Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 12345

# Self-Containment Packager Framework for TOSCA Cloud Service Archives

Yaroslav Nalivayko

**Course of Study:** Softwaretechnik

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

**Supervisor:** Michael Zimmermann, M. Sc.

**Commenced:** 10. February 2017

**Completed:** 10. August 2017

**CR-Classification:** I.7.2

# Abstract

In recent years, Cloud computing is gaining more and more popularity. But it's difficult to create one Cloud application that is suitable for work with different Cloud service providers. **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplication (TOSCA) provides a solution for this problem. This standard adds an additional level of abstraction to the Cloud applications, in other words, a layer between the external interfaces of a Cloud application and a Cloud service provider's API. With the help of TOSCA, it's possible to describe several models of interaction with many different APIs, what allows to automate the rapid redeployment between providers, which are using completely different API. Description of a Cloud application is stored in a **C**loud **S**ervice **AR**chive (CSAR), which contains all components necessary for the Cloud application life-cycle. The University of Stuttgart implemented this specification in the runtime environment named OpenTOSCA.

Cloud systems are often described in such way that during they deployment, additional packages and programs need to be downloaded via the Internet. Even with a single server, this can slow down the deployment of a Cloud application. And if the Cloud application consists of a large number of servers, each of them downloading a large amount of data during the deployment, this can significantly increase both time and money consumption.

This document considers the concept, architecture and implementation of the developed software solution which will recognize external dependencies in a CSAR, eliminate them, resupply the CSAR with all packages necessary for deployment and also change the internal structure of the CSAR to display the achieved self-containment.

# 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

**CSAR** **C**loud **S**ervice **AR**chive. 19

**IaaS** **I**nfrastructure **a**s **a** **S**ervice. 16

**NIST** **N**ational **I**nstitute of **S**tandards and **T**echnology. 15

**OASIS** **O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards. 17

**PaaS** **P**latform **as** **a** **S**ervice. 16

**SaaS** **S**oftware **as** **a** **S**ervice. 16

**TOSCA** **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplication. 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 mainframe 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] The transfer of a Cloud application configured to interact with the **A**pplication **P**rogramming **I**nterface (API) of one provider to work with another provider and another API is a difficult but important task. The ability to move a Cloud application to the more suitable provider quickly is a key to the development of competition and reducing the cost of maintenance. **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplication (TOSCA) [OAS13a] is a standard to solve this problem. TOSCA defines a meta-model to describing Cloud application's definition and management portable and interoperable. The use of TOSCA allows to simplify and automate the management of Cloud applications by different providers. According to TOSCA standard a structure and management data are stored in a **C**loud **S**ervice **AR**chive (CSAR). This archive contains the description of a Cloud application, its external functions, internal dependencies and the data for the deployment and operation.
OpenTOSCA [IAA13] is an open source constantly improving and expanding ecosystem for the TOSCA standard developed by the University of Stuttgart. OpenTOSCA processes data in CSAR format and performs specified operations. Installation operations often contain links to external packages and programs which will be subsequently downloaded over the Internet for the deployment of a Cloud application. These downloads can add expenses to the time required to download packages, money spent on rent of an idle server and Internet traffic for megabytes of pre-known data. For many Cloud applications, this may mean a few seconds of delay. But for a large distributed application which contains a lot of identical nodes requiring the installation of the same external packages and programs, the costs can increase significantly.
The other problems of external dependencies are security and stability. To ensure the security of information some firms restrict the access to Internet. In other networks the Internet access is extremely limited. For example, there can be no broadband access,

slow communication only over a satellite at certain hours, etc. An attempt to deploy a Cloud application with external dependencies in such networks may not succeed.
To solve these problems a software solution for removing external dependencies in CSARs will be developed and implemented during this work. This software will analyze a CSAR, identify dependencies to external packages and resolve them by downloading the necessary data to install the package as well as data for all depended packages. Then all downloaded data will be added into the CSAR's structure to represent the changes made.
This software must easily be expanded (in other words - to be a framework) since it is impossible to predict and describe all possible types of external dependencies. The output of the framework is a CSAR which contains additions to the original structure, like all the packages necessary for the deployment of the Cloud application, with the minimum possible level of access to the Internet during operation.

## Structure

The work structure is as follows:

**Chapter 2 – Basis.** This chapter explains the basic terms of this work, which include definitions and descriptions of Cloud applications (section 2.1), TOSCA standard (section 2.2), OpenTOSCA environment (section 2.3) and packet management (section 2.4).

**Chapter 3 – Requirements.** It clarifies requirements for the framework.

**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 contains the description of the implementation. It explains the design and development of individual components of the software.

**Chapter 6 – Add New Package Manager Module.** The new package manager will be added into the framework to proof the ease of extensibility.

**Chapter 7 – Validation.** In this chapter the output of the developed program will be presented and validated.

**Chapter 8 – Summary.** The results of the work will be summarized in the last chapter.

# 2 Basis

In this chapter, the fundamentals used in this work will be explained. These include definitions for Cloud computing and Cloud applications, description of TOSCA standard and its implementation: OpenTOSCA. At the end, a package management and tools for its automation are described.

## 2.1 Cloud computing and Cloud application

Understanding the problem requires a clear definition of the term "Cloud computing". 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 used. 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]
But computing is too abstract term, for our purpose we need something more practical, like an application. Also, the are no generally accepted definitions of Cloud application, but it can be obtained from the definition of Cloud computing.
*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 user will be provided. A composite Cloud application which consist of multiple small applications will be called a **Cloud system**. An owner of the physical platform, where Cloud computing takes place is called a **provider**. An owner of the Cloud application renting a provider's platform is called a **user**.

### Service Models

Cloud applications provide a wide range of different services. 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 **as 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 **as 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 **as 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 and development, big data analyses, file storage and so on. Cloud computing allows using resources effectively, to distribute the load to a system from several physical servers 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 user doesn't need to maintain a team of administrators for

the event of various problems.
A user doesn't have a direct access to the infrastructure (servers and operating systems) when using a PaaS or a SaaS service models. He can operate only with the provided Application Programming Interface (API). An API provides a set of methods to communicate with provider's infrastructure. Each provider defines 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 redeploy an application to another provider.

## 2.2 Topology and Orchestration Specification for Cloud Applications

The Topology and Orchestration Specification for Cloud Applications (TOSCA) standard developed by **O**rganization for the **A**dvancement of **S**tructured **I**nformation **S**tandards (OASIS) [OAS] provides a new way to enable portable automated deployment and management of Cloud applications. TOSCA describes the structure of an application as a topology containing components and relationships between them. TOSCA application is a Cloud application described according to the TOSCA standard. This standard can be used not only to describe all stages of a Cloud application life-cycle but also to serve as a layer between the Cloud application and provider's API, allowing to implement a single application suitable for working with different providers.

### Structure of TOSCA Applications

TOSCA specification provides a language to define components (described in section 2.1) and relationships between them using $Service\ Templates$. In addition it describes the management procedures which create or modify services using orchestration processes. The description of elements of a TOSCA structure used in this work is provided.
A $Service\ Template$ is the main component in a TOSCA structure. It defines the structure (Topology Template) and process models (Plans) of the service. There can be many Service Templates within one TOSCA application. The combination of topology and orchestration in a Service Template defines what is needed to be preserved across deployments in different environments to enable interoperable deployment of Cloud services and their management throughout the complete lifecycle (e.g. scaling, patching, monitoring, etc.). This is useful when an application is ported to alternative Cloud environments. [OAS13b]
$Plans$ provide capabilities to manage Cloud applications, especially their creation and termination. These components combine management capabilities to create high-level

management 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* defines the properties of such a component and the operations available to manipulate the component. 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 direction of the connection. A *Relationship Type* defines semantics and properties of the relationship. A Node Type and Relationship Type can by instantiated multiple times. Those types are like abstract classes in high-level programming languages and Templates are objects of those classes.

A simple cloud application for weather calculating 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 defined components. It will a *compute* operation for the python script, an *install* operation for the python environments and *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 *weater calculator*. For each specified *Node Type*, a 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 content necessary for a management 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*. An *Implementation Artifact* represents the executable of an operation described by a Node Type. An *Deployment Artifact* represents the executable for materializing instances of a node. An *Artifact Type* describes a common type of an artifact: python script, installation package and so on. An *Artifact Template* represents information about the artifact. For example the location of the artifact and other attendant data are stored in the *Artifact Template*. As in the example with nodes above, types are like classes, templates are like objects, and artifacts represent content or a value of an object, but these values can't be changed. *Node Type Implementation* defines the artifacts needed for implementing the corresponding Node Type. If a Node Type contains *deploy* and *shutdown* operations, then the corresponding Node Type Implementation can contain two Implementation Artifacts with scripts for these operations and one Deployment Artifact with data necessary for the deployment. Implementations are like final classes

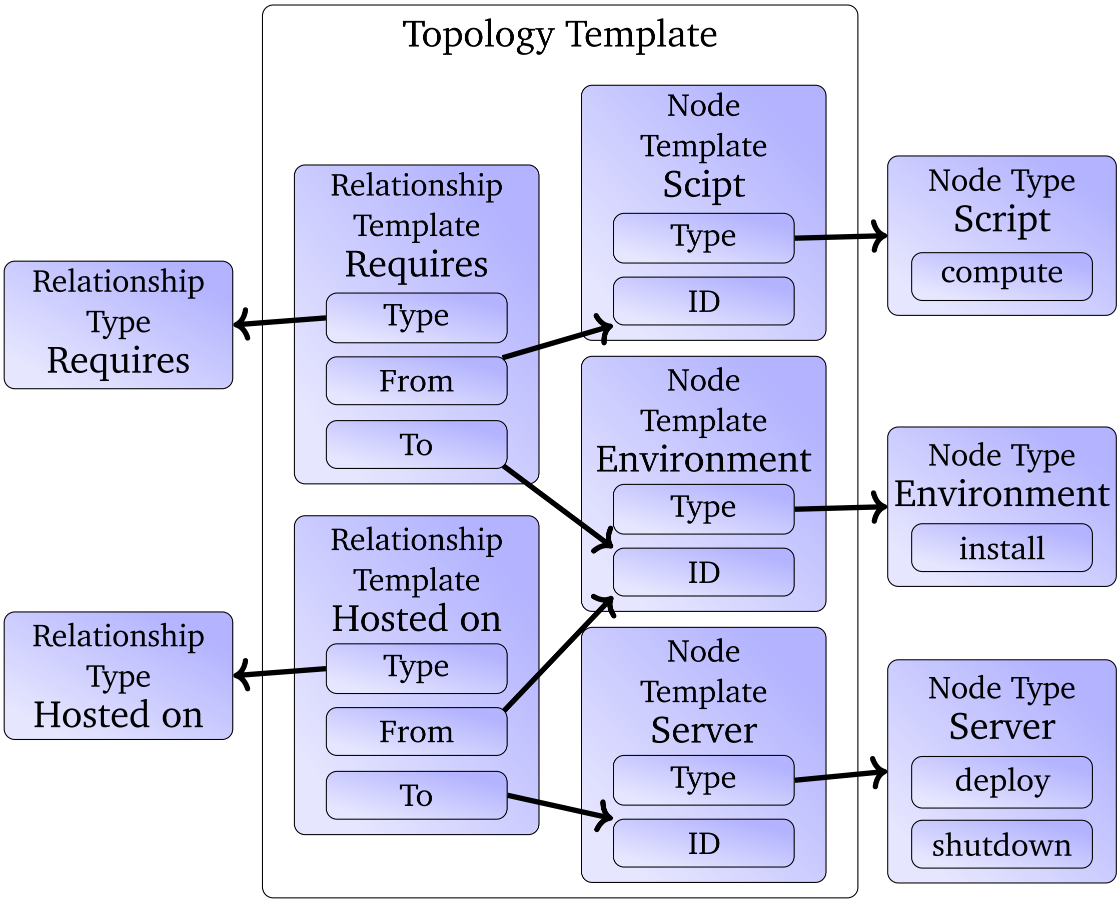


**Figure 2.1:** Example: a cloud application for weather calculation

between Node Types and Node Templates, but in TOSCA standard, the Implementation will be chosen only during execution. Types, Templates, and Implementations defining a TOSCA application are stored in definition document which have the XML format. The combination of topology and orchestration in a Service Template defines what is needed to be preserved across deployments in different environments to enable interoperable deployment of Cloud services and their management throughout the complete lifecycle (e.g. scaling, patching, monitoring, etc.). This is useful when an application is ported to alternative Cloud environments. [OAS13b]

## CSAR

To store 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 of TOSCA application. They include definition documents, artifacts and so on. 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 CSAR itself (TOSCA version, CSAR version, creator and so on). All other pairs represent metadata of files from the CSAR. The metadata is used by a TOSCA runtime environment to process given files correctly.

## Encapsulation of CSARs

The encapsulation must be achieved through the download of external packages and generation of a new TOSCA node for each of them. But it can be interesting to analyze other techniques to encapsulate a CSAR. At first, we will described the methods not representing packages in a TOSCA topology and then the 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 device 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 in a TOSCA structure. The main problem is the creation of the custom repositories. When a TOSCA application consists of many small devices with limited capabilities it can be difficult to start many big custom repositories.

### Generate Shared Repository

Another opportunity is to create a single repository for all devices in a TOSCA application. It can be difficult to choose the right location for such a server, but since an application represents the connected system 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

This method was suggested by IAAS. 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 a TOSCA application and dependencies between packages.

#### Sets of Packages

A set of depended packages from a dependencies tree related to an external reference can be archived 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. Of course, some packages can be saved in different archives redundantly, but a small size of a TOSCA application's structure will be achieved. It will be impossible to trace dependencies (since all packages are represented by one node), but it can help to avoid a difficult structure which consists of hundreds of nodes.

## 2.3 OpenTOSCA

OpenTOSCA provides an open source web-based ecosystem for TOSCA applications. This ecosystem consists of three parts: the TOSCA **runtime environment**, the graphical modeling TOSCA 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.

### Runtime Environment

The runtime environment enables a fully automated plan-based deployment and management of Cloud applications contained in a CSAR. The architecture of the environment is visualized by Figure 2.2. Requests to the Container API are passed to the Control component, which orchestrates the different components, tracks their progress, and interprets the TOSCA application. The Core component offers common services to other components, e. g., managing data or validating XML. Management operations of nodes and relationships are either provided by running (Web) services, e. g., the Amazon EC2 API, or by Implementation Artifacts contained in the CSAR. In the latter case, the Implementation Artifact Engine is responsible to run these artifacts in order to make them available for plans. The plugin architecture of the Implementation Artifact Engine ensure extensibility. Implementation Artifacts, e. g., a SOAP Web service implemented

as Java Web archive, are processed by a corresponding plugin of the engine which knows where and how to run this kind of artifact. The plugins deploy the respective artifacts and return the endpoints of the deployed management operations to be stored in the Endpoints database. The deployment of Web Archives on Tomcat [Apa] and Axis Archives on Apache Axis [Apa06] is supported [Zim13]. The Plan Engine handles plans in the same manner. It is also build according to a plugin architecture and supports different workflow languages, e.g., Business Process Model and Notation (BPMN) or Business Process Execution (BPEL) Language, and their runtime environments. [BBH+13]
A processing is done in following manner. First, the CSAR is unpacked and the files are put into the files store. Then, the TOSCA definitions documents are loaded, resolved, validated, and processed by the Control component, 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. Finally, the Plan Engine binds and deploys the application's management plans. The endpoints of the management plans are stored in the Plans database. [BBH+13]

### Winery

Winery provides a complete set of functions for graphically create, edit and delete elements in the TOSCA topology presented by a CSAR. It consists of four parts: the type and template management, the topology modeler, the BPMN4TOSCA plan modeler [KBBL12], and the repository.
The type, template and artifact management enables managing all TOSCA types, templates and related artifacts. This includes node types, relationship types, policy types, artifact types, artifact templates, and artifacts such as virtual machine images.
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: BPMN4TOSCA. That means 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, the exchange format of TOSCA files and related artifacts. [Win] An example of the TOSCA topology visualization is presented in Figure 2.3.

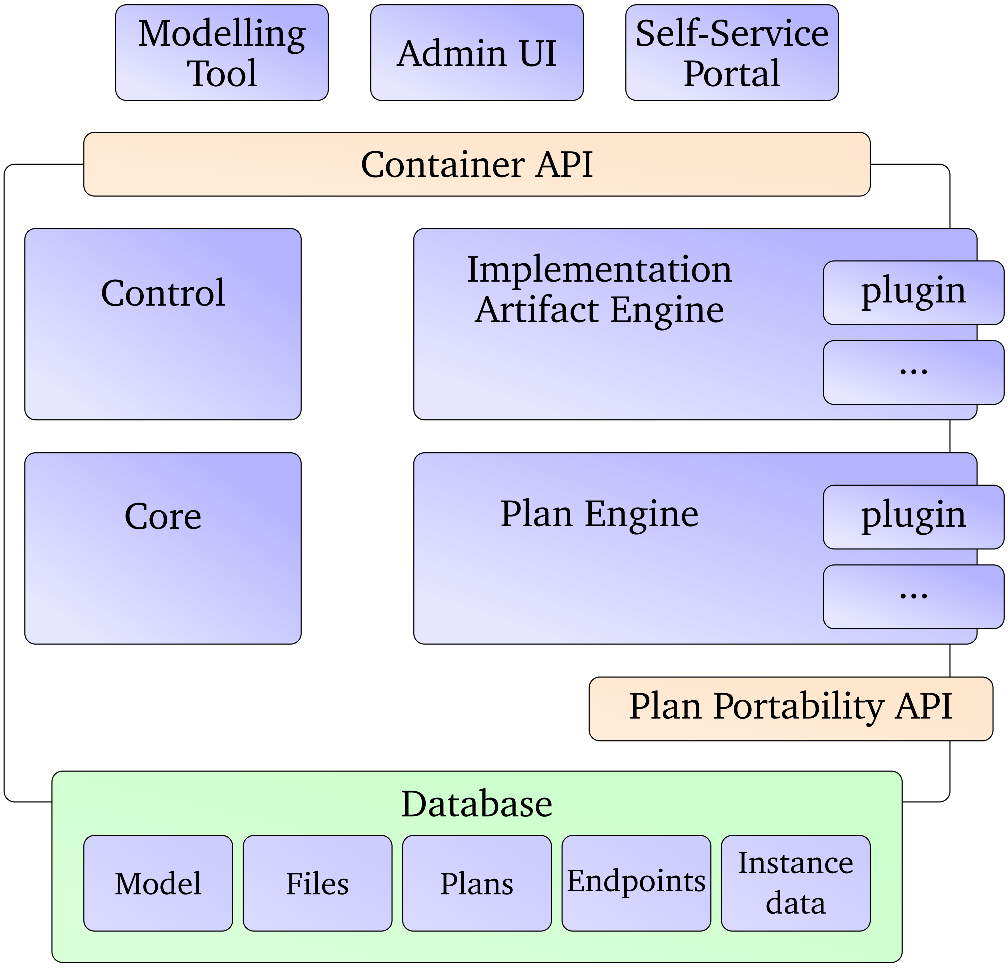


**Figure 2.2:** OpenTOSCA Architecture

## 2.4 Package Management

Packages and package management processes are described in this section. An installation of packages from external source represents an external reference and therefore they will be considered in order to identify such references. Package is an archive file containing both data for installation of the program component and a set of metadata like name, function, version, producer, and a list of dependencies to other packages. [Chr07] These packages can present not only a complete program but also a certain component of a large application. For a user, a package manager is a set of software tools which automate the process of installing, updating, configuring and removing packages. But from the operating system side, a package manager is used for managing the database of packages, their dependencies, and versions, to prevent erroneous installation of programs and missing dependencies. This task is especially complex in computer

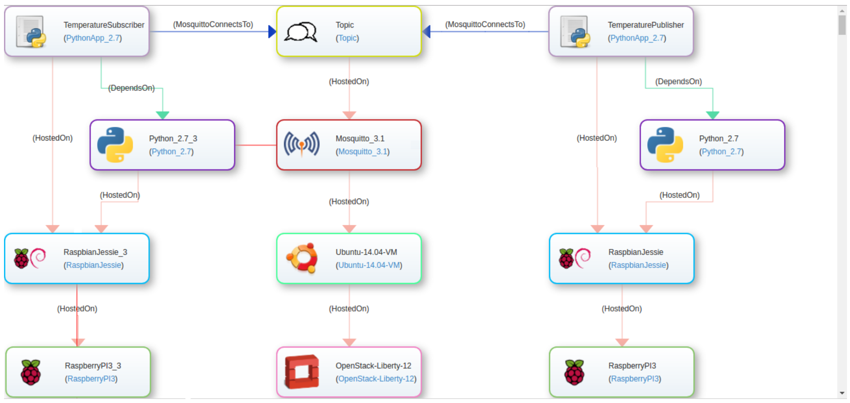


**Figure 2.3:** TOSCA topology visualized by $Winery$.

systems relying on dynamic library linking. Those systems share executable libraries of machine instructions across applications. In these systems, 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 systems, a package manager uses official repositories appropriate for the operating system and the architecture of device where it's 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 which differs them from the normal tree graph structures.

## Example Dependencies Handling

The *apt-get* package manager will be considered to provide an example of a dependencies handling. 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 a database of packages and their condition. These relations are presented in Figure 2.4. *apt-get* has many functions: install, remove, update, autoremove, download and so on.

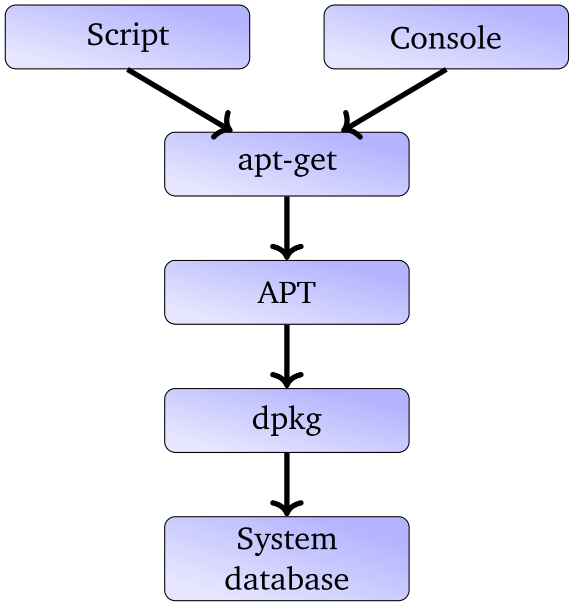


**Figure 2.4:** Package management

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 these 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 starting with the bottom of the 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 packages from the dependencies tree, the *package* will be ready to work.
A *package* can be deleted during 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 also 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 Tools

To ease the package management, various tools can be used. We will consider some of them to determine the form of files which contains external references. Usually they are presented by commands in an executable file, which checks an environment and installs the necessary packages using a package manager. Such files are called scripts and are commonly used. Popular management tools like Bash, Ansible, Chef and CFEngine will be described below.

### Bash

Bash is a Unix command language written as a free software. It provides enough capabilities to be used as a management tool. 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 systems [RF17]. Instead of typing commands direct into a command line, a script can be executed directly. [Ham11] These scripts can be used to configure a system, install package, create files, check environment and so on. Bash is a very popular and ease language, therefore a huge number of problems have solutions in Bash scripts already.

### Ansible

Ansible is an open-source automation engine that automates software provisioning, configuration management, and application deployment. 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 which can be grouped together as a "cookbook" for easier management define a series of resources that should be in a particular state: packages that should be installed, services that should be running, or files that should be written. 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. These files define the necessary state of files, packages, users, processes, services and so on. [CFE]

# 3 Requirements

In this chapter, requirements for the program and its components being developed will be defined. Furthermore it will be described what we expect as a result of the application's work.
It's necessary to develop new software, which will resolve external references and complement the TOSCA topology to represent the changes in a given CSAR file. Since the main purpose of the software is to resolve references, further it can be called $References\ resolver\ framework$. 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 identify references to external packages. That includes a wide range of package installation command types which contact package repositories with the help of different package managers. To find such commands the software must be able to unpack the data and access all artifacts contained in a input CSAR . Since one can't develop an application which support all possible types of external references the structure of the application must ensure an 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 reference type. The two types of modules must be developed. One type of modules will be responsible for configuration management tool and the other type will process package managers.
When an external reference is identified it must be resolved. T hat includes remove of package installation command and addition of the package 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 to 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. As a last step the extracted input CSAR with additions must be packed to the output CSAR. The concept and architecture of a program which satisfies the requirements will be described in chapter 4. The implementation of the program is 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 must be resolved. The output CSAR must be able to be deployed properly without downloading these packages over the Internet. In addition, the dependencies trees for packages from new nodes should be represented in the TOSCA topology.
In order to validate the output CSAR, the TOSCA topology can be verified and the defined artifacts can be validated through those installations on a test machine.

## Summarize

The requirements are summarized here. The developed framework must:

1. identify references to external packages in a input CSAR.
2. delete the external references.
3. add data into the CSAR to encapsulated installations of the packages.
4. handle different configuration management tools and package managers.
5. have an easy expandable structure.
6. represent the packages into the TOSCA topology.
7. 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 framework which can satisfy the requirements will be explained and substantiated. Solutions to some additional problems will be presented.

## 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. In section 4.1.1, it will be explained how to determine the Node Templates which use the given artifact. Then language modules and package manager modules functionality will be presented. In section 4.1.4, it will be expressed how to create a new node for a TOSCA topology. After that, a problem of the determination the architecture of the final platform will be explained and a solution presented. In addition, it will be described, how the results can be validated.

### 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

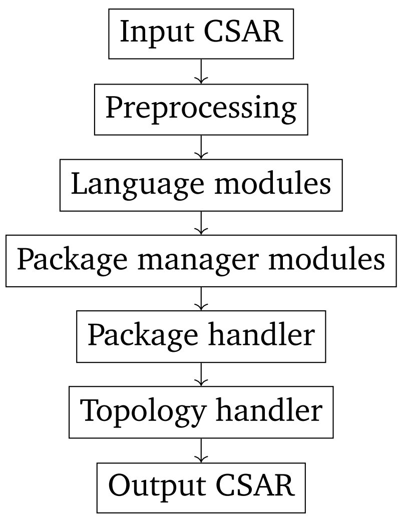


**Figure 4.1:** General description of the software's work flow

references. According to TOSCA standard, references between Node Templates can only be created in the same Service Template. That means that each Node Template which uses artifacts with external references must be found. Furthermore, Service Template where these Node Templates are instantiated must be determined to create there a Node Template for the new nodes and reference them to the Node Templates with external references. The pointers to artifacts are contained by Artifact Templates which are used by Node Type Implementations. By composing all the information the simple 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 instantiated only once.

This structure can be described as a tree with an Artifact as a root, and Service Templates as leaves (The example is on Figure 4.2) and will be called the internal dependencies tree.
There is an additional problem in the reference between a Node Type and a Node Type Implementation. 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, so the dependency from Artifact to Node Types can be built.

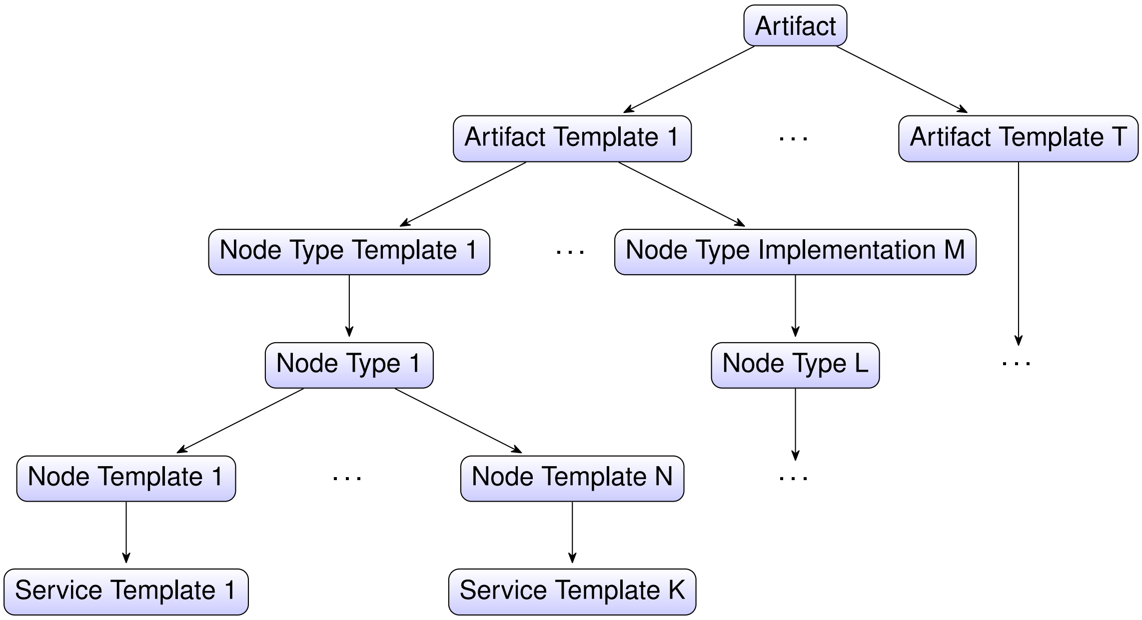


**Figure 4.2:** An example tree describing how to find Service Templates and Node Templates for a given script

- Find all Service Templates and all contained Node Templates they contain. Each Node Template contains a reference to 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.

### 4.1.2 Search for Artifacts

Since external references are stored in artifacts, we need to find all of them in order to identify the references. The first simple solution is to analyze the structure of TOSCA application and identify all artifacts used. But this method brings an possibility to miss some artifacts because some of them can be called from other artifacts. This case is presented by Figure 4.3. In this example Implementation Artifact Engine calls the "Artifact 1" which calls the "Artifact 2". The "Artifact 2" isn't called by Implementation Artifact Engine directly and therefore will not be considered by the method described above. The found solution is to analyze all files presented in a input CSAR and resolve external references in all of them. We need to describe methods to identify files not described in the TOSCA topology for each supported configuration management tool.

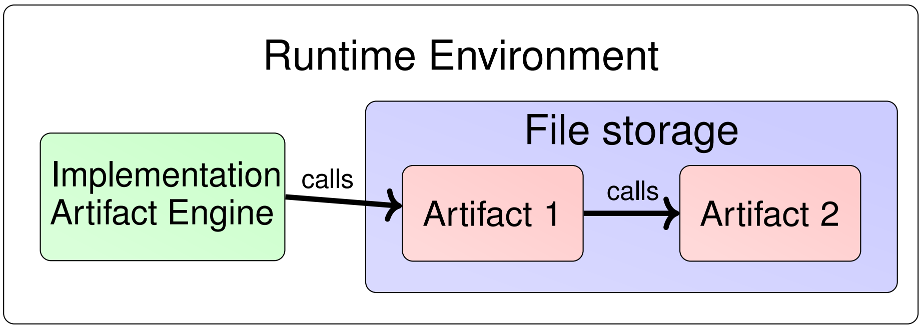


**Figure 4.3:** Bad artifacts call sequence

**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.
```

### 4.1.3 Modules and Extensibility

It is impossible to identify all types of external references, even when only one language and one package manager are used (an example 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.
The framework should handle different languages, each of which can support various package managers. The best solution is to develop a modular system, where modules handle different languages and package managers. The framework will contain a language modules and each language module will contain package managers modules. A language module should filter files not belonging to the language and the accepted files will be transmitted to the corresponding package manager modules. This principle can be illustrated by Figure 4.4.
A package manager module resolves an external reference and transmits the package name from this reference to a package handler described in section 4.2.5.
Ease of adding of new modules to the framework will prove the correctness of the architecture.

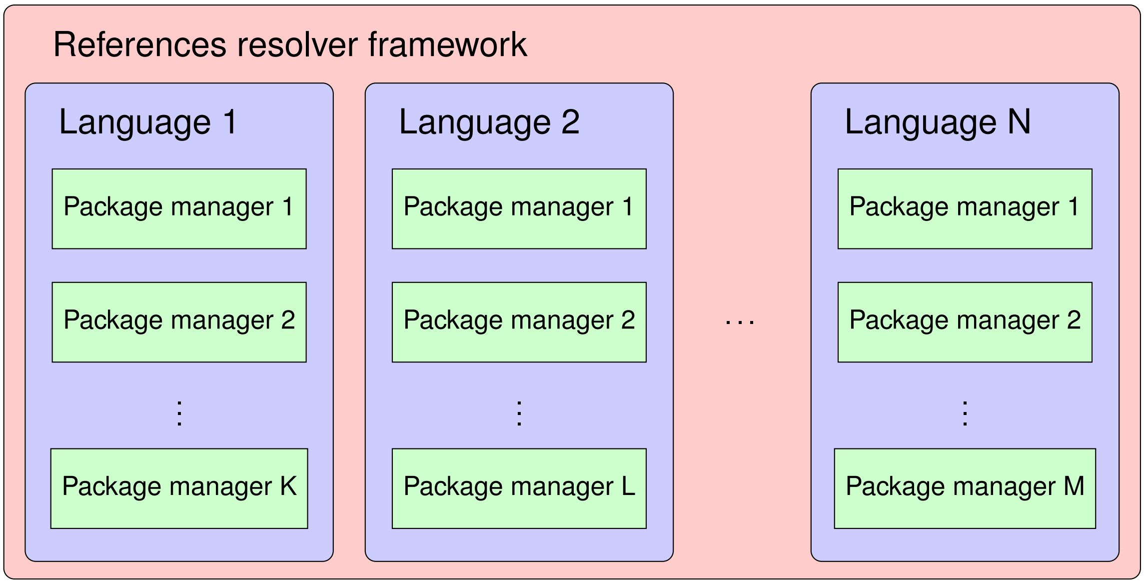


**Figure 4.4:** An example scheme representing several language modules containing package manager modules

At the beginning the most popular combination must be developed: the $bash$ language with the $apt - get$ package manager. This simple and powerful tool allows to install, delete or update the set of packages in one line of code. A line-by-line parser which analyses scripts and finds the installation commands can parse such commands and will be implemented. After the modules for this combination will be implemented and validated, new language and package manager modules can be added.

### 4.1.4 Representing Downloaded Packages in a TOSCA-Topology

A package node denotes the defined and instantiated element of the TOSCA topology, the purpose of which is to install the package. The addition of new package 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 package node common definition will be represented by a Node Type. It must contain the $install$ operation, which represents the capability to install the node.
- Artifacts (the downloaded data and the installation script) will be referenced by Artifact Templates.
- A Node Type Implementation will combine the artifacts to implement the $install$ operation.

- A Node Template will instantiate the package node in the corresponding Service Templates. To determine the corresponding Service Template the autor will use the preprocessing described in the section 4.1.1.
- A Reference Template will provide topology information allowing the observer (a user or a runtime environment) to determine which nodes the package must be installed for. References will be created from the Node Template which needs the package to the Node Template of the created package nodes.

After an execution of these steps, a definition of a package node will be finished and this node can be used.

### 4.1.5 Determining Architecture of the Final Platform

It can be difficult to choose the architecture of the device where packages will be installed. Unfortunately, it is impossible to analyze the structure of any CSAR and give an unambiguous answer to the question which architecture which node 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 with a bash script containing "*apt-get install python*" command can be deployed on different devices within one Service Template (for example with the arm, amd64 and i386 architectures) and will result in the loading 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*
  The script can analyze the system where it was started (for example using the "$uname -a$" command) and depending on the result, it will install the package corresponding to the system's architecture.
- *Unified architecture*
  The architecture will be defined by the user 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. As a consequence, chaos may occur while mirroring these different packages with different

versions to the TOSCA topology. The only found robust solution is to create a set of archives for each installed package. Using this method, data for one architecture are stored into one archive. Such archives will contain the entire dependency tree for the given package. But this approach contradicts one of the main ideas of this work: the dependencies trees should be mapped to the topology.
The *artifact specific architecture* method carries an additional complexity to the user of the framework. It will make a user to 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 method of the *unified architecture* was chosen as the simplest and easiest to implement. If it will be necessary, this method can easily be expanded to the *artifact specific architectures* method (by removing the user input at start, and choosing an architecture for each artifact separately) or to *deployment environment analysis* (by downloading packages for all available architectures and adding the architecture determining algorithm to the installation scripts).

### 4.1.6 Validation

Checking the output of the framework is an important stage in the development of the program. It is necessary to verify both the internal correctness of the output CSAR and the possibility to deploy generated package nodes. The validity of internal dependencies can be checked by *Winery* tool from OpenTOSCA. This tool for creating and editing CSAR archives is also great for visualizing the results. Checking the deployment of the generated package nodes can be done manually by entering commands which start the artifact's execution.

## 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***, 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 describes the processes of adding the new files (to handle the metadata),

archiving/unarchiving, and choosing 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 full path to of the file. The value contains the type of the file. This type will be used by a runtime environment to chose the right behavior. As already was mentioned in section 4.1.5, an architecture of the final platform will be chosen for the entirely CSAR. A command line interface must be provided for a user, to allow him to chose the architecture. 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: $preprocessing$, $processing$, $finishing$.
During the $preprocessing$ stage, the CSAR will be unarchived, common files added, and internal dependencies trees generated. Figure 4.5 illustrates those steps. During the $processing$ stage, all $language\ modules$ will be activated, the 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.

### 4.2.3 Language Modules

Each $language\ module$ describes a handling of one language and chooses files written in the language. It also contains a list of supported package manager modules. Each language module must provide the capability to generate a TOSCA node for the given package and this node must use the same 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 script and definitions for Artifact Templates, a Node Type, and a Node Type Implementation should be created by a language module. As it is already mentioned above, during the $processing$ stage a language module analyzes all files one by one and checks their belonging to the language. Any files not belonging to the described language are filtered out. The remaining files are transferred to the language module's $package\ manager\ modules$. For example, a $bash$ module will pass only files with ”$.sh$” extension which start with the ”$\#!/bin/bash$” line. An $ansible$ module should have an additional functionality to unpack zip archives where ansible playbooks can be stored. Since ansible playbooks don't contain a specific header or marker, the single sign of ansible files is the "$.yml$" extension.

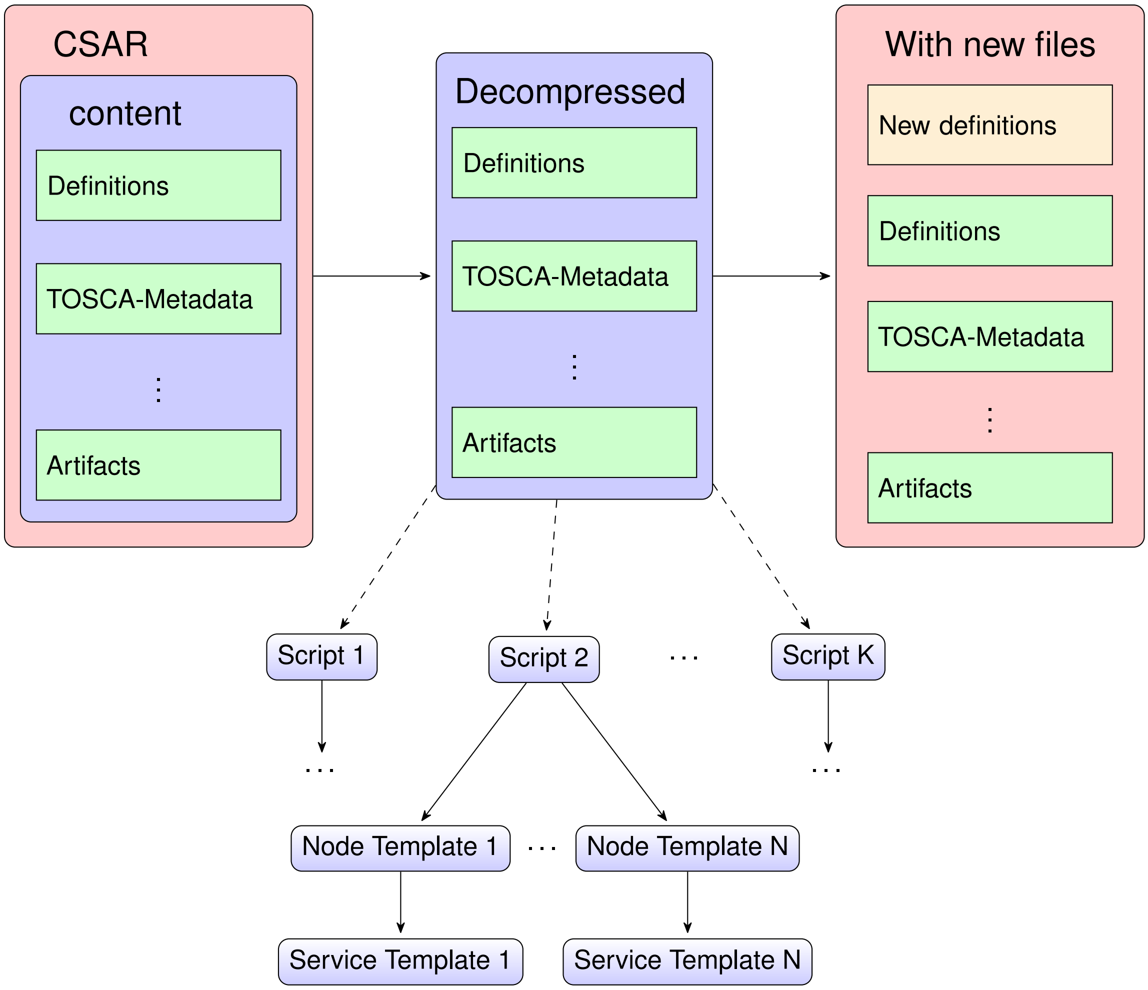


**Figure 4.5:** Preprocessing: decompression, adding files and generating dependencies

### 4.2.4 Package Manager Modules

A *package managers module* finds external references, resolves them and transmits the package name to the *package handler* described in the next section. Figure 4.6 illustrates data flow between language modules, package manager modules and the package handler.
To resolve an external reference a package manager module will parse the given file. In the case of the apt-get module for bash, the module will read a file line-by-line searching for the commands starting with "apt-get install". Such commands must be commented out and their arguments should be divided into separate package names which will be transferred to the package handler.

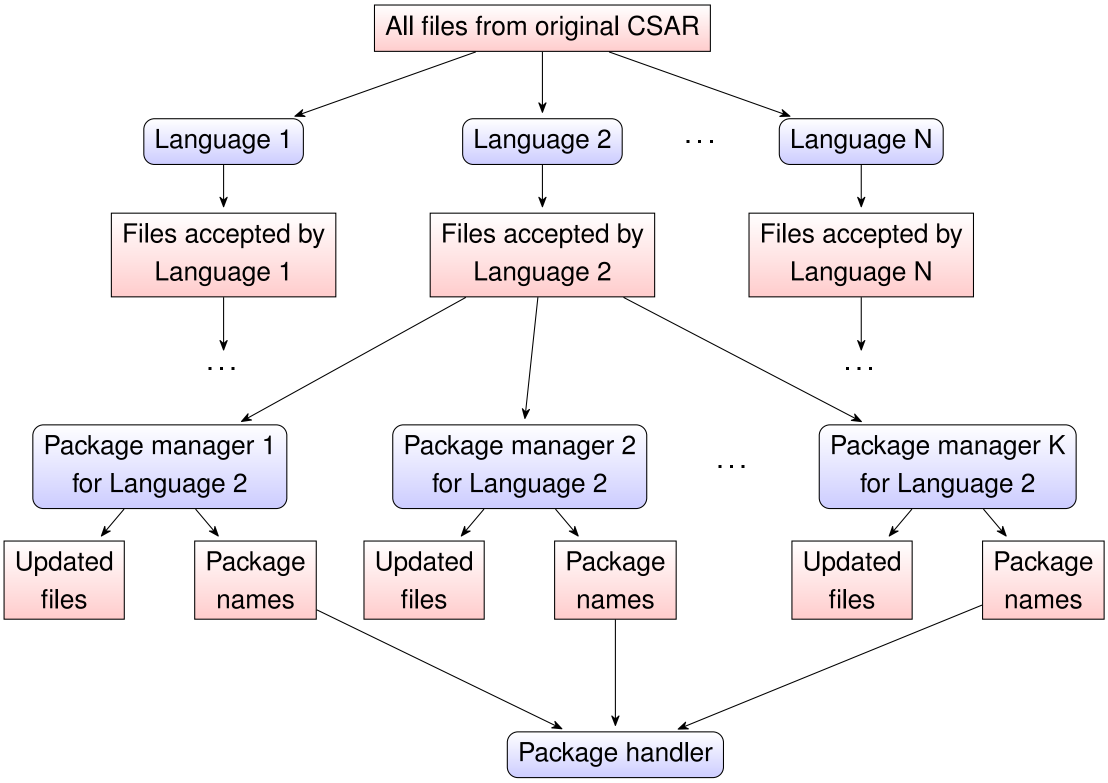


**Figure 4.6:** The data flow scheme between language modules, package manager modules and the package handler.

### 4.2.5 Package Handler

The *package handler* communicates 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 a given package. This component will download the installation data using given package name and an architecture specified by the CSAR handler. Then it transfers the package name to the *topology handler* and repeats the actions for all depended packages recursively. The download the data the command "apt-get download **package**" can be used. 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 the "apt-cache depends **package**" command. The output of such command should be parsed in order to extract 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. It can be: retry the download, ignore the package, rename the package or even break the framework's execution.

### 4.2.6 Topology Handler

This element should handle 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 TOSCA definitions for new package node include Node Templates and Relationship Templates which were described in section 4.1.4. 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 framework and its elements, whose behavior was described in chapter 4. Java language was chosen, because of its simplicity and strength.

## 5.1 Global Elements

This section describes the elements used throughout the whole framework's execution. A ZIP handler provides a functionality to operate ZIP archives, a CSAR handler keeps an interface to interact with a CSAR and a Utils helps to solve problems common to 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 most valuable 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 is 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 $createFile$ other elements of the framework can create a file with the given content. The $getPathLength$ function 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 and so on. The function $correctName$ can be used to obtain an acceptable name from a given name.

## 5.2 References Resolver

This is the main module which starts by framework startup and is executed into three stages: preprocessing, processing and finishing. These stages will be described shortly.

### 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 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 document which are used to store TOSCA definitions. Those 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.

**Listing 5.1** 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() {}
            }
      }
}
```

- $Script\ Artifact$ 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 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 is used.

### Processing

During this stage, all language modules listed in the framework are started. For the references resolver element that is only two strings of code, but they start the main functionality of the framework. The languages 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 and translate to the corresponding package manager modules during their instantiation. This system allows the modules to access the CSAR's content at any time.

### 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 becomes a more encapsulated CSAR whose level of access to Internet will be significantly lower which implements the requirement 7.

## 5.3 Language Modules

This section will describe the language modules. Since the framework is initially oriented to easy extensibility, an abstract model for the modules will be defined so, that new modules can be added by implementing this model. This abstract model serves to implement requirement 5. The realization of the Bash and Ansible modules will be provided at the end of the section which responds to requirement 4.

### 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 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 creating such definitions.

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 language Bash was implemented. The Bash module should accept only files written on the Bash language. To chose such files some signs inherent to all Bash scripts can be used. 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-get* module described later. The Bash module must provide a capability for the given package to create a definition of a TOSCA node which uses the Bash language to install the package. Such a Bash TOSCA node is defined by a Package Type, an Implementation, an Artifact, and a Script Artifact. The Package Type is a Node Type with an "*install*" operation and a name received from the *getNodeName* function. The Package Implementation is a Node Type Implementation which refers the Package Artifact and the Script Artifact to implement the Package Type's "*install*" operation. The 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 test 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. Listing 5.3 represents those operations. As a sign of Ansible files, the "*.yml*" extension is used, since

**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;
}
```

its playbooks don't contain any specific header.
Creation of an Ansible TOSCA node for a package is a complicated operation. As the first step, the original files should be analyzed to determine the 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 a playbook and a configuration file will be created in a temporary folder. When the installation data are downloaded and added to 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

**Listing 5.3** Ansible proceeding

```
public void proceed()
      throws FileNotFoundException, IOException, JAXBException {
      if (ch == null)
            throw new NullPointerException();
      for (String f : cr.getFiles())
            for (String suf : extensions)
                  if (f.toLowerCase().endsWith(suf.toLowerCase()))
                        if (suf.equals(".zip"))
                              proceedZIP(f);
                        else
                              proceed(f, f);
}

public void proceed(String filename, String source)
      throws FileNotFoundException, IOException, JAXBException {
      for (PacketManager pm : packetManagers)
            pm.proceed(filename, source);
}

private void proceedZIP(String zipfile)
      throws FileNotFoundException, IOException, JAXBException {
      boolean isChanged = false;
      String folder = new File(cr.getFolder() + zipfile).getParent()
            + File.separator + "temp_RR_ansible_folder" + File.separator;
      List<String> files = zip.unZipIt(cr.getFolder() + zipfile, folder);
      for (String file : files)
            if (file.toLowerCase().endsWith("yml"))
                  proceed(folder + file, zipfile);
      if (isChanged) {
            new File(cr.getFolder() + zipfile).delete();
            zip.zipIt(cr.getFolder() + zipfile, folder);
      }
      zip.delete(new File(folder));
}
```

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 and to delete them. Those modules satisfy the requirements 1 and 2. An abstract model will be defined to make the extensibility easier,

**Listing 5.4** Abstract package manager model

```
public abstract class PacketManager {

      // Name of manager
      static public String Name;

      protected Language language;

      protected Control_references cr;

      /**
      * Proceed given file with different source (like archive)
      *
      * @param filename
      * @param source
      * @throws FileNotFoundException
      * @throws IOException
      * @throws JAXBException
      */
      public abstract void proceed(String filename, String source)
            throws FileNotFoundException, IOException, JAXBException;
}
```

which is needed for requirement 5. 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. Its description which is provided in listing 5.4 contains only one function $proceed$ , that finds and eliminates external references, as well as passes the found package names to the package handler.

### 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 "$apt$-$get\ install$" 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 a complicated task too. It's worth mentioning that the processing uses a simple state machine and regular expression from the *java.util.regex* package.

## 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 a list with dependent packages for the given package.

### Package Downloading

The download operation is performed using the recursive function *getPackage*. Download of installation data will satisfy the requirement 3The arguments of the function are described shortly.

- *language* is a reference to the language module which has accepted the original artifact.
- *packet* 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.

This function downloads packages, calls the language's function *createTOSCA_Node* to create the TOSCA node for the package and the topology handler's functions *addDependencyToPacket* or *addDependencyToArtifact* to update the topology. Then this function calls itself recursively for all depended packages. After those operations, a dependencies three for the *packet* will be built.
The command *apt-get download* package is used for download the package. If the process fails, a user input is provided to solve the problem. The user will be able to rename the package, ignore it or even break the processing.

### List with Dependent Packages

To obtain the dependent packages the $getDependensies$ function was developed. It becomes a *package* as an argument and uses the command $apt\text{-}cache\ depends$ *package* 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.

### Type of Dependency

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

## 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 a Service Templates, where this Node Template was defined.

**Listing 5.5** 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);
```

### Updating Service Templates

To update Service Templates two functions are provided.

- $addDependencyToPacket(sourcePacket, targetPacket, dependencyType)$ generates a dependency between two package nodes.
- $addDependencyToArtifact(sourceArtifact, targetPacket)$ generates a dependency between the original node and a package node.

Both functions find all Node Templates which instantiate the given $sourcePacket$ 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 the $targetPacket$ 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 Relationship Template references both Node Templates. The type of dependency is the value of the $dependencyType$ for the $addDependencyToPacket$ function and the $preDependsOn$ for the $addDependencyToArtifact$.
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 5.5. Together with definitions from the preprocessing stage and definitions made by languages this satisfy the requirement 6.

## 5.7 Environment

In this section the environment necessary for the correct work of the developed software will be described. Since the framework is written in Java, to start it a Java Development Kit version 1.8 or above is needed. Additionally, the apt-get package manager must be installed to properly download packages and identify dependencies between them. If a user want to download packages for specific architecture the package manager must be setup to access this architecture's repository.

# 6 Add New Package Manager 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 a common information about the *aptitude*.
In section 6.2 the module is implemented and in section 6.3 is integrated into the Bash language module.

## 6.1 Aptitude

This section describes the *aptitude* package manager. Like to the *apt-get*, the *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 (An example in Figure 6.1). An another advantage compared to the *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.

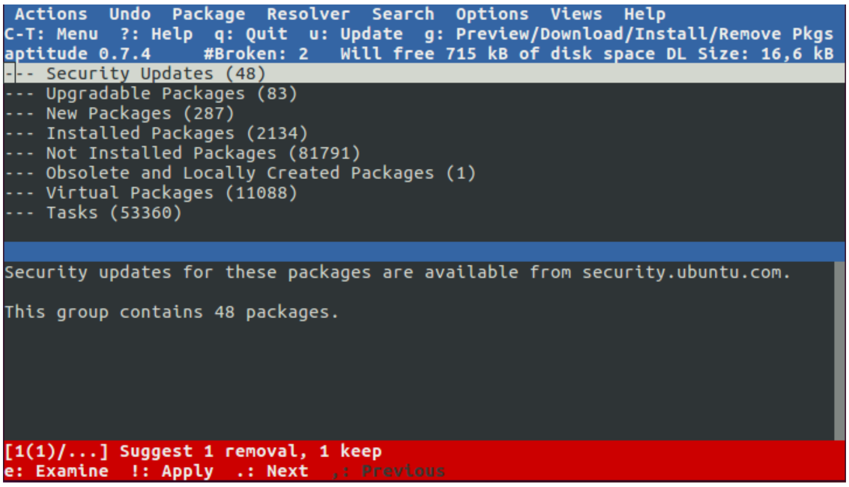


**Figure 6.1:** The command line visual interface for the *aptitude* package manager.

**Listing 6.1** The *aptitude* inherited from the *PackageManager* abstract class

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

## 6.2 Implementing the New Package Manager 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 *ansible* 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.

**Listing 6.3** The aptitude *proceed* function

```
@Override
public void proceed(String filename, String source)
      throws FileNotFoundException, IOException, JAXBException {
      if (ch == null)
            throw new NullPointerException();
      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);
}
```

Now focus on the *proceed* function. A line-by-line file analyzer is needed. It must modify the data and in the case of modification, the entire file should be rewritten. The *isChanged* variable indicates that the file must be rewritten with a new content from the *newFile* variable. Now an *aptitude* line parser will be implemented, which reads a line from the *line* variable and stores it or it's changed version to the *newFile* variable. If the data is changed, then the *isChanged* variable must be set to true. Any *ansible* 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*.
During the parsing which is described 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.

## 6.3 Integrating Aptitude into the Bash Module

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 *packetManagers* variable). This is done by the Bash's constructor with the command: "*packetManagers.add(new PM_aptitude(this, ch));*".
After these operation the new module is ready to identify and resolve external references which use aptitude to install packages.

**Listing 6.4** The aptitude 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]);
                  ch.getPackage(language, words[packet], source);
            }
      }
      newFile += "#//References resolver//" + line + '\n';
}
else
      newFile += line + '\n';
```

# 7 Validation

In this chapter, the developed framework will be validated. An input CSAR will be described in section 7.1. The processing by the framework is described in section 7.2. The output CSAR will be added to and displayed by Winery in section 7.3. Generated Artifacts will be checked in section 7.4.

## 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 7.2. The service uses a server virtualization environment named $vSphere$ (the $VSphere\_5.5$ node). In the environment works the $Ubuntu$ virtual server (the $Ubuntu$-14.04-$VM$ node). The $Ubuntu$ hosts two applications: $Python$ (the $Python\_2.7$) and the $Flink$ $Simple$ (the $Flink\_Simple\_1.0.3$ node). An analyze shows two external references. The $Python$ node installs the python package and the $Flink\ Simple$ node - the Java package. 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.

## 7.2 Processing

To start the framework an Java environment is used. After the start, a user should enter the input CSAR name, the output CSAR name, and the architecture. After that, the framework works fully automatically, analyzing the artifacts and resolving any external references. Figure 7.1 provides the example.

## 7.3 Displaying with Winery

Winery was installed to test the correctness of the output CSAR. This is an environment for the development of TOSCA systems and is useful for checking the results. The input CSAR's representation by Winery is displayed in Figure 7.2. Those external references will be resolved by the framework and exchanged by new nodes in output CSAR.

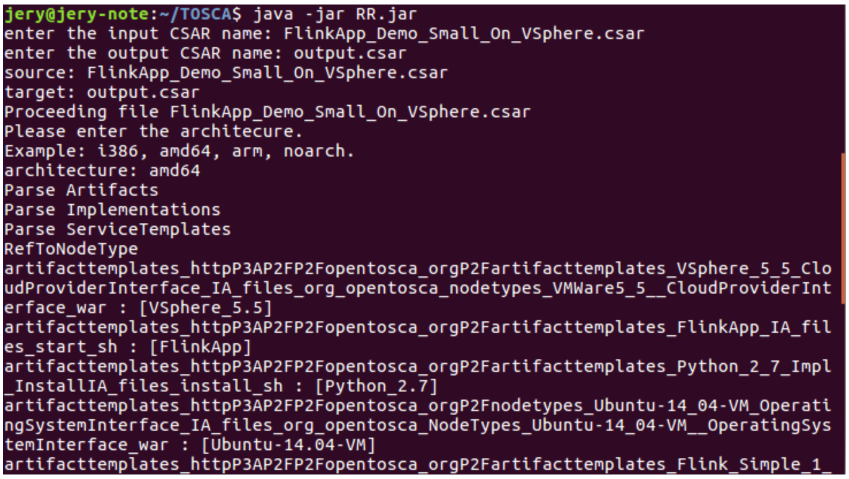


**Figure 7.1:** Processing by the framework.

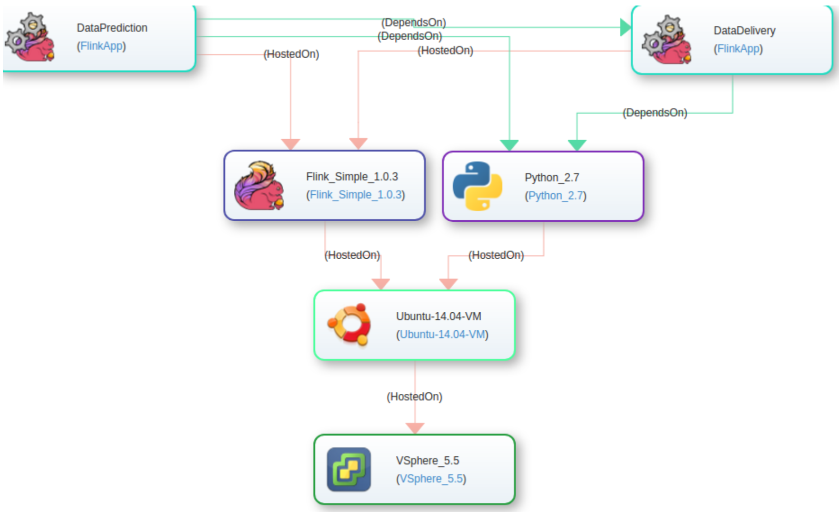


**Figure 7.2:** Source CSAR represented by $Winery$.

## Add to Winery

The output CSAR is added to Winery. Due to a significant increase in size, this can be a fairly lengthy procedure. It was only six nodes in the input CSAR, but after the processing, the output CSAR contains more than 100 of nodes. During the addition to Winery, the CSAR's syntax is tested. In a case of errors, messages will be displayed.

### Display by Winery

The output CSAR is displayed by Winery. Due to the high number of nodes, the processing can take a long time. At the time, the internal references are validated. If something was defined not properly, these erroneous nodes or links between them will not be displayed. The representation of the output CSAR is shown on Figure 7.3 (Only the part of the CSAR is visible).
The structure seems very difficult to follow. To verify the topology some nodes was moved manually. Figure 7.4 displays the result. The correctness of dependencies was verified by checking several nodes with the *apt-cache depends* command. By opening the content of the new nodes, it was verified, that there are right artifacts.

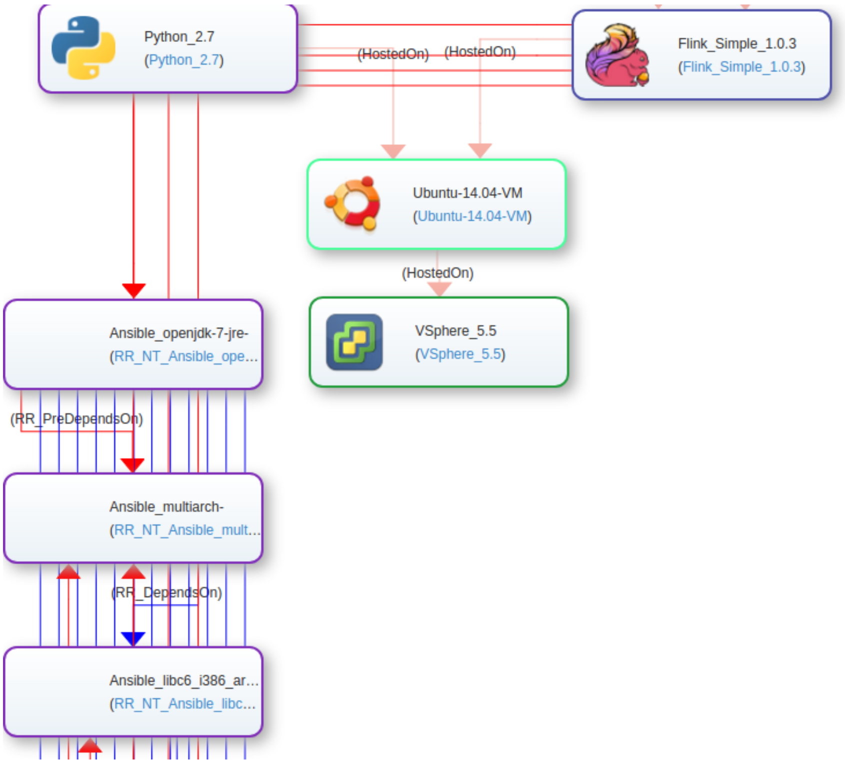


**Figure 7.3:** The output CSAR represented by *Winery*.

## 7.4 Validate Artifacts

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

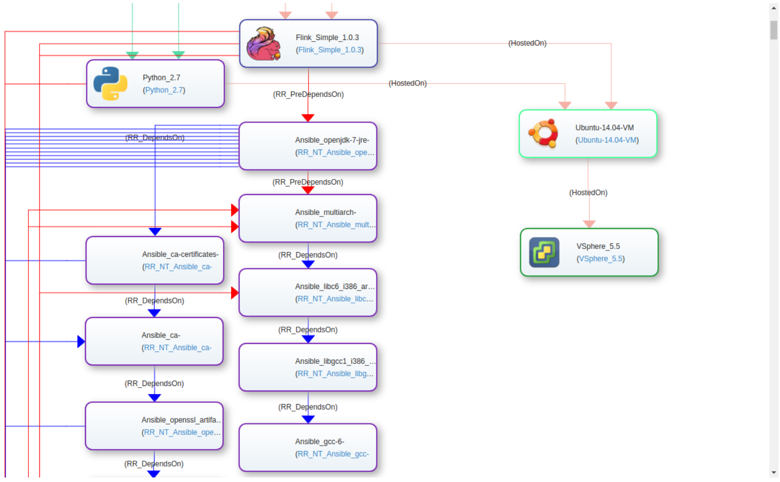


**Figure 7.4:** The output CSAR represented by $Winery$, some nodes moved manually.

**Listing 7.1** Check Bash installation script

```
user@user:~$ sudo RR_python2_7-minimal.sh
(Reading database ... 286091 files and directories currently installed.)
Preparing to unpack python2_7-minimal.deb ...
Unpacking python2.7-minimal (2.7.12-1ubuntu0~16.04.1) over
    (2.7.12-1ubuntu0~16.04.1) ...
Setting up python2.7-minimal (2.7.12-1ubuntu0~16.04.1) ...
Processing triggers for man-db (2.7.5-1) ...
```

### Validate Bash Scripts

Since the 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. An example of the $python2.7$ installation is presented in listing 7.1. The process ended without any warnings or errors, which means that it was completed successfully. This way any Bash installation script can be checked.

### Validate Ansible Playbooks

To check an ansible playbook manually we need to extract the zip file containing the playbook. During the regular execution, this work will be done by the runtime environment. The call of the ansible runtime which proceeds the playbook is a simple procedure too. An example is provided in Figure 7.5. $Ok$ signals that the installation was completed successfully.


**Figure 7.5:** An ansible playbook's execution process

# 8 Summary

External references can negatively affect the performance of Cloud applications. Unfortunately, many TOSCA applications access external sources to install various packages during a deployment. If we have a high level of information security or a limited access to the Internet, these external dependencies lead to a lot of problems which can be solved by encapsulation. The purpose of this work was to develop the software for the elimination of such dependencies and the encapsulation of a TOSCA application.
The software was developed in Java language, which ensures its portability, ease of maintenance and extensibility. To enable the ability to handle new types of external dependencies, the software was implemented in the form of a modular framework with separate modules for processing of languages and package managers. This allows a user to add their new handlers for package managers and languages easily. In the first version, the framework handled only *Bash* scripts which use the *apt-get* package manager. To check the simplicity of the extensibility, the processing of *Bash* scripts with the *aptitude* package manager and *Ansible* playbooks with the *apt* package manager was added.
The framework handles a CSAR as follows. The structure of the CSAR is analyzed to determine the internal dependencies between artifacts and Node Templates. Then each language module selects artifacts written in the language. All such artifacts are transferred to the package manager modules of the language for processing. They find external dependencies, remove them and pass the names of the required packages to the package handler. It loads each package along with all its dependencies and sends information about them to the topology handler, which creates TOSCA nodes for these packages and defines TOSCA dependencies from the original nodes to the new ones. Later, a runtime environment can analyze these dependencies 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 a module which can be added into the framework, how to implement its basic functions, pass data to the packet handler, and connect the module to *Bash* language.
In the end, the results of the framework were validated. The output SCARs were visualized and analyzed with the help of Winery. Generated artifacts were verified and executed.
As a result, the framework that eliminates external dependencies in a CSAR and satisfies the requirements was obtained. It handles *Bash* language with *aptitude* and *apt-get* package managers and *Ansible* language with *apt* package managers. The framework can be easily expanded to handle additional types of external references.

# 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. 27).

[CFE] CFEngine community. *CFEngine*. URL: https://cfengine.com/learn/learnhow-to-write-cfengine-policy/ (cit. on p. 27).

[Chr07] T. Chris. *OPIUM: Optimal Package Install/Uninstall Manager*. Mar. 15, 2007 (cit. on p. 23).

[Ecl] Eclipse Stardust Foundation. *Stardust*. URL: https://projects.eclipse.org/projects/soa.stardust (cit. on p. 22).

[Ham11] N. Hamilton. *The A-Z of Programming Languages: BASH/Bourne-Again Shell*. Computerworld: 2. 2011. URL: https://www.computerworld.com.au/article/222764/a-z_programming_languages_bash_bourne-again_shell/?pp=2&fp=16&fpid=1 (cit. on p. 26).

[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, 21).

[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. 22).

[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. 16).

[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. 27).

[Nat] National Institute of Standards and Technology. URL: https://www.nist.gov/ (cit. on p. 15).

[OAS] OASIS. *Organization for the Advancement of Structured Information Standards*. URL: https://www.oasis-open.org/ (cit. on p. 17).

[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 pp. 17, 19).

[Ope16] OpenTOSCA community. *OpenTOSCA for the 4th Industrial Revolution*. University of Stuttgart. 2016. URL: http://www.opentosca.org/demos/smart-prediction-demo/index.htm (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. 15, 16).

[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. 26).

[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. 22).

[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