more powerful device. It is difficult to estimate the changes which will occur in a TOSCA topology while applying such a method.

One Node for One Package

A new TOSCA node will be created for each downloaded package. All dependencies between packages will be mirrored to a TOSCA topology. This is a very visual method, facilitating the understanding of dependencies between packages.

Sets of Packages

The set of depended packages from a dependencies tree related to an external reference can be combined and represented in a TOSCA topology as a single node. An installation of such a node will lead to the installation of all needed packages. This way a small size of a TOSCA application's structure will be achieved. It will be impossible to trace dependencies (since many packages are represented by one node), but it can help to avoid a difficult structure which consists of hundreds of nodes.

4.1.4 Encapsulation of CSARs with extarnal files

It was developed two modes to handle download commands in CSARs: $addition$ and $integration$. They will be presented shortly.

Addition

The method is to add a new TOSCA node for each downloaded file. This node will contain a deployment artifact presenting the downloaded file and an implementation artifact which will try to put the file into the right place. Additionally a reference between the old node with an external reference and the added node must be created.

Integration

We can't be sure that the file added by the method described above will be available for other nodes. Sometimes it can be difficult to get right position for the downloaded file. We need to provide another opportunity for a user of the framework. In the integration mode, the downloaded file must be integrated into the node which contain the file download command. The command can be exchanged by a file move command to put the file into the right position.

4.1.5 Representing Downloaded Packages in a TOSCA-Topology

Using some developed methods like Addition for external files or Sets of packages for external packages, downloaded data must be represented in the structure of a TOSCA Application. We will introduce two new terms which can facilitate understanding: a package node and a file node. These nodes denote to the defined and instantiated TOSCA node, the purpose of which is to install packages or store files respectively. Defined means, that the corresponding definitions of Node Type, Node Type Implementation, Artifact Types and Artifact Templates are integrated into the processed CSAR. And to instantiate the nodes one need to create a Node Template. The addition of new package and file nodes to the TOSCA topology can be divided into several steps.

- One must add definitions for common elements like Artifact Types or Relationship Types. This can be done once.
- The common definition of the created node will be represented by a Node Type. It must contain the $install$ operation, which represents the capability to install the node. For the package nodes that will cause the installation of the package and the file move operation for the file nodes.
- Artifacts (downloaded data and configuration files) will be referenced by Artifact Templates.

- A Node Type Implementation will combine the Artifact Templates to implement the *install* operation.
- A Node Template will instantiate the node in the corresponding Service Templates. To determine the corresponding Service Template the author uses the method described in section 4.1.1.
- A Reference Template will provide topology information allowing the observer (a user or a runtime environment) to determine dependencies between nodes. References will connect Node Templates contained external references and Node Templates resolving external references.

After an execution of these steps, the definition of a new TOSCA node will be finished and it can be used.

4.1.6 Determining Architecture of the Final Platform

To download the right instance of a package one must define its architecture. But it is impossible to analyze the structure of any CSAR and determine the architecture of the device which the nodes will be deployed on. There are many pitfalls here.
A single Service Template can use several physical devices with different architectures. Many Node Types and Node Templates instantiated on different platforms can refer to the same Implementation Artifact. This way one simple Implementation Artifact, for example with a bash script containing the "*apt-get install python*" command, can be executed by different Node Templates on different devices (for example with the arm, amd64 and i386 architectures) and will result in the download and installation of three different packages. For an end user, the ability to use such a simple command is a huge advantage, but for the framework, it can greatly complicate the analysis. The following methods of architecture selection were designed.

- *Deployment environment analysis*
  All instances of packages for all supported architectures will be integrated into the processed CSAR. The right instance will be chosen during deployment.
- *Unified architecture*
  The architecture will be defined for the whole CSAR.
- *Artifact specific architecture*
  The architecture will be defined for each artifact separately.

The *deployment environment analysis*, which at first sight seems to be the most reliable solution, brings many additional problems. Packages for different platforms can differ not only by architecture but also by the version and the list of dependencies. Chaos may

occur when using methods mirroring dependencies between packages into a TOSCA topology. The only found robust solution is to use an extended variant of the Sets of packages method described in section 4.1.3. Using this method, all the data needed for installation of a package for one architecture will be stored in one set. Framework must create a set of installation data for each supported architecture. The right set will be chosen during the deployment of the CSAR. The $artifact\ specific\ architecture$ method carries an additional complexity to the framework. One must analyze each artifact and decide which architecture it will be executed on. This can be complicated by the fact that the same artifact can be executed on different architectures. The $unified\ architecture$ method was chosen as the simplest and easiest to implement. If it will be necessary, this method can easily be expanded to $artifact\ specific\ architectures$ or to $deployment\ environment\ analysis$.

4.2 Architecture

This section will present the architecture of the framework and the detailed description of its elements. The main elements are a ***CSAR handler***, a ***references resolver***, ***language modules***, ***package manager modules***, ***download tool modules***, a ***package handler***, and a ***topology handler***.

4.2.1 CSAR handler

The CSAR handler provides an access to a CSAR and maintains its consistency. It defines methods to process the metadata when adding new files, to archive/unarchive the CSAR, and to choose the final platform architecture.
The input CSAR is initially archived and must be decompressed in order to handle the content first. When all external references will be resolved, the content will be archived to an output CSAR. A new name-value pair must be added to the metadata for each new file integrated into the CSAR during processing. The name represents the internal path to the file and the value contains the type of the file. This type will be used by a runtime environment to choose the right behavior. As it was mentioned in section 4.1.6, the architecture of the final platform will be chosen for the entirely CSAR. This can be done once at start. The chosen architecture must be saved to the CSAR for the case of future processing by the framework to avoid the collisions between architectures of packages.

4.2.2 References Resolver

References Resolver is the main element, whose execution is divided into three stages: $preprocessing$, $processing$, $finishing$. The first stage is the preprocessing. CSAR is an archive file and it must be decompressed in order to manipulate the content. The CSAR handler can be used for this task. As it was mentioned in section 4.1.5, one need to add common definitions for used Artifact Types and References Types. This will be done during the preprocessing stage. Other important task is the generation of internal dependencies trees, which were described in section 4.1.1. After these operations starts the $processing$ stage. During this stage all $language\ modules$ will be activated. Their operation is described in more detail in the next section. To finish the work all results will be packed into the output CSAR during the $finishing$ stage. It's also necessary to update metadata of the CSAR to display the files added.

Language Modules

Each $language\ module$ describes a handling of one language used to write configuration files. This module must accept only the files written in the language. It also contains a list of supported package manager modules and download tool modules. Each language module must provide the capability for the given package to generate a TOSCA node which must use the language to install the package. For example a Bash module must provide capability to define new package nodes which use Bash to install the packages. This means that a configuration file and definitions for Artifact Templates, a Node Type, and a Node Type Implementation should be created by a language module.
As it was already mentioned above,a language module analyzes all files one by one and checks their belonging to the language during the $processing$ stage. Any files not belonging to the described language are filtered out. The remaining files are transferred to the language module's $package\ manager\ modules$ and $download\ tool\ modules$. Some modules can have an additional functionality. For example an $ansible$ module should provide capability to unpack zip archives where ansible playbooks can be stored.

Package Manager Modules

These modules process specified package managers. They are integrated into language modules and become from them files to analyze. To identify an external reference a package manager module will parse the given files. They must identify package installation commands which use the handled package manager. Package manager modules find external references, remove them and transmit the package names to the $package\ handler$ which is described in section 4.2.3 and returns the list of all required packages back to the package manager module which will transfer them farther back to a language module. Figure 4.5 illustrates data flow between language modules, package manager modules and the package handler.

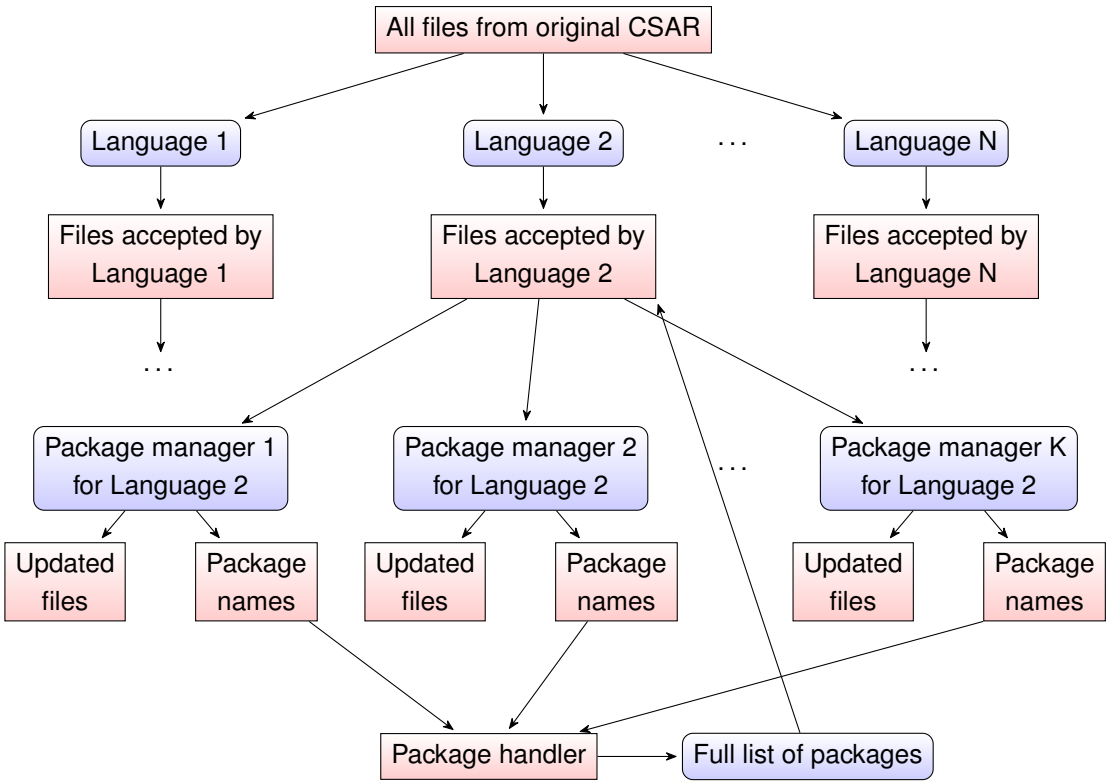

Figure 4.5: The data flow scheme between language modules, package manager modules and the package handler.

An example will be provided for the *apt-get* module for the Bash language. The *apt-get* package manager was described in section 2.4. The module will read the given file line-by-line searching for the commands starting with the "apt-get install" string. Such commands use the *apt-get* package manager to install packages from external sources and must be removed from the file. The arguments of this command is the list of package names which need to be installed. These names must be transferred to the package handler.

Download Tool Modules

Each download tool module process the defined data download tool. It analysis the files provided by a language module which it was defined in. Identified external references must be resolved by removing installation commands, downloading files and integrating them into the processed CSAR. Behavior features will be defined by the selected operation mode.

4.2.3 Package Handler

The $package\ handler$ provides interface to communicate with an operating system's package manager. It can download installation data, determine the type of dependency between packages and provide a list with dependent packages for the given package.
To download the installation data this component will use the given package name and the architecture specified by the CSAR handler. Then it transfers the package name to the $topology\ handler$ and repeats these actions for all depended packages recursively. During the process it stores all names of packages from the dependencies tree of original package and sends them back to the calling package manager module. If the $apt\text{-}get$ package manager is used the command "$apt\text{-}get\ download$ **package**" can be used to download the data. The architecture can be specified by a ":$architecture$" suffix, for example, a "$package$:arm" mean the package for the arm architecture. The list of dependencies will be obtained using an operating system package analyzer. Its output should be parsed in order to extract the names of depended packages. Type of dependency can be achieved in the same manner. Of course, in case of a fault during a download of a package, a user interface should be provided to find a solution. For example it can be: retry the download, ignore the package, rename the package or even break the framework's execution.

4.2.4 Topology Handler

This element handles the TOSCA topology and it has two main tasks: create internal dependencies trees and generate TOSCA definitions for packages provided by the package handler. To build the trees the analysis of the TOSCA topology will be used during the preprocessing stage. This procedure was described in section 4.1.1. The needed definitions for instantiation of a new TOSCA node include Node Templates and Relationship Templates which were described in section 4.1.5. To create the definitions in the right places the generated internal dependencies trees will be used. The internal dependencies trees must be updated to represent changes after addition of new Node Templates and Relationship Templates.

5 Implementation

This chapter provides an information about the implementation of the prototype and its elements, whose behavior was described in chapter 4. The application can resolve external references presented by package installation commands. It handles the Bash language with apt-get package manager and the Ansible configuration management tool with the apt package manager. The developed software can operate in two modes: One node for one package and Sets of packages.
Java language was chosen for implementation, because of its simplicity and strength. To compile the framework a Java Development Kit version 1.8 or above is needed. If the software is already compiled, an Java Runtime Environment version 1.8 or above must be installed in order to execute the application. Additionally, the apt-get package manager which functions only in Unix-like operating systems must be installed to download packages and identify dependencies between them. If a user wants to download packages for the specific architecture the package manager must be setup to access this architecture's repository.

5.1 Global Elements

This section describes the elements used throughout the whole framework's execution. The ZIP handler provides a functionality to operate ZIP archives, the CSAR handler keeps an interface to interact with a CSAR and Utils helps to solve problems common for many other elements.

Zip Handler

This is a small element with straight functionality. It serves to pack and unpack ZIP archives which are used by the TOSCA standard to pack applications. It was decided to use the $java.utils.zip$ package for this task. The functions for archiving and unarchiving are called $zipIt$ and $unZipIt$ respectively.

CSAR Handler

This element provides an interface to access the content of a CSAR and stores information about files associated with it. The mostly used data are: the name of a temporary extraction folder, the list of files from the input CSAR, the meta-file entry, and the architecture of the target platform. All this data are encapsulated into the CSAR handler. The set of public functions allowing to operate with this element is available.

- $unpack$ and $pack$ functions are used to extract the CSAR into the temporary folder and pack the folder to the output CSAR. These functions use the $ZIP\ handler$.
- $getFiles$ returns the list with files presented by the input CSAR.
- $getFolder$ returns the path to the folder which the CSAR was extracted to.
- $getArchitecture$ returns the chosen architecture of the target platform.
- $addFileToMeta$ adds information about the new file to the meta-data.

Here is an example usage of the element. When the CSAR handler extracts the input CSAR to the temporary extraction folder during the $unpack$'s call, it saves the folder's name. Then other elements can use the $getFolder$ function to get this name and access the data.

Utils

This class provides the $createFile$, $getPathLength$, and $correctName$ methods used by many other elements. The main purpose of these functions is to make the code cleaner. Using the $createFile$ function other elements of the framework can create a file with the given content. The $getPathLength$ method returns the deep of the given file's path what is very useful for creating references between files. OpenTOSCA uses some limitations to names of TOSCA nodes. Those names can't contain slashes, dots, etc. The function $correctName$ can be used to obtain an acceptable name from the given string.

5.2 References Resolver

This is the main module which starts by framework startup and is executed into three stages: preprocessing, processing and finishing which were described in section 4.2.2. In this section, some implementation aspects will be presented briefly.

Preprocessing

At the preprocessing stage, the CSAR is unpacked, common TOSCA definitions are generated and internal dependencies trees are built. As the first step, a user interface is provided to get the names of the input CSAR, output CSAR, mode of operation and the architecture of the final platform. To unpack the CSAR the function $unpack$ from the CSAR handler is used.
The $javax.xml.bind$ package was chosen for creating the common TOSCA definition. This Java package allows to generate $Descriptions$ - Java classes describing an XML documents which store TOSCA definitions. The following $Descriptions$ where created:

- $DependsOn$ and $PreDependsOn$ defines Relationship Types which determine dependencies between packages.
- $Package\ Artifact$ describes a deployment Artifact Type for package installation data.
- $Script\ Artifact$ specify an implementation Artifact Type for a script installing a package.
- $Ansible\ Playbook$ represent a implementation Artifact Type for a package installation via an Ansible playbook.

An example of $Description$ of the Script Artifact can be found in listing 5.1. To build internal dependencies trees the topology handler described in section 5.6 was used.

Processing

During this stage, all language modules listed in the framework are started. For the references resolver element that is only two following strings of code, but they start the main functionality of the framework.

```
\\All languages are stored in the "languages" variable
for (Language language : languages)
      language.proceed();
```

The language modules check all files presented in the input CSAR. The list of these files is stored in the CSAR handler, a pointer to which the modules became, store and translate to the supported package manager modules during their instantiation. This system allows the modules to access the CSAR's content at any time.

Listing 5.1 Java class containing the $Description$ for the Script Artifact Type definition

```
public class RR_ScriptArtifactType {
      @XmlRootElement(name = "tosca:Definitions")
      @XmlAccessorType(XmlAccessType.PUBLIC_MEMBER)
      public static class Definitions {
            @XmlElement(name = "tosca:ArtifactType", required = true)
            public ArtifactType artifactType;
            @XmlAttribute(name = "xmlns:tosca", required = true)
            public static final String
                tosca="http://docs.oasis-open.org/tosca/ns/2011/12";
            @XmlAttribute(name = "xmlns:winery", required = true)
            public static final String winery =
                  "http://www.opentosca.org/winery/extensions/tosca/2013/02/12";
            @XmlAttribute(name = "xmlns:ns0", required = true)
            public static final String
                ns0="http://www.eclipse.org/winery/model/selfservice";
            @XmlAttribute(name = "id", required = true)
            public static final String id="winery-defs-for_tbt-RR_ScriptArtifact";
            @XmlAttribute(name = "targetNamespace", required = true)
            public static final String targetNamespace =
                  "http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes";

            public Definitions() {
                  artifactType = new ArtifactType();
            }

            public static class ArtifactType {
                  @XmlAttribute(name = "name", required = true)
                  public static final String name = "RR_ScriptArtifact";
                  @XmlAttribute(name = "targetNamespace", required = true)
                  public static final String targetNamespace =
                        "http://docs.oasis-open.org/tosca/ns/2011/12/ToscaBaseTypes";
                  ArtifactType() {}
            }
      }
}
```

Finishing

When all external references will be resolved, the framework can enter its last stage. At this stage, the changed data should be packed into the output CSAR, whose name was entered during the preprocessing stage. The function $pack$ from the CSAR handler is used. After this operation, one receives a more encapsulated CSAR which implements the requirements, presented in section 3, and has lower level of access to the Internet.

5.3 Language Modules

This section will describe the implemented language modules. Since the framework is initially oriented to easy extensibility, an abstract model for the modules will be defined in such way, that new modules can be added by implementing this model. The realization of the Bash and Ansible modules will be provided at the end of the section.

Language Model

To specify the common functionality and behavior of different language modules, the language model is used. In Java, this model is described by an abstract class. The abstract class $Language$ is presented in listing 5.2. The common variables for all language modules are the name of the language, the list with package manager modules, and the extensions of files. The common functions are presented below.

- $getName$ returns the name of this language.
- $getExtensions$ returns the list of file extensions for this language.
- $proceed$ checks all original files. Files written in the language should be transferred to every supported package manager module.
- $getNodeName$ uses a package name to generate the name for a Node Type, which will install the package using this language.
- $createTOSCA\_Node$ creates the definitions for a TOSCA node. Since the created TOSCA nodes must install packages using the same language as the original node, all languages must provide the method for the creation such definitions.

The $createTOSCA\_Node$ function must be implemented in two variants. The first option is to accept a single package as an argument, which is needed for the One node for one package mode. The second implementation is needed for the Sets of packages mode where a lot of packages can by installed by one node. In second variant, this function must take a set of packages as an argument. New language modules must be inherited from the language model and then can be added to the framework.

Bash Module Implementation

The processing of the popular Bash language was implemented in the prototype. The Bash module should accept only files written in the Bash language. All the files presented in Artifact Templates with a Bash script type will be accepted. Additionally the module

Listing 5.2 Abstract language model

```
public abstract class Language {

	// List of package managers supported by language
	protected List<PacketManager> packetManagers;

	// Extensions for this language
	protected List<String> extensions;

	// Language Name
	protected String Name;

	// To access package topology
	protected Control_references cr;

	// List with already created packages
	protected List <String> created_packages;

	/**   Generate node name for specific packages
	* @param packet
	* @param source
	* @return
	*/
	public abstract String getNodeName(String packet, String source);

	/**   Generate Node for TOSCA Topology
	* @param packet
	* @param source
	* @return
	* @throws IOException
	* @throws JAXBException
	*/
	public abstract String createTOSCA_Node(String packet, String source)
		throws IOException, JAXBException;
	public abstract String createTOSCA_Node(List<String> packages, String source)
		throws IOException, JAXBException;
}
```

will accept files not presented in Artifact Templates at all, but possessing specific signs. These signs can be the file extensions (".sh" or ".bash") and the first line ("#!/bin/bash"). Each file which contains those signs will be passed to supported package managers modules, in our case to the $apt\text{-}get$ module described later.
The Bash module must provide a capability for the given packages to create a definitions of package nodes which use Bash to install the packages. Such a Bash package node is defined by Package Type, Implementation, Package Artifact and Script Artifact. Package Type is a Node Type with the "$install$" operation and a name received from the $getNodeName$ function. Implementation is a Node Type Implementation which refers the Package Artifact and the Script Artifact to implement the operation. Package Artifact and the Script Artifact are Artifact Templates referencing the installation data and a Bash installation script respectively. The installation script contains the Bash header and an installation command, like "$dpkg$ *-i* **installation_data**". The topology handler will instantiate the package node by defining a Node Template. Those definitions and the installation script are created by the $createTOSCA\_Node$ function.

Ansible Implementation

Ansible configuration management tool was added to validate the extensibility of the framework. Since Ansible playbooks are often packed into archives, it may be necessary to unpack them first and then to analyze the content. Thus, the files are either immediately transferred to the package manager modules, or they are unzipped first. To filter Ansible files not represented by Artifact Templates, the "$.yml$" extension is used, because Ansible playbooks don't contain any specific header.
Creation of an Ansible TOSCA node for a package is a complicated operation, because one must define the configuration of the created Ansible playbook and to archive the configuration, playbook and installation data. As the first step, the original files should be analyzed to determine the Ansible configuration (the set of options like a user name or a proxy server). If the implemented analyzer is unable to find all necessary options, a user interface will be provided to fulfill any missing parameters. After that, the Ansible playbook installing the package and the configuration file describing the playbook will be created in a temporary folder. After addition of installation data into the folder, it can be packed to a zip archive. This archive is an implementation artifact, which the Artifact Template should be created for. A Node Type with an "$install$" operation should be defined. And finally, a Node Type Implementation linking the operation and the Artifact Templates should be generated. A Node Template will be added by the topology handler.

5.4 Package Manager Modules

In this section, package manager modules will be specified. The main task of this modules is to identify external references, delete them and call the package handler. An abstract model will be defined to make the extensibility easier. The apt-get module for Bash and an apt module for Ansible will be implemented.

Package Manager Model

The model is described by an abstract class. It contains only one function $proceed$ that finds and eliminates external references, as well as passes the found package names to the package handler and return the list of all required packages.

Apt-get for Bash

The apt-get package manager module is a simple line-by-line file parser which searches for the lines starting with the "$apt\text{-}get\ install$" string, comments them out and passes the command's arguments to the package handler's public function $getPackage$.

Apt for Ansible

Since Ansible package installation commands which use the apt package manager can be written in many different ways, then the processing will be more difficult than a simple line parser. Therefore, to handle Ansible playbook a state machine and regular expressions from the $java.util.regex$ package are used.

5.5 Package Handler

Package handler provides an interface for an interaction with the package manager of the operating system. It allows to $download\ packages$, to $determine\ the\ type\ of\ dependencies$ between them and to $obtain\ the\ list\ with\ dependent\ packages$ for the given package.

Download Packages

The download operation is performed using the recursive function $getPackage$. The arguments of the function are described shortly. $language$ is a reference to the language module which has accepted the original artifact. $packagename$ is a name of the package. $listed$ holds a list with already downloaded packages. It is not necessary to download them again, but new dependencies must be created. $source$ defines the parent element of the package. It will be the original artifact file for the root package, and the depending package for other packages. $sourcefile$ is a name of the original artifact.
The command $apt\text{-}get\ download$ **packagename** is used for download the package. If the process fails, a user input is provided to solve the problem. A user will be able to rename the package, ignore it or even break the processing. If dependent packages are available, the function calls itself recursively for each dependent package. After these operations, a dependencies three for the $packagename$ will be downloaded.
In the One node for one package mode, the function calls the language's function $createTOSCA\_Node$ using the $language$ variable to create the TOSCA node for the package. Then it calls the topology handler's functions $addDependencyToPacket$ or $addDependencyToArtifact$ to update the topology.

Obtain List with Dependent Packages

To obtain the dependent packages for the given package the $getDependensies$ function was developed. It becomes a *packagename* as an argument and uses the command $apt\text{-}cache\ depends$ *packagename* to build a list with dependencies for the *package*. The $apt\text{-}cache$ command is a part of the $apt\text{-}get$ package manager and uses a packages database to print the dependencies. The output is parsed to find strings like "Depends: *dependent_package*". These dependent packages are combined to a list and returned back.

Determine Type of Dependency

To determine the type of dependency between two packages the $getDependencyType$ function is used. It becomes the names of the source package and the target package and uses the $apt\text{-}cache\ depends$ command to get the type. It can be $Depends$, $preDepends$ or $noDepends$ dependency.

5.6 Topology Handling

The topology handler serves to update the TOSCA topology. It builds the internal dependencies trees during the preprocessing stage. The trees are used to find the right places for definitions of Node Templates and Dependency Templates.

Building Internal Dependencies Trees

At the preprocessing stage, this element analyzes all original definitions and constructs internal dependencies trees. To read those definitions from the XML files the package $org.w3c.dom$ was used.
As the first step, all definitions of Artifact Templates are analyzed and pairs consist of an Artifact Template's ID and an artifact itself are built. Then each Node Type Implementation will be read and Node Types and Artifact Template's IDs found. Now each artifact has a set with Node Types where it is used. After the analysis of Service Templates, analog sets of Node Templates for each artifact will be created. In addition, for each Node Template one should keep the Service Templates, where this Node Template was defined.

Updating Service Templates

To update Service Templates two functions are provided.

- $addDependencyToPackage(sourcePackage, targetPackage, dependencyType)$ generates a dependency between two package nodes.
- $addDependencyToArtifact(sourceArtifact, targetPackage)$ generates a dependency between an original node and a package node.

Both functions find all Node Templates which use the given $sourcePackae$ or $sourceArtifact$. Besides, they find Service Templates where the Node Templates are defined. The search is done with the help of the internal dependencies trees. For each found Node Template a package node for installation of the $targetPackage$ package should be instantiated by creating a new Node Template. Then the dependency between the found Node Template and the new Node Template is created by defining a Relationship Template. The type of dependency is the value of the $dependencyType$ for the $addDependencyToPackage$ function and the $preDependsOn$ for $addDependencyToArtifact$. To update the existing TOSCA definition the $org.w3c.dom$ and $org.xml.sax$ packages are used. The definition of a new Node Template for the given $topology$ and $package$ is presented in listing 5.3.

Listing 5.3 Creating of a new Node Template

```
Element template = document.createElement("tosca_ns:NodeTemplate");
template.setAttribute("xmlns:RRnt",
    RR_NodeType.Definitions.NodeType.targetNamespace);
template.setAttribute("id", getID(package));
template.setAttribute("name", package);
template.setAttribute("type", "RRnt:" + RR_NodeType.getTypeName(package));
topology.appendChild(template);
```

6 Example Integration of new Module

This chapter shows the extensibility of the framework by adding a new *aptitude* package manager module for the Bash language. Section 6.1 provides common information about *aptitude*. In section 6.2 the module is developed and integrated into the Bash language module.

6.1 Aptitude Package Manager

This section describes the *aptitude* package manager. Like to *apt-get*, *aptitude* is a command line program, where a package can be installed using the *aptitude install package* command. In additional, it can be started in a pseudo-graphical mode to provide a visual interface shown in Figure 6.1). An another advantage compared to *apt-get* is the capability to search for packages by a part of the name (or by any other attributes) using the *aptitude search text* command.

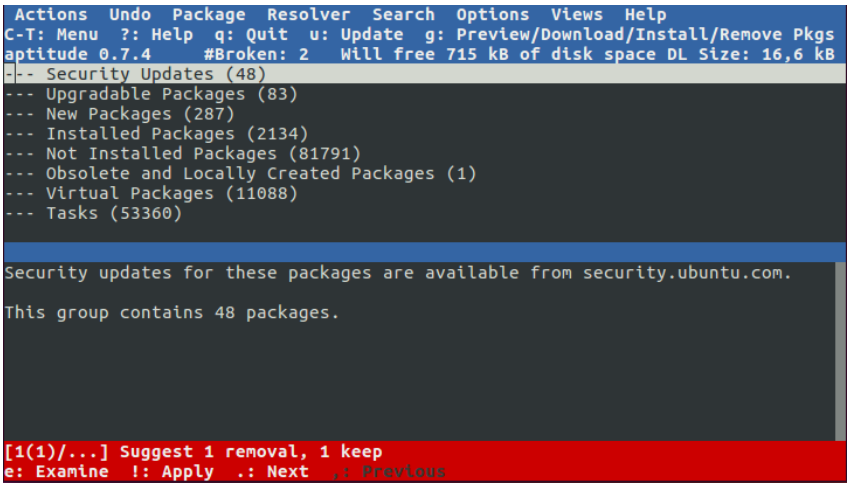

Figure 6.1: The command line visual interface for the *aptitude* package manager.

Listing 6.1 The $aptitude$ module inherited from the $PackageManager$ abstract class

```
public final class PM_aptitude extends PackageManager {
	@Override
	public List<String> proceed(String filename, String source)
		throws FileNotFoundException, IOException, JAXBException {
		// TODO Auto-generated method stub
	}
}
```

Listing 6.2 The $aptitude$ module with some common elements

```
public final class PM_aptitude extends PackageManager {

	// name of the package manager
	static public final String Name = "aptitude";

	/**
	* Constructor
	*/
	public PM_aptitude(Language language, CSAR_handler ch) {
		this.language = language;
		this.ch = ch;
	}

	@Override
	public List<String> proceed(String filename, String source)
		throws FileNotFoundException, IOException, JAXBException {
		// TODO Auto-generated method stub
	}
}
```

6.2 Development of Aptitude Module

The implementation of the $aptitude$ module will be described here. At first, the $aptitude$ class will be inherited from the abstract class $PackageManager$. This is presented in listing 6.1. After that, the $aptitude$ class can be used as a regular package manager module, but it lacking functionality. It is necessary to add the common code, like the constructor and the manager's name. After these operations, the $aptitude$ module can be presented in listing 6.2. Since the package manager will read files from an input CSAR, the CSAR handler is stored by the constructor to the ch variable for a further use. In addition, a pointer to the language (to Bash in this case) is stored too, to be propagated later to the package handler.
Now consider the $proceed$ function. A line-by-line file analyzer is needed. It must modify the data and in the case of changes, the $isChanged$ variable should be set to $true$. The

Listing 6.3 The *aptitude* module *proceed* function

```
@Override
public void proceed(String filename, String source)
      throws FileNotFoundException, IOException, JAXBException {
      if (ch == null)
            throw new NullPointerException();
      List<String> output = new LinkedList<String>();
      System.out.println(Name + " proceed " + filename);
      BufferedReader br = new BufferedReader(new FileReader(filename));
      boolean isChanged = false;
      String line = null;
      String newFile = "";
      while ((line = br.readLine()) != null) {
            // TODO parsing will be done here
      }
      br.close();
      if (isChanged)
            Utils.createFile(filename,newFile);
      return output;
}
```

$isChanged$ variable indicates that the file must be rewritten with a new content from the $newFile$ variable. In the $output$ variable a set of packages from the package handler will be stored and returned back to the Bash module. Described behavior is implemented in listing 6.3.
An $aptitude$ line parser will be implemented. It must read a line from the $line$ variable and store it or it's modified version into the $newFile$ variable. If the data was changed, then the $isChanged$ variable must be set to $true$. Any $aptitude$ package installation calls should be detected, commented out and its arguments (package names) should be propagated one by one to the package handler's function $getPackage$. The result of the $getPackage$ function will be added to the $output$ variable.
During the parsing which is defined in the listing 6.4, the line is divided into words. Each found package name is transmitted to the packet handler as an argument of its public function $getPackage$. In additional, this function must take the language and the source artifact's name as the arguments.
Now the $aptitude$ module can be added to the Bash module. The only thing to do is to add the $aptitude$ to the Bash's list of package manager modules (the list is stored in the *packageManagers* variable). This is done by the Bash's constructor with the command: "*packageManagers.add(new PM_aptitude(this, ch));*". After these operation the new module is ready to identify and resolve external references which use the $aptitude$ package manager to install packages.

Listing 6.4 The *aptitude* module line parser

```
String[] words = line.replaceAll("[;&]", "").split("\\s+");
// skip spaces at the beginning of string
int i = 0;
if (words[i].equals(""))
      i = 1;
// looking for aptitude
if (words.length >= 1 + i && words[i].equals("aptitude")) {
      // aptitude found
      if (words.length >= 3 + i && words[1 + i].equals("install")) {
            System.out.println("aptitude found:" + line);
            isChanged = true;
            for (int packet = 2 + i; packet < words.length; packet++) {
                  System.out.println("package: " + words[packet]);
                  output.addAll(ch.getPackage(language, words[packet], source));
            }
      }
      newFile += "#//References resolver//" + line + '\n';
}
else
      newFile += line + '\n';
```

7 Example Usage of the Tool

In this chapter, the usage of the developed framework will be presented and validated. An input CSAR will be described in section 7.1. Output CSARs will be displayed by Winery in section 7.2. Generated Artifacts will be verified in section 7.3.
To start the framework an Java environment is used. After the start, a user should enter the input CSAR name, the output CSAR name, the architecture and choose the mode of operation. Then, the framework works fully automatically, analyzing the artifacts and resolving any external references. The output of the framework in the One node for one package mode and in the Sets of packages mode will be called the first CSAR and the second CSAR respectively.

7.1 Input CSAR

The handled CSAR provides a service for Automating the Provisioning of Analytics Tools based on Apache Flink [Ope16]. The structure of the service is defined in Figure 7.1. The service uses a server virtualization environment named $vSphere$ (the $VSphere\_5.5$ node). In the environment operates an $Ubuntu$ virtual server (the $Ubuntu$-14.04-VM node). The $Ubuntu$ hosts two applications: $Python$ (the $Python\_2.7$ node) and the $Flink\ Simple$ (the $Flink\_Simple\_1.0.3$ node). The service has two submodules: a Data Prediction and a Data Delivery, both are hosted on the $Flink\ Simple$ node and require the Python node. An analyze shows two external references. The $Python$ node installs the python package and the $Flink\ Simple$ node - the Java package.

7.2 Visualize with Winery

Winery was used to validate the correctness of the output CSARs. This is a tool for the development of TOSCA systems and is useful for verify the results. The input CSAR's representation by Winery is displayed in Figure 7.1. It contains external references but they are resolved by the framework and exchanged by new nodes in the output CSARs.
The output CSARs were added to Winery. Due to a significant increase in size of the first CSAR, this can be a fairly lengthy procedure. Fully dependencies three for the Java

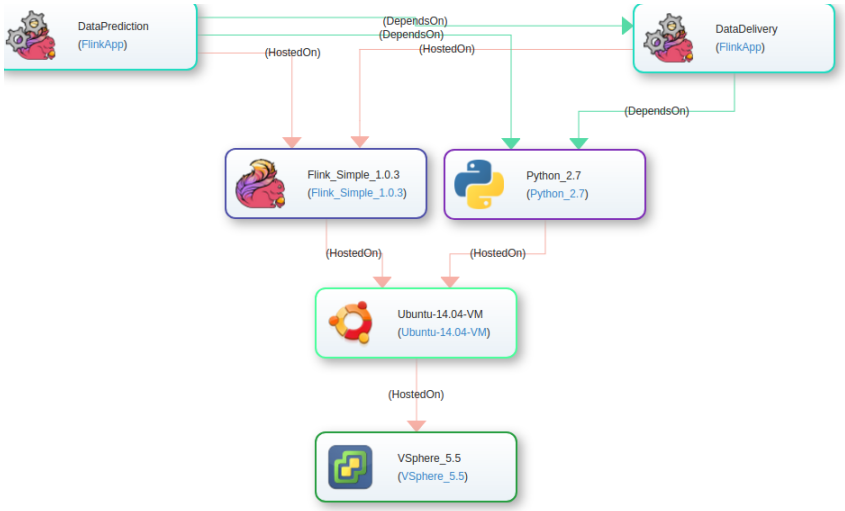

Figure 7.1: The input CSAR represented by *Winery*.
http://www.opentosca.org/demos/smart-prediction-demo/index.html

package consists of about 60 packages and the python requires about 50 more. In the One node for one package mode for each of these packages a new node must be created. It was only six nodes in the input CSAR, but after the processing, the first CSAR contains more than 100 of nodes. This problem doesn't occur in the Sets of packages mode. For two external references only two additional nodes were created. Therefore the addition of the second CSAR into Winery passes much faster. During the addition, the syntax of the CSARs is tested. In a case of errors, messages will be displayed.
Then the CSARs were displayed. To visualize a CSAR, Winery must validate and interpret it's internal references. Due to the high number of nodes, the processing of the first CSAR lasts longer as the processing of the second CSAR. If something was defined not properly, these erroneous nodes or references between them will not be displayed. The representation of the first CSAR is shown on Figure 7.2, but only a part of the CSAR is visible. The structure seems very difficult to follow. An observer can identify four old nodes, two new nodes and a mix of references. To verify the topology some nodes was moved manually. Figure 7.3 displays the result. Now its possible to identify references between some nodes. The second CSAR with manually moved nodes is presented on Figure 7.4, which shows the clear structure. New nodes installing python and Java were created and referenced to nodes which contained external references earlier.
The correctness of dependencies was verified by checking several references with the *apt-cache depends* command. By opening the content of some nodes, it was verified, that there are right artifacts in new nodes and that external references were deleted from old nodes. For example it was noted that the node installing python from the second CSAR contains more then 50 Deployment Artifacts and one Implementation Artifact which installs the corresponding packages.

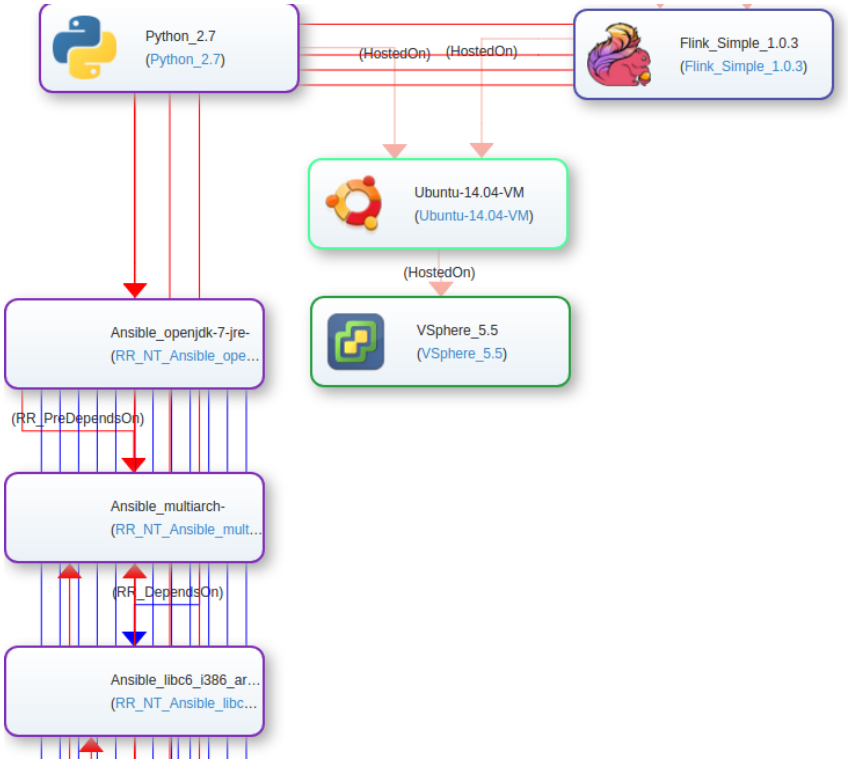

Figure 7.2: The CSAR processed in One node for one package mode and represented by Winery.

7.3 Validate Artifacts

It is necessary to verify whether it is possible to install new packages using the generated artifacts. At first Bash scripts and then ansible playbooks will be tested.

Validate Bash Scripts

Since Bash is used in the Linux's command line, it will be pretty easy to check Bash installation scripts by starting them. Of course that must be done with the necessary privileges. For the script file installing the $python2.7\text{-}minimal$ package, the command will be $sudo\ RR\_python2\_7\text{-}minimal.sh$. The installation ended without any warnings or errors, which means that it was completed successfully. This way any generated Bash script can be validated.

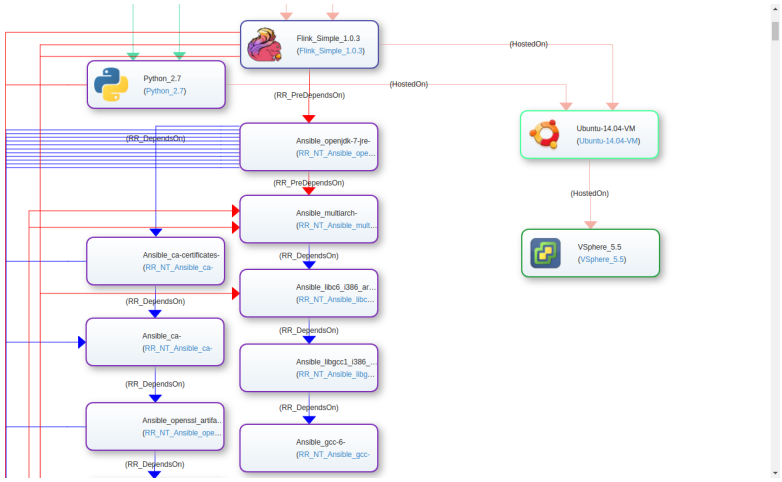

Figure 7.3: The CSAR processed in One node for one package mode and represented by Winery with some nodes moved manually.

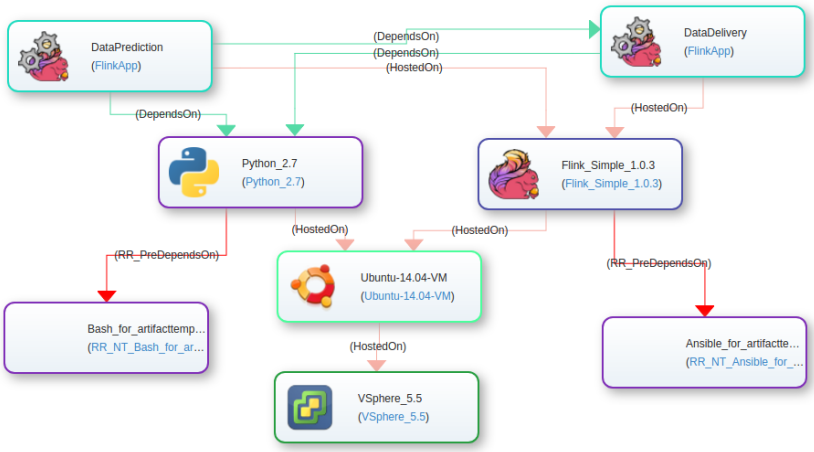

Figure 7.4: The representation of the CSAR processed in the Sets of packages mode.

Validate Ansible Playbooks

To validate an ansible playbook we need to extract the zip file containing the playbook manually. During the regular execution, this work will be done by a runtime environment. The call to the ansible runtime which proceeds the playbook is a simple procedure too. For the playbook with the *main.yml* name, the command will be *sudo ansible-playbook main.yml*. *Ok* at the end of the output signals that the installation was completed successfully.

8 Conclusion and Future Work

External references can negatively affect Cloud Applications. If we have a high level of information security or limited access to the Internet, these external dependencies can lead to a lot of problems with performance, stability and security. Unfortunately, during deployment, many TOSCA applications access external sources to install packages or download files. The purpose of this work was to develop a solution resolving external dependencies through encapsulation and to implement it in a prototype.
A concept of the modular framework was designed. According to the concept, modules will process different configuration management tools, package managers and data download utilities. Each module is responsible for identification and resolution of specified type of external references. To resolve a reference it's necessary to download the data needed during deployment. The downloaded data is integrated into the topology of the TOSCA application. Several modes of integration were presented, which can be suitable in different use cases.
The prototype of the modular framework was developed in the Java language. The program identifies and resolves references to external packages. The application can handle $Bash$ scripts and $Ansible$ playbooks. It has two modes of operation. In the first mode dependencies between packages are mapped to a TOSCA topology. Other mode serves to generate compacter CSAR with small number of nodes. The framework handles a CSAR as follows. The structure of the CSAR is analyzed to determine the internal references. Each package will be download along with all dependent packages and integrated into the CSAR. TOSCA nodes are generated for these packages. During deployment, a TOSCA runtime environment can analyze these nodes and install the necessary packages.
In order to show the extensibility of the framework, the addition of the $aptitude$ package manager module into the $Bash$ module was described in detail. It was shown how to create the module which can be added into the framework, how to implement its basic functions, pass data and integrate the module into the $Bash$ module.
At the end, the results of the framework's execution were validated. The output CSARs were visualized and analyzed with the help of Winery. Generated artifacts were verified and executed.

Future Work

The developed framework represents a prototype of the software which will be able to make the processed CSAR fully self-contained. Now it can be extended or even its concept can be rethought. There are many different directions available to expand the existing prototype. The package manager modules can be reworked to handle installation commands in better way. For example, the support of variables can be added into the apt-get module for Bash. Some new package manager or languages modules can be developed and added into the framework to handle for example Shef, CFEngine, yum, pacman, etc. Described, but not yet implemented download tool modules can be added in future.
The entire concept can be reworked in order to achieve higher level of abstraction. All types of modules can be grouped up to one abstract type. This will allow to add some new types of elements, like archive module with zip or rar submodules, which can handle archives separately without big changes in the structure of the software. Another direction of the improvement of the software's convenience is to implement a visual interface which allows a user to choose artifacts which must be processed and modes of the processing separately. Such visualization can be based on some existing program from OpenTOSCA, for example on Winery. This can be useful for composite Cloud Applications consisting of many parts with diverse architectures which must be handled in different ways separately.

Bibliography

[Ank+15] H. M. Ankita Atrey et al. *An overview of the OASIS TOSCA standard: Topology and Orchestration Specification for Cloud Applications.* 2015 (cit. on p. 13).

[Ans16] Ansible community. *Ansible (software)*. 2016. URL: http://docs.ansible.com/ansible/playbooks_best_practices.html#task-and-handler-organization-for-a-role (cit. on p. 26).

[Apa] Apache Software Foundation. *Apache Tomcat*. URL: http://tomcat.apache.org/ (cit. on p. 22).

[Apa06] Apache Software Foundation. *Apache Axis*. 2006. URL: http://axis.apache.org/ (cit. on p. 22).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA - A Runtime for TOSCA-based Cloud Applications." English. In: *Proceedings of 11th International Conference on Service-Oriented Computing (ICSOC'13)*. Vol. 8274. LNCS. Springer Berlin Heidelberg, Dec. 2013, pp. 692–695. DOI: 10.1007/978-3-642-45005-1_62. URL: http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2013-45&engl=1 (cit. on p. 22).

[Bun14] S. Bundesamt. *12 % der Unternehmen setzen auf Cloud Computing*. Dec. 19, 2014. URL: https://www.destatis.de/DE/PresseService/Presse/Pressemitteilungen/2014/12/PD14_467_52911.html (cit. on p. 13).

[Bun17] S. Bundesamt. *17 % der Unternehmen nutzten 2016 Cloud Computing*. Mar. 20, 2017. URL: https://www.destatis.de/DE/PresseService/Presse/Pressemitteilungen/2017/03/PD17_102_52911.html (cit. on p. 13).

[Bur13] M. Burgess. *Cfengine V2.0 : A network configuration tool*. 2013. URL: http://www.iu.hio.no/~mark/papers/cfengine_history.pdf (cit. on p. 26).

[CFE] CFEngine community. *CFEngine*. URL: https://cfengine.com/learn/learnhow-to-write-cfengine-policy/ (cit. on p. 26).

[Chr07] T. Chris. *OPIUM: Optimal Package Install/Uninstall Manager*. Mar. 15, 2007 (cit. on p. 24).

[cUR97] cURL. *command line tool and library for transferring data with URLs*. 1997. URL: https://curl.haxx.se/ (cit. on p. 27).

[Ecl] Eclipse Stardust Foundation. *Stardust*. URL: https://projects.eclipse.org/projects/soa.stardust (cit. on p. 23).

[HP09] J. S. C. Harold C. Lim Shivnath Babu, S. S. Parekh. "Automated control in cloud computing: challenges and opportunities." In: *ACDC '09 Proceedings of the 1st workshop on Automated control for datacenters and clouds*. June 19, 2009 (cit. on p. 13).

[IAA13] IAAS Universität Stuttgart. *OpenTOSCA - Open Source TOSCA Ecosystem*. 2013. URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/ (cit. on pp. 13, 22).

[Jan06] M. Jang. *Linux Annoyances for Geeks: Getting the Most Flexible System in the World*. 2006 (cit. on p. 24).

[KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. "BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications." In: *Business Process Model and Notation*. Ed. by J. Mendling, M. Weidlich. Vol. 125. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2012, pp. 38–52. ISBN: 978-3-642-33154-1. DOI: 10.1007/978-3-642-33155-8_4 (cit. on p. 23).

[Laz16] M. Lazar. *Current Cloud Computing Statistics Send Strong Signal of What's Ahead*. Nov. 3, 2016. URL: https://www.insight.com/en_US/learn/content/2016/11032016-current-cloud-computing-statistics.html (cit. on p. 18).

[Met15] C. Metz. *The Chef, the Puppet, and the Sexy IT Admin*. Wired. 2015. URL: https://www.wired.com/2011/10/chef_and_puppet/ (cit. on p. 26).

[Mic+10] A. F. Michael Armbrust et al. "A view of cloud computing." In: *Communications of the ACM* (2010) (cit. on p. 17).

[Nat] National Institute of Standards and Technology. URL: https://www.nist.gov/ (cit. on p. 17).

[OAS] OASIS. *Organization for the Advancement of Structured Information Standards*. URL: https://www.oasis-open.org/ (cit. on p. 19).

[OAS13a] OASIS. *OASIS Standard. Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 25, 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html (cit. on p. 13).

[OAS13b] OASIS. "Topology and Orchestration Specification for Cloud Applications Version 1.0." In: *OASIS Committee Specification 01. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html*. Mar. 18, 2013 (cit. on p. 19).

[Ope16] OpenTOSCA community. *OpenTOSCA for the 4th Industrial Revolution*. University of Stuttgart. 2016. URL: http://www.opentosca.org/demos/smart-prediction-demo/index.html (cit. on p. 59).

[Pet11] T. G. Peter Mell. *The NIST Definition of Cloud Computing*. Recommendations of the National Institute of Standards and Technology, Special Publication 800-145. National Institute of Standards and Technology, Sept. 2011 (cit. on pp. 17, 18).

[RF17] C. Ramey, B. Fox. *Bash reference manual: reference documentation for Bash edition 4.4, for Bash version 4.4*. Jan. 24, 2017 (cit. on p. 27).

[Sta16] Statista. *Umsatz mit Cloud Computing** weltweit von 2009 bis 2016 und Prognose bis 2020 (in Milliarden US-Dollar)*. 2016. URL: https://de.statista.com/statistik/daten/studie/195760/umfrage/umsatz-mit-cloud-computing-weltweit-seit-2009/ (cit. on p. 13).

[Win] Winery. *Eclipse Winery*. URL: https://projects.eclipse.org/projects/soa.winery (cit. on p. 23).

[Xia99] M. L. Xiaoyi Ma. *BITS: A Method for Bilingual Text Search over the Web*. Linguistic Data Consortium, 1999 (cit. on p. 27).

[Zim13] M. Zimmermann. "Konzept und Implementierung einer generischen Service Invocation Schnittstelle für Cloud Application Management basierend auf TOSCA." bachelor thesis. Institut für Architektur von Anwendungssystemen, Universität Stuttgart, Apr. 17, 2013 (cit. on p. 22).

All links were last followed on July 30, 2017.

Declaration

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

---

place, date, signature