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:	Dipl.-Inf. Michael Zimmermann
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. **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplication (TOSCA) provides a whole range of tools for managing and automation of cloud application. The TOSCA standard adds an additional level of abstraction to the cloud applications, in other words, a layer between the external interfaces of cloud application and 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. The University of Stuttgart implemented this specification in the runtime environment named OpenTOSCA. Description of a cloud application is stored in the **C**loud **S**ervice **AR**chive (CSAR), which contains all components necessary for a cloud application life-cycle.

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 cloud application. And if 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. During this work a software solution which will eliminate external dependencies in CSAR, resupply them with all packages necessary for deployment and also change the internal structure to display the achieved self-containment will be developed and implemented. For example, all commonly used "apt-get install" commands, which download and install packages, must be removed. Appropriate package must be downloaded and integrated into CSAR structure. Furthermore, all depended packages needed for new packages must also be added recursively.

This Document considers the concept and architecture for mentioned framework. In addition some aspects of implementation will be described.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

API **A**pplication **P**rogramming **I**nterface. 17

CSAR **C**loud **S**ervice **AR**chive. 25

TOSCA **T**opology and **O**rchestration **S**pecification for **C**loud **A**pplication. 17

1 Introduction

Cloud Applications Market increases with great speed. Globally annual growth is about 15%. [Sta16] Furthermore observed the growth in the number of firms, which are using Cloud applications. And that are 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, then to maintain an own server. As well as it is also easier and cheaper to send a small package by mail, than to keep your own car (server) and driver (administrator) for a rare traffic.
The growing popularity of cloud applications makes the automation and the ease of management increasingly important. Under the management is understood the deployment, administration, maintenance and the final roll-off of cloud applications.
The common problem of cloud applications is *affection*. 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 quickly move a cloud application to the more suitable provider is a key to the development of competition and reducing the cost of maintenance.
Topology and **O**rchestration **S**pecification for **C**loud **A**pplication (TOSCA) [13] provides an opportunity to solve this problem. TOSCA defines the language, which allows describing cloud application and their 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 cloud application is stored into **C**loud **S**ervice **AR**chive (CSAR). This archive contains the description of the cloud application, its external functions and internal dependencies, and the data needed for the deployment and operation.
OpenTOSCA [Stu13] is an open source ecosystem (runtime environment) for TOSCA standard developed in University of Stuttgart, which is constantly improved and expanded. OpenTOSCA processes data in CSAR format and performs the actions specified in it.
Often these actions contain links to external packages and programs necessary for deployment of the cloud application, which will be subsequently downloaded over the Internet. This downloads can add expenses to the time required to download packages, money spent on rent an idle server and Internet traffic for megabytes of pre-known

data. If a cloud application consists of only one deployed server, this may mean a few seconds of delay. But when an application deploys a large number of linked servers (cloud system), the costs can increase significantly.
Other problems of external dependencies are security and stability. To ensure the security of information, some firms restrict the Internet access. 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 cloud application with external dependencies in such networks may well not succeed.
To solve these problems a software solution for resolving external dependencies in CSARs will be developed in implemented during this work. This software will analyze the 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) and adding them to the CSAR's structure. The simplest example is to find in given CSAR all the commands like "apt-get install package", delete this command, download the package and all depended packages and add them to the CSAR.
This software must be easily expanded (in other words - that will 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 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 is structured as follows:

Chapter 2 – Basis: This chapter explains the basic terms of this work. These 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: Here are clarified requirements for the framework.

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

Chapter 5 – Implementation: This chapter contains the description of the implementation. It explains the design and development of individual components of the framework.

Chapter 6 – Add new package manager module: New package manager will be added in this chapter, to proof ease of extensibility.

Chapter 7 – Check: Output of the framework will be checked here.

Chapter 8 – Summary Summarize the results of the work.

2 Basis

In this chapter, the common terms will be explained.

2.1 Cloud Computing and Cloud application

2.1.1 Definitions

Unfortunately, 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) is commonly used. This definition appropriately describes the concept of cloud computing used in this paper, and therefore this definition will be used and presented below.

Definition 2.1.1 (Cloud computing)
Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [Pet11]

Also, the are no generally accepted definitions of cloud application, but it can be obtained from the definition of cloud computing.

Definition 2.1.2 (Cloud application)
A Cloud application is an application that is executed according to a cloud computing model. [tec]

In addition, a short definition of cloud system will be provided.

Definition 2.1.3 (Cloud system)
Composite cloud applications which consists of multiple small application will ba called a cloud system.

An owner of physical platform, where cloud computing takes place is called a *provider*. An owner of cloud application, renting a provider's platform is called a *user*.

Service models

NIST distinguishes between three types of service models.

- Software as a Service (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.
- Platform as a Service (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.
- Infrastructure as a Service (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).

Deployment models

Similarly, NIST distinguishes between four types of deployment models.

- Private cloud. The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.
- Community cloud. The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the

community, a third party, or some combination of them, and it may exist on or off premises.

- Public cloud. The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.
- Hybrid cloud. The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds).

2.1.2 Usage

Now cloud computing and application can be found everywhere, and they 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 service uses a single one physical server and this server will be disabled, then entire service will be completely unavailable too. But if 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.
Usually, a user doesn't have direct access to the infrastructure (servers and operating systems), he uses only 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, on the other hand, it becomes more difficult to redeploy an application to another provider.

2.2 Topology and Orchestration Specification for Cloud Applications (TOSCA)

Definition

The OASIS [OAS] Topology and Orchestration Specification for Cloud Applications (TOSCA) standard provides a new way to enable portable automated deployment and management of a cloud applications. TOSCA describes the structure of applications as topology containing their components and relationships between them. TOSCA application is a cloud application described according to TOSCA standard. TOSCA can be used not only for describing all stages of a cloud application life-cycle but also 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

TOSCA specification provides a language to describe service components (described in section Service models) and relationships between them using a Service Topology. In additional it defines the management procedures which create or modify services using orchestration processes.
Descriptions of the TOSCA's main components used in this work is provided below.

- $Service\ Template$ is the main component in TOSCA structure. It contains an information about structure (Topology Template) and interfaces (Plans) of the cloud application.
- A $Plan$ provides an interface to manage cloud application. These components combine management capabilities to create higher-level management tasks, which can then be executed fully automated to deploy, configure, manage, and operate the application. Plans are started by an external message and call management operations of the nodes in the topology.
- $Topology\ Template$ describes the topology of cloud application, instantiating nodes (Node Templates) and relations between them (Relationship Templates).
- $Node\ Template$ specifies the occurrence of a Node Type as a component of a service.
- $Node\ Type$ defines the properties of such a component and the operations available to manipulate the component.

- *Relationship Template* specifies the occurrence of a Relationship Type as a relationship between Node Templates in a Topology Template. The Relationship Template indicates the elements it connects and the direction of the relationship by defining one source and one target element (in further Source Element and Target Element).
- *Relationship Type* defines the semantics and any properties of the relationship.
- *Artifact* represents the content needed for a management such as executables (e.g. a script, an executable program, an image), a configuration file or data file, or something that might be needed for other executables (e.g. libraries). TOSCA distinguishes two kinds of artifacts: Implementation Artifacts and Deployment Artifacts.
- *Implementation Artifact* represents the executable of an operation described by Node Type.
- *Deployment Artifact* represents the executable for materializing instances of a node.
- *Artifact Type* describes a common type of an artifact: python script, installation package and so on.
- *Artifact Templates* represents information about the artifact. Artifact location and other attendant data are stored here.
- *Node Type Implementation* defines the artifacts needed for implementing the corresponding Node Type. For example, if Node Type contains *deploy* and *shutdown* operations, then Node Type Implementation can contain two Implementation Artifacts with scripts for these operations and one Deployment Artifact with data needed for the deployment.

Types and Templates defining a TOSCA application are stored in definition documents, which have the XML structure.

Usage

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. [OAS13]

2.2.1 CSAR

A **C**loud **S**ervice **AR**chive (CSAR) is used to store the TOSCA application. It is a ZIP-file with ".csar" extension that contains all the data needed for instantiation and management of TOSCA application. These include definition documents, artifacts and so on. In this form, a TOSCA application can be processed by a TOSCA runtime environment.

Structure

The root folder must contain a "Definitions" and a "TOSCA-Metadata" folders. The "Definitions" folder contains definition documents and one of them must define Service Template. The "TOSCA-Metadata" folder must contain TOSCA metadata in form of 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 in the CSAR. The metadata is used by TOSCA runtime environment to correctly proceed given files.

Terms

Input CSAR is a CSAR, which can contain external references and will be processed by the framework.
Output CSAR is a CSAR, which was processed by the framework and doesn't contain external references (at least those that are implemented in the framework).

2.3 OpenTOSCA

OpenTOSCA provides an open source ecosystem for TOSCA applications. This ecosystem consists of three parts: [Stu13]

- OpenTOSCA **Container**, a TOSCA runtime environment
- **Winery**, a graphical modeling TOSCA tool.
- **Vinothek**, a self-service portal for the applications available in the container.

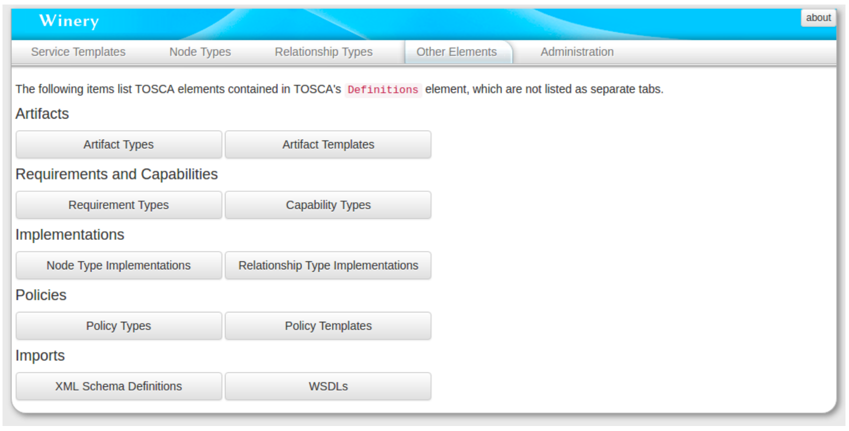
Description for the runtime environment and Winery will be provided in more details.

Runtime environment

The runtime environment enables fully automated plan-based deployment and management of cloud applications in the CSAR container. 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 works under the Tomcat server and therefore visual interface is available in a browser, example in figure 2.1. Winery provides a complete set of functions for creating, editing and deleting all elements of the TOSCA topology. An example of TOSCA topology is presented in figure 2.2.



The following items list TOSCA elements contained in TOSCA's `Definitions` element, which are not listed as separate tabs.

Figure 2.1: Visual interface for $Winery$.

2.4 Package management

This section will describe the package management process and basic concepts for this area.

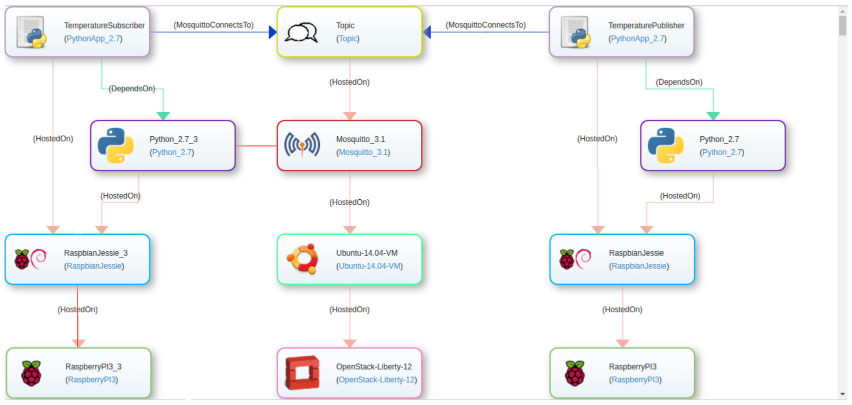

TemperatureSubscriber (PythonApp_2.7)
(MosquittoConnectsTo)
Topic (Topic)
(MosquittoConnectsTo)
TemperaturePublisher (PythonApp_2.7)
(DependsOn)
(HostedOn)
(DependsOn)
(HostedOn)
Python_2.7_3 (Python_2.7)
Mosquitto_3.1 (Mosquitto_3.1)
(HostedOn)
Python_2.7 (Python_2.7)
(HostedOn)
(HostedOn)
(HostedOn)
RaspbianJessie_3 (RaspbianJessie)
Ubuntu-14.04-VM (Ubuntu-14.04-VM)
RaspbianJessie (RaspbianJessie)
(HostedOn)
(HostedOn)
(HostedOn)
RaspberryPI3_3 (RaspberryPI3)
OpenStack-Liberty-12 (OpenStack-Liberty-12)
RaspberryPI3 (RaspberryPI3)

Figure 2.2: TOSCA topology presented by $Winery$.

Package managers

Package Manager is a set of software tools that automate the process of installing, updating, configuring and removing of program components. A package can describe and contain not only a whole program but also a certain component of a large application, hereinafter both will be referred as program components. Package managers are used to managing the database of packages, their dependencies and versions, to prevent erroneous installation of programs and missing dependencies.

Packages

A package is usually an archive containing both data for installation of the program component and a set of metadata like name, function, version, producer and list of dependencies.

Dynamic libraries

Computer systems which rely on dynamic library linking, share executable libraries of machine instructions across packages and applications. In these systems, complex relationships between different packages requiring different versions of libraries result in a challenge colloquially known as "dependency hell". Good package management is vital on these systems.

Repository

To give users more control over the kinds of software that they are allowing to be installed on their system, software is often downloaded from a number of software repositories. By default in Unix systems, a package manager uses official repositories appropriate for the operating system and the device architecture, but it's possible to use additional repositories, like third-party repositories or repositories for another architecture.

Dependencies

Package managers distinguish between two types of dependencies: $required$ and $preRequired$.
Dependency $package1$ $required$ $package2$ indicates that $package2$ must be installed for proper operation of $package1$.
Dependency $package1$ $preRequired$ $package2$ indicates that $package2$ must be installed for proper installation of $package1$.
An example for obtaining the dependency list for the Python package is shown in Listing 2.1.

Listing 2.1: Example of using *apt-cache* to obtain dependency list for package python

```
user@user:~$ apt-cache depends python
python
PreDepends: python-minimal
Depends: python2.7
Depends: libpython-stdlib
Conflicts: <python-central>
Breaks: update-manager-core
Suggests: python-doc
Suggests: python-tk
Replaces: python-dev
```

Dependency tree

In the example above $package2$ is needed for $package1$, but $package2$ itself can require additional packages. A structure describing all needed packages and dependencies between them for given root-package is called a dependency tree. Dependency type $required$ can lead to cycles in a dependency tree, which differs them from the normal tree graph structures.

2.5 Languages for package management automation

To automate the package management, various languages are used. Those languages will be described, which will be used in the framework.

2.5.1 Bash

Bash is a Unix shell and command language written as a free software. Bash is a command processor that typically runs in a text window, where the user types commands that cause actions. Bash can also read and execute commands from a file, called a script. [wikb]

2.5.2 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. [wika]

3 Requirements

Since the main purpose of the developed framework is to Resolve References, further the RR can be used as an abbreviation. RR should eliminate external dependencies in a TOSCA topology represented by a CSAR file. RR must be easily extendable to provide the ability to eliminate a large number of dependency types.
As a first step, a minimal configuration which handles $Bash$ language with the apt-get package manager and $Ansible$ language with the apt package manager will be developed. These software handlers of languages and package managers will be called language modules and package manager modules. As an example, the $Bash$ and apt-get modules will remove package installation commands from bash-scripts (apt-$get\ install\ package$). Then both the $package$ itself and all the depended packages from his dependencies tree will be downloaded. It is also necessary to update the topology of the TOSCA, by adding new nodes and dependencies. To do so, common definitions will be added, like Relationship Types and Artifact Types. Then new nodes will be defined by Node Types, Node Type Implementations, Artifacts Templates, and instantiated by Node Templates. Relations between nodes will be instantiated by Relationship Templates. These Templates must be added to the right Service templates, where the nodes containing external references are instantiated. To find the Service Templates and Node Types corresponding to a certain artifact, it can be useful to apply preprocessing to the entire TOSCA topology. After implementing the minimal configuration, it should be easy to add more language modules and package manager modules, like $Aptitude$ for Bash or completely new language like $Chef$. In order to proof the correctness of the corresponding TOSCA topology, Winery described in section 2.3 will be used.

Stages of the processing

Here an example is provided, representing how the framework should work.

- Begin
  An input CSAR will be extracted.
- Preprocessing
  During preprocessing stage, RR needs to analyze internal references. In additional,

common Tosca definitions for artifacts and relations between packages will be added.

- Processing with language modules
  Each file from the input CSAR will be processed by Language modules.
- Processing with packet manager modules
  If the file belongs to an Language, it will be processed by the packet manager module belonging to the Language to find and resolve external references. Package name from this reference will be moved forward.
- Package handling
  Using the package name the package will be downloaded and TOSCA definitions created. These actions will be recursively repeated for all dependent packages, creating the dependency tree in the TOSCA topology.
- Topology handling
  Using information about internal references and dependencies the TOSCA Topology will be updated by creating new Node and Reference Templates.
- End
  Meta-file should be updated and all data packed back to the CSAR.

These steps will be represented by the modules described in section 4.2 and implemented in chapter 5.

Result

As a result of the 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 additional, the topology for the packages must be mirrored from the package manager's database to the TOSCA topology.

4 Concept and Architecture

4.1 Concept

In this section, the main concept of this work will be described. The general structure of framework is represented in diagram 4.1.

4.1.1 Analysis existing TOSCA-Topology

To properly update the TOSCA topology, it is necessary to add references from the nodes where external references were to newly created nodes, which resolve the external references. According to the TOSCA standard, only references between Node Templates in the same Service Template can be created. That means that each Node Template, which uses artifacts with external references must be found. Furthermore, Service Template where the Node Templates are instantiated must be found 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 a simple references chain can be built:

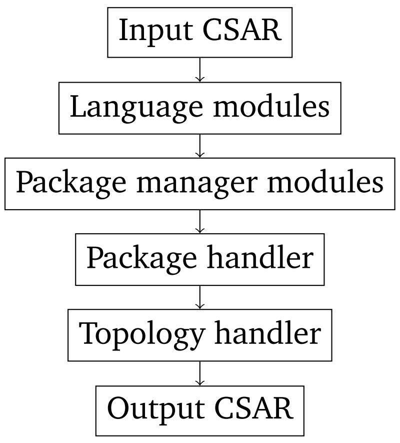


Figure 4.1: General description of the framework's work flow

$Artifact \rightarrow Artifact\ Template \rightarrow Node\ Type\ Implementation \rightarrow Node\ Type \rightarrow Node\ Template \rightarrow Service\ Template$
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.

Thus structure can be described by a tree with an Artifact as the root, and Service Templates as leaves (The example is on figure 4.2) and will be called the internal dependencies tree.

An additional problem is in the reference between a Node Type and a Node Type Implementation. Node Type can have several implementations, but which one will be used will be determined only during the deployment. The chosen solution to this problem is to use each Node Type Implementation in hope, that they will not conflict. The following steps can be executed during the preprocessing, to build the internal dependencies tree.

- Find all Artifact Templates to build references from Artifacts to Artifact Templates.
- Find all Node Type Implementations. Because they contain references both to the Node Type and to the Artifact Templates, then the dependency from Artifact to Node Types can be built.
- Find all Service Templates and in them contained Node Templates. 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 can be built (with references $Artifact \rightarrow Node\ Template$ and $Artifact \rightarrow Service\ Template$).

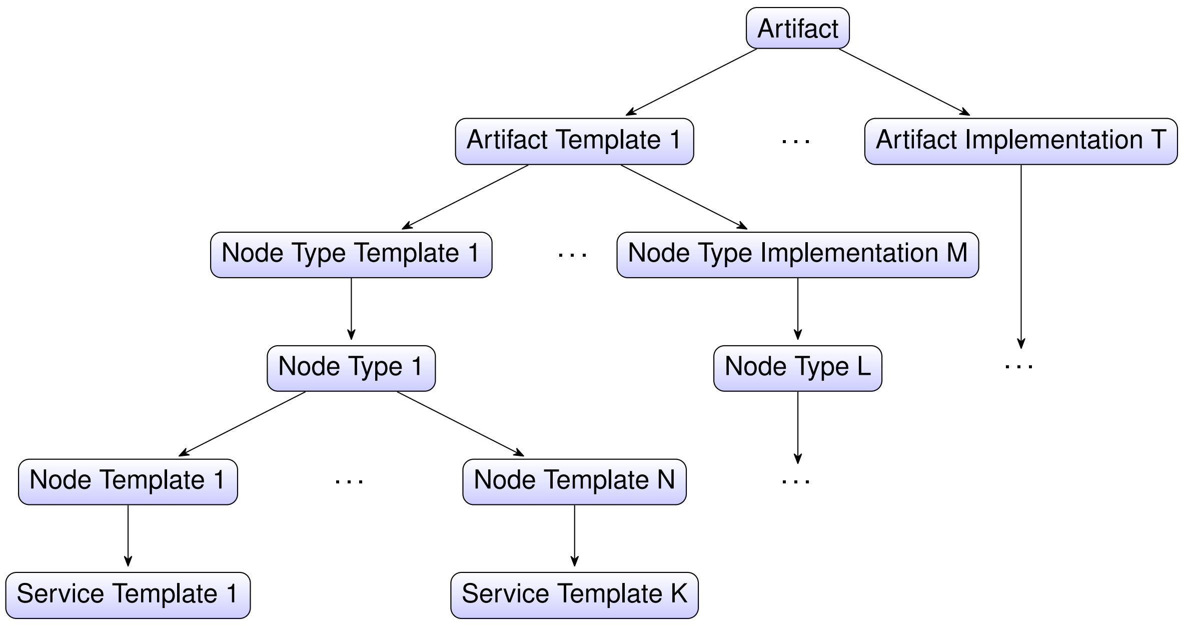


Figure 4.2: An example tree describing how to find Service Templates and Node Templates for a given script

Listing 4.1 Unreadable bash script

```
#!/bin/bash
set line = abcdefgijklmnoprst
set word1 = ${line:0:1}${line:14:1}${line:17:1}
set word2 = ${line:6:1}${line:4:1}${line:17:1}
$word1-$word2 install package
```

4.1.2 Analysis for external references

Unfortunately, it is impossible to identify all possible external references, even when one language and one package manager are used (example in the listing 4.1).

Since this work is aimed at creating of the easily expanded and supplemented tool, initially only basic usage of package managers will be considered. Ease of adding modules to the framework will proof the correctness of architecture.
At the beginning, the most popular combination will be implemented: the $bash$ script with the $apt - get$ package manager. This simple and powerful tool allows to install, delete or update the set of packages in one line. After the modules for this combination will be implemented, new languages and package managers should be added.

4.1.3 Representing downloaded packages in TOSCA-Topology

A package node denotes to the defined and instantiated element of TOSCA topology, the purpose of which is to install the package. The adding of new package nodes to TOSCA topology can be divided into several steps.

- Add definitions for common elements, like Artifact Types or Relationship Types. This can be done once at the preprocessing stage.
- The package node main definition will be represented by a Node Type.
- Artifacts (The downloaded data and the installation script) will be referenced by Artifact Templates.
- Node Type Implementation will combine the artifacts.
- Node Template will instantiate the package node in the corresponding Service Templates. To determine corresponding Service Template the preprocessing described in the section Analysis existing TOSCA-Topology will be used.
- Reference Template will provide topology information, allowing the observer (a user or a runtime environment) to determine, for which nodes the package must be installed. References will be created from the Node Template needing the package to Node Template of created package nodes.

4.1.4 Determining architecture of a final platform

An another problem appears during choosing 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, on which architecture which node will be deployed. There are many pitfalls here.
A single Service Template can use several physical devices with different architectures. One Implementation Artifacts can be referred by different Node Types and Node Templates, instantiated on different platforms. 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 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 a whole CSAR.
- *Artifact specific architecture*
  The architecture will be defined separately for each artifact.

Analysis of methods

Unfortunately, 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, a chaos can be produced by mirroring these different packages with different versions to the TOSCA-topology. The only robust solution seems to be to create for each installed package a set of archives (one archive for one architecture), containing 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 obligate a user to analyze each artifact and decide on which architecture it will be executed. 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 be easily 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.).

4.1.5 Extensibility

The framework should handle different languages, each of them can support various package managers. This principle can be illustrated by a figure 4.3.

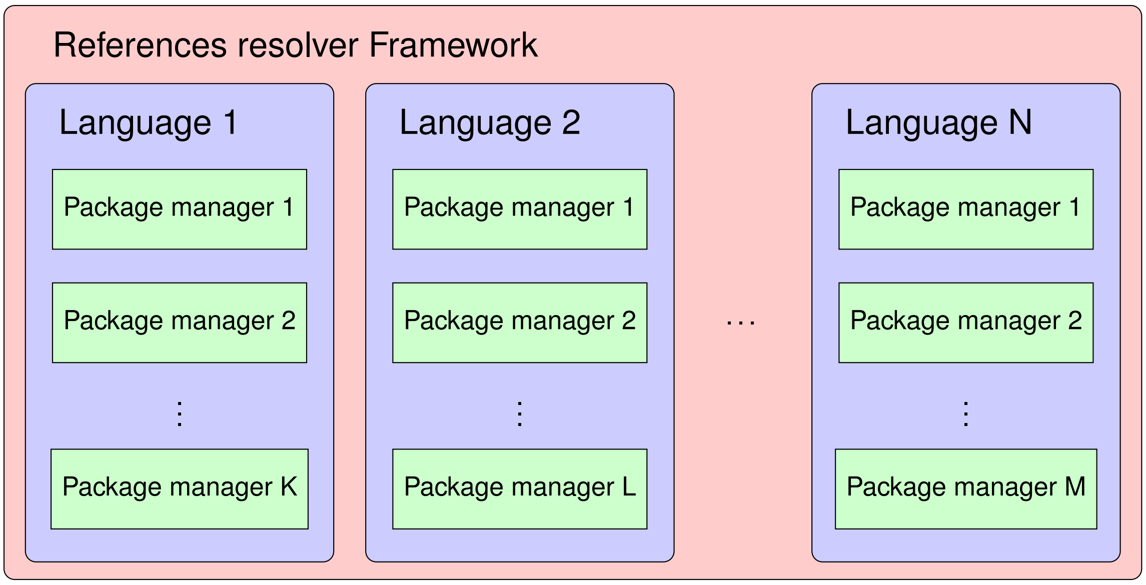


Figure 4.3: An example scheme representing several language modules containing package manager modules

4.1.6 Result's checking

Checking the output of the framework is an important stage in the development of the program. It is necessary to verify both the overall validity of the output CSAR and the possibility to deploy generated package nodes. To test for overall correctness it is possible to use *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 manual by starting corresponding commands.

4.2 Architecture

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

4.2.1 CSAR handler

The CSAR handler provides access to CSAR and maintains it's consistency. It describes the process of adding new files (to handle the metadata), decompression, architecture processing, etcetera.

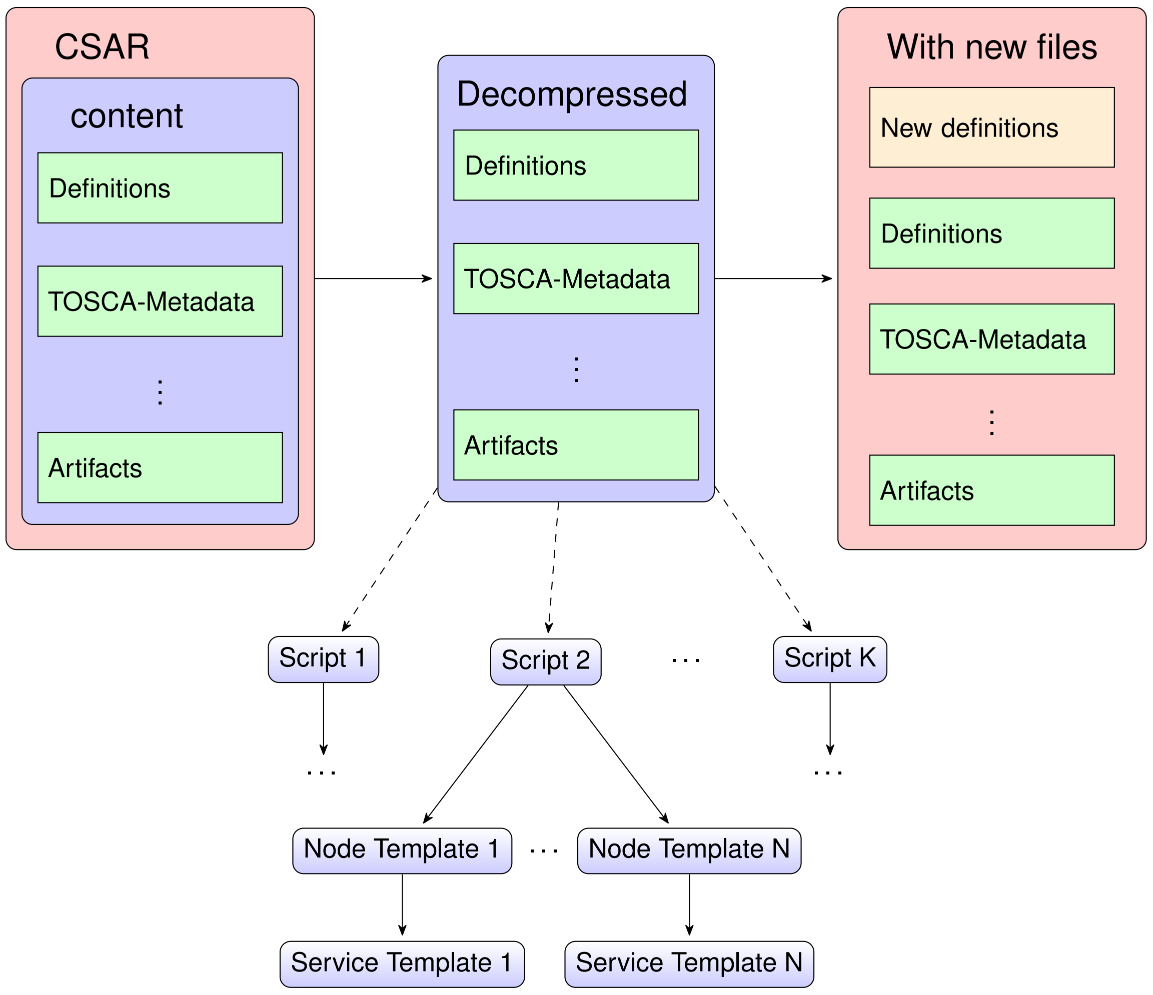


Figure 4.4: Preprocessing: decompression, adding files and generating dependencies

4.2.2 References resolver

This is the main element, the execution of which can be divided into three stages: *preprocessing*, *processing*, *finish*.
At the begin the *preprocessing* will be executed for decompression, adding files (The list of common files, which are added during preprocessing is described in the section 4.1.3) and generating internal dependencies trees (the generation is described in the section 4.1.1). Figure 4.4 illustrates the stages of the preprocessing.

During the *processing*, all *language modules* will be activated, which are described in the next section.
To finish the work all results will be packed back to the archive during the *finish* stage.

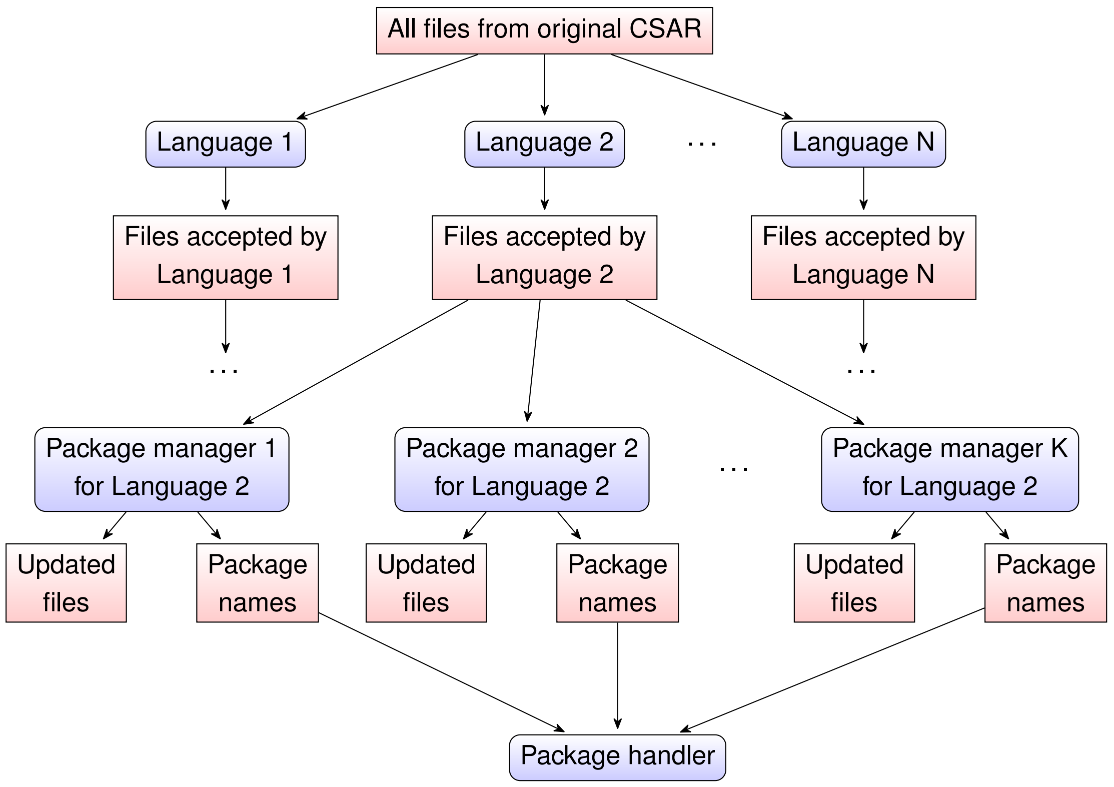

All files from original CSAR
Language 1
Language 2
...
Language N
Files accepted by Language 1
Files accepted by Language 2
Files accepted by Language N
...
...
Package manager 1 for Language 2
Package manager 2 for Language 2
...
Package manager K for Language 2
Updated files
Package names
Updated files
Package names
Updated files
Package names
Package handler

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

4.2.3 Language modules

Each *language module* describes one language and helps to choose files written in the language. As already mentioned above, during *processing* stage a *language module* analyzes all files one by one and checks its belonging to the specified 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 can pass through only files with ".*sh*" extension and starting with the "#!/*bin*/*bash*" line.

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 4.5 illustrates data flow between language modules, package manager modules, and package handler.

4.2.5 Package Handler

The *package handler* becomes a package name, downloads a data for the package installation, transfers the package name to the *topology handler* and recursively repeats the actions for all depended packages.

4.2.6 Topology Handler

Topology handler adds a package to the topology. This includes adding new files and updating existing files. Necessary steps were described in section 4.1.3.

5 Implementation

This chapter provides the information about the implementation of the framework and his elements (or modules), which was described in chapter 4. The Java language was chosen, because of his simplicity and strength. In the Java language, the modules and elements are represented by classes.

5.1 Global modules

This section describes the modules used throughout the whole framework's execution.

CSAR handler

CSAR handler provides an interface to access the CSAR content and stores information about files associated with it. For example:

- The temp extraction folder.
- The list of files from the CSAR.
- The meta-file entry.
- The architecture of The target platform.

All this data are encapsulated into CSAR handler. To access them public the functions can be used.

- $unpack$ and $pack$ extract the CSAR to the temp folder and pack the temp folder back to the CSAR. These functions use the $ZIP\ handler$ module described below.
- $getFiles$ returns the list containing all the files presented in the CSAR.
- $getFolder$ returns the path to the folder, where the CSAR was extracted.
- $getArchitecture$ returns the architecture used for the CSAR.
- $addFileToMeta$ adds an information about the new file to the meta-data.

Listing 5.1 The common functions to handle zip archives

```
/**
* Unzip it
*
* @param zipFile input zip file name
* @param outputFolder output folder
*/
static public List<String> unZipIt(String zipFile, String outputFolder);

/**
* Zip all files in folder
*
* @param zipFile output ZIP file location
* @param folder, containing files to zip
* @throws FileNotFoundException, IOException
*/
static public void zipIt(String zipFile, String folder)
```

Utils

This class provides methods, used by many other modules.

- $createFile(filename, content)$ creates the file with the given content.
- $getPathLength(path)$ returns the deep of the path.
- $correctName(name)$ adapts the name for use by the OpenTOSCA.

Zip handler

This is a small module with strait functionality. It serves to pack and unpack zip archives, which are used by the CSAR standard. To handle archives it was decided to use the package $java.utils.zip$. The functions of archiving and unarchiving are called respectively $zipIt$ and $unZipIt$. The Java declaration of this functions is provided in the listing 5.1

5.2 References resolver

This is the main module which starts by framework startup and is executed into three stages.

Preprocessing

At the preprocessing stage, the CSAR is unpacked, common TOSCA definitions generated and internal dependencies trees build.

Unpacking

To unpack the CSAR the function *unpack* from the CSAR handler is used.

Generating TOSCA Definitions

To generate common TOSCA definitions the *javax.xml.bind* package was chosen. Descriptions for common definitions were created. (A definition defines the element of TOSCA standard. A description is used to create the definition.)

- *DependsOn* and *PreDependsOn* describe Relationship Types for the dependency types between packages.
- *Package Artifact* describes a deployment Artifact Type for a package installation data.
- *Script Artifact* describes an implementation Artifact Type for scripts installing packages.
- *Ansible Playbook* describes a deployment Artifact Type for a package installation via Ansible playbook.

An example description of the *Script Artifact* can be found in the listing 8.1. Each description is presented by a separate Java class.

Build internal dependencies trees

Internal dependencies are mainly used by the Topology Handler. Therefore, these two modules were combined within the one Java class named *Topology Handler*. At the preprocessing stage, it analyses all origin definitions to build internal dependencies trees, as was described in section 4.1.1. To read origin definitions from the XML files the package *org.w3c.dom* was used.

Listing 5.2 The processing stage

```
for (Language l : languages)
      l.proceed(cr);
```

Processing

During this stage, all described language modules are started. Since the language modules are stored in *language* variable, this simple stage can be presented by the listing 5.2.

Finishing

To finish the work the changed data should be packed back to CSAR. The function *pack* from the CSAR handler is used.

5.3 Search for external references

This section will describe the search for external references in the original artifacts. For this purpose serve Language modules and Package manager modules. Since the framework is initially oriented to easy extensibility, abstract models for Language modules and Package manager modules will be defined. New languages and package managers can be added by implementing these models.

Language model

To describe the common functionality and behavior of different language modules, the Language model is used. In the Java, this abstract model is described by an abstract class. The abstract class *Language* is presented in the listing 8.3. The common components for all language modules are:

- The name of the language.
- The set of package manager modules.
- The extensions of files.

And the common functions are:

- *getName* returns the name of this language.

- *getExtensions* returns the list of extensions for this language.
- *proceed* checks all original files and transfers results to package manager modules.
- *getNodeName* returns the name for Node Type, which will install package with this language.
- *createTOSCA_Node* creates the TOSCA definitions for the package. Since the created package nodes must install a package using the same language as the original node, all languages must provide the method for creating the definitions.

Package handler model

Like to languages, an abstract class for package handlers is defined at first. His description contains only one function *proceed* (In the listing 8.4), that finds and eliminates external references, as well as passes the found package names to the package handler.

Bash module implementation

The processing of popular Bash language was implemented. As the signs of belonging to the Bash language, the file extension (".sh" and ".bash") and the first line ("#!/bin/bash") are used. All files which satisfy this conditions are passed to package managers modules, in our case - to the *apt-get* module.
A Bash package node is defined by Node Type, Node Type Implementation, Package Artifact, Script Artifact. This package node will be instantiated later by the topology handler. The definitions are created by *createTOSCA_Node* method presented in the listing 8.5. Consider it in more details. To avoid creating of the same nodes, the names of created nodes are stored in the *created_packages* list. Then the node name is generated using *getNodeName* and TOSCA definitions for this name are created.

Apt-get Bash implementation

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*", comments them out and passes this command's arguments to package handler's public function *getPackage*. The code can be found in the listing 8.6.

Ansible implementation

To test the extensibility of the framework, the Ansible language was added. Since ansible playbooks are often packed to archives, therefore it may be necessary to unpack them first and then analyze the content. Thus, the files are either immediately transferred to the package handler, or they are unzipped first. Listing 8.7 presents these operations. As a sign of the ansible language, the "*.yml*" extension is used, since its playbooks don't contain any specific header.
Creating a TOSCA node for this language is a complicated operation. The basic moments are:

- Analyze original files to determine the ansible configuration (the set of options like username or proxy).
- It can be necessary to complement the configuration using a user input.
- Create the folder with necessary files (the executable *.yml* file and a subfolder with the installation data).
- Pack these files to the Zip file.
- Create TOSCA definitions of the package node. The ansible package node is defined by Node Type, Node Type Implementation, and Ansible Artifact.

Apt implementation

Since the package installation written in the *ansible* language with the *apt* package manager can be described 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.

5.4 Package Handler

Package handler provides an interface for interaction with the package manager of the operating system. It allows to load packages and to determine the type of dependencies between them.

Listing 5.3 The *getPackage* definition

```
/**
* Download package and check its dependency
*
* @param language, language name
* @param packet, package name
* @param listed, list with already included packages
* @param source, name of package or file depending on the package
* @param sourcefile, name of original file contained external reference.
* @throws JAXBException
* @throws IOException
*/
public void getPacket(Language language, String packet, List<String> listed, String
    source, String sourcefile)
```

Package downloading

This operation is performed using one recursive function *getPacket* defined in the listing 5.3. This function downloads packages for the dependency three, calls the language's function *createTOSCA_Node* to create package nodes and the topology handler's functions *addDependencyToPacket* and *addDependencyToArtifact* to update the topology.
The Arguments of the *getPacket* function will be described shortly.

- *language* is used to call the right *createTOSCA_Node* function.
- *packet* is a package name to be downloaded.
- *listed* holds a list with packages already presented in the dependency tree. No need to download them again, but new dependencies will be created.
- *source* defines the parent element in the dependency tree. For the root package that will be the original artifact file, for other packages - the depending package.
- *sourcefile* is the name of the original artifact with external dependencies. This name will be used by the *language* to generate package node and by topology handler to create the dependency.

For downloading the command *apt-get download* package is used. If a download fails then the user input is used to solve the problem.

Listing 5.4 Creating of 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);
```

Dependencies

To determine the dependency type the command *apt-cache depends package* is used. Example output was presented in section 2.4.

5.5 Topology handling

Topology handler serves to update the TOSCA topology. For this purpose the Build internal dependencies trees is executed during preprocessing stage.

Update Service Templates

To update Service Templates two functions are provided.

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

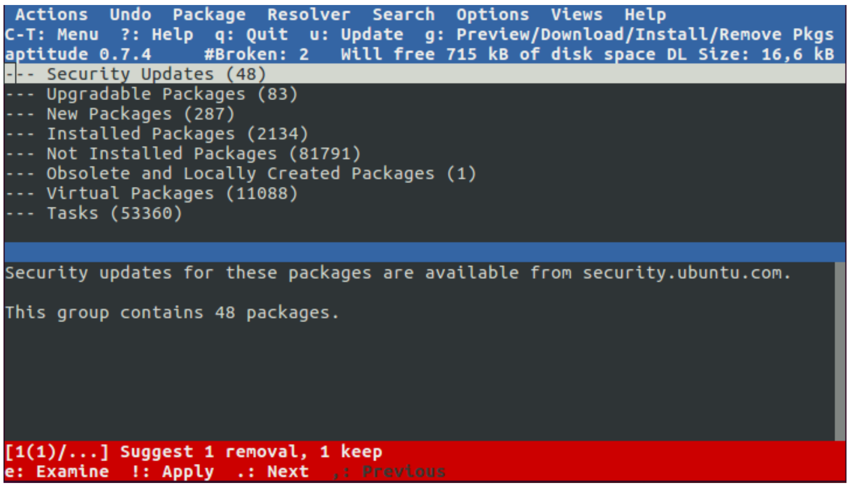
The both functions finds all Node Templates and Service Templates for the given $sourcePacket$ or $sourceArtifact$ using the internal dependencies trees. For each found Node Templates a package node for the $targetPacket$ package is instantiated by creating new Node Template. Then the dependencies between found Node Templates and new Node Templates is created by instantiating Relationship Templates. The type of dependency is the value of the $dependencyType$ for $addDependencyToPacket$ and always the $preDependsOn$ for $addDependencyToArtifact$.
To update existing TOSCA definition the $org.w3c.dom$ and $org.xml.sax$ packages are used. Creating of new Node Template is presented in the listing 5.4.

6 Add new package manager module

This chapter will show extensibility of framework by adding new package manager module for *aptitude* package manager.
Section 6.1 provides common information about the package manager.
In section 6.2 module is implemented and in section 6.3 is integrated into bash language module

6.1 Aptitude

This section describes the *aptitude* package manager. Like *apt-get* is *aptitude* command line program and just like the *apt-get* a package can be installed using command *aptitude install package*. In additional it can be started in pseudo-graphic mode, to provide visual interface (figure 6.1). Another additional capability compared to *apt-get* is a search



```
 Actions  Undo  Package  Resolver  Search  Options  Views  Help
C-T: Menu  ?: Help  q: Quit  u: Update  g: Preview/Download/Install/Remove Pkgs
aptitude 0.7.4     #Broken: 2   Will free 715 kB of disk space DL Size: 16,6 kB
--- Security Updates (48)
--- Upgradable Packages (83)
--- New Packages (287)
--- Installed Packages (2134)
--- Not Installed Packages (81791)
--- Obsolete and Locally Created Packages (1)
--- Virtual Packages (11088)
--- Tasks (53360)

Security updates for these packages are available from security.ubuntu.com.

This group contains 48 packages.

[1(1)/...] Suggest 1 removal, 1 keep
e: Examine  !: Apply  .: Next  ,: Previous
```

Figure 6.1: Command line visual interface for *aptitude* package manager.

for packages by a part of the name (or by other attributes) using the *aptitude search* command.

6.2 Implementing new package manager module

Process of the implementing of *aptitude* will be described here. At first *aptitude* will be inherited from class *PackageManager*.

Listing 6.1: Aptitude inherited from PackageManager

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

}
```

Now common code is added, like constructor method and model's name.

Listing 6.2: Aptitude with common parametrs

```
public final class PM_aptitude extends PackageManager {

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

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

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

Since package manager need to read files, the CSAR handler is stored cy constructor. In addition the language is stored too, to be propagated later to Package Handler.
Now focus on *proceed* function. Line-by-line analyzer is needed, which can modify the data and in this case file will be rewritten.

Listing 6.3: Aptitude proceed

```
      @Override
public void proceed(String filename, String source)
      throws FileNotFoundException, IOException, JAXBException {
      if (cr == 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);
}
```

$isChanged$ indicates when file must be rewritten with new content from $newFile$ variable. Now an aptitude parser will be implemented, that reads one line from $line$ variable and stored it ore altered version to $newFile$. Package installation calls will be detected, commented out and package name will be propagated to Package Handler.

Listing 6.4: Aptitude parse

```
String[] words = line.replaceAll("[;&]", "").split("\\s+");
// skip space at the beginning of string
int i = 0;
if (words[i].equals(""))
      i = 1;
// look for apt-get
if (words.length >= 1 + i && words[i].equals("aptitude")) {
      // apt-get 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("packet: " + words[packet]);
                  cr.getPacket(language, words[packet], source);
            }
      }
      newFile += "#//References resolver//" + line + '\n';
} else
      newFile += line + '\n';
```

For parsing purposes the line is divided into words. Packet Handler is called by $getPackage$ function.

6.3 Integrating Aptitude into Bash module

Now the aptitude module will be added to Bash language. The only one thing to do is a adding the *aptitude* to Bash's set of package manager. This is done during Bash's construction with a sting: "*packetManagers.add(new PM_aptitude(this, cr));*".
Now the new package manager module is ready to work.

7 Check

In this chapter, the developed framework will be checked. The output CSAR will be added to and displayed by Winery. Generated Artifacts will be checked in command line.

7.1 Check by Winery

Winery was installed to test the correctness of output CSAR. This is an environment for development TOSCA systems and is useful for checking results.
The INSERT NAME from INSERT ADDRESS will be used as a source CSAR. His representation in winery is displayed on figure 7.1. This CSAR has a purely simple structure.

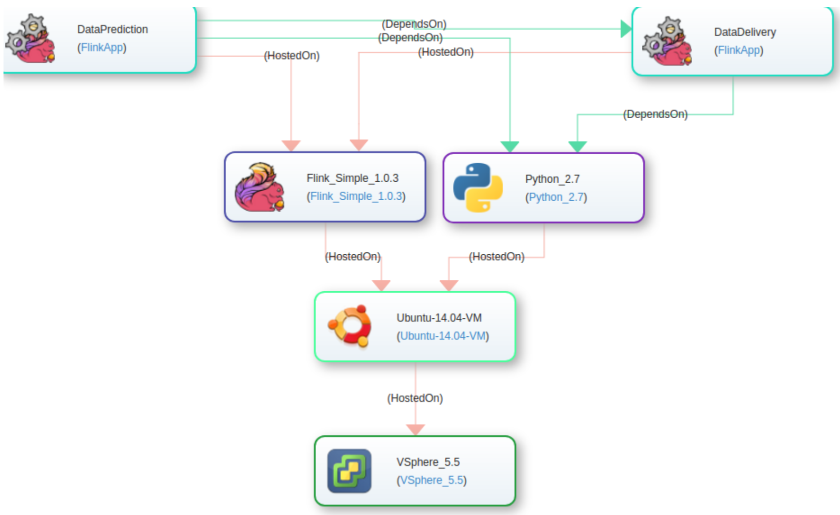


Figure 7.1: Source CSAR represented by $Winery$.

Flink Simple Application has a two submodules: Data Prediction and DataDelivery, both a hosted on Flink Simple Platform and require Python2.7. That all runs over Ubuntu 14. Deep analysis shows that Flink Simple Platform installs java and node Python2.7 installs program python2.7. Those external references are resolved during processing by the framework and exchanged by new nodes in output CSAR. This output CSAR will be added to Winery. Due to significant increase in size, this can be a fairly lengthy procedure. There where 10 nodes in source CSAR, then after processing bz the framework, there are already more then 100 of nodes. During the addition, a CSAR's syntax is tested. In case of errors, messages will be displayed. Then Service Template will be displayed. Again, due to high number of nodes, preprocessing can take a long time. But at the time, the correctness of the links will be checked. If something was defined not properly, the nodes or links between them will not be displayed. Representation of the output CSAR in the winery is shown on figure 7.2 (Only a part of CSAR is visible). It seems

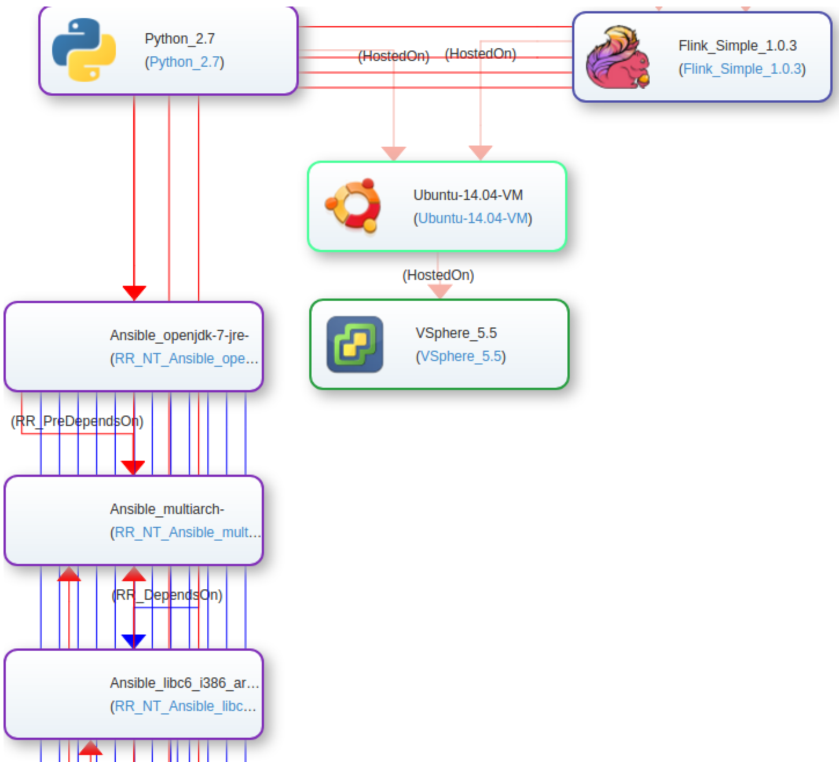


Figure 7.2: Output CSAR represented by $Winery$.

pretty beloved. To verify the TOSCA's structure some nodes was moved manually (figure 7.3). By checking several nodes with *apt-cache depends* command, the correctness of dependencies can be verified. By opening the content of the new nodes, it can be verified, that the are scripts and packages for installation.

7.2 Check installations

Also is is necessary to check whether it is possible to install new packages using the generated installation scripts. First bash scripts will be tested, then ansible playbooks.

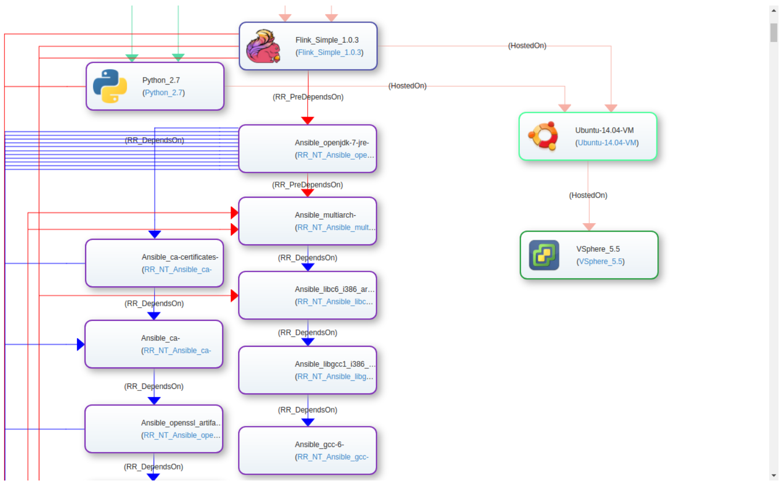


Figure 7.3: Output CSAR represented by $Winery$, nodes manually moved.

Listing 7.1 Check bash installation script

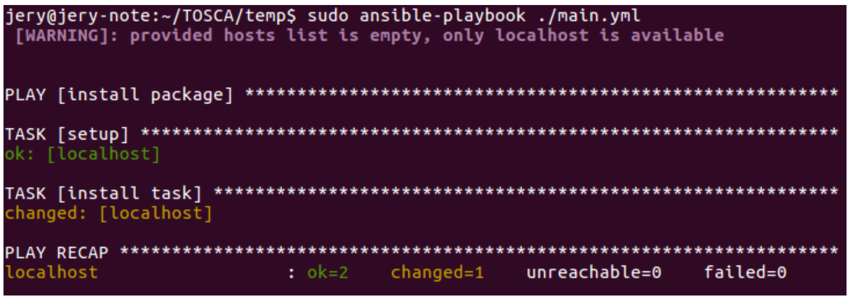
```
user@user:~$ sudo ./RR_python2_7-minimal.sh
(Reading database ... 286091 files and directories currently installed.)
Preparing to unpack python2_7-minimal.deb ...
Unpacking python2.7-minimal (2.7.12-1ubuntu0~16.04.1) over (2.7.12-1ubuntu0~16.04.1) ...
Setting up python2.7-minimal (2.7.12-1ubuntu0~16.04.1)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

7.2.1 Check bash scripts

Since bash is used in Linux's command line it will be pretty easy to check bash installation scripts by starting them (of course that must be done having necessary privileges). Example of python2.7 installation is presented in Listing 7.1.
Since the process ended without warnings and errors, it was completed successfully. This way every bash installation script can be checked.

7.2.2 Check ansible playbooks

To check ansible playbook manually we need to extract zip file containing playbook. During regular execution this work must be done by runtime environment. Calling ansible runtime to proceed the playbook is a simple procedure too. Example is provided on figure 7.4.
Ok signals that the installation was successfully completed.



```
jery@jery-note:~/TOSCA/temp$ sudo ansible-playbook ./main.yml
 [WARNING]: provided hosts list is empty, only localhost is available


PLAY [install package] *********************************************************

TASK [setup] *******************************************************************
ok: [localhost]

TASK [install task] ************************************************************
changed: [localhost]

PLAY RECAP *********************************************************************
localhost                  : ok=2    changed=1    unreachable=0    failed=0
```

Figure 7.4: Ansible playbook execution process

8 Summary

Listings

Listing 8.1 Generate the Artifact Type definition for scripts

```
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() {}
}
}


}
```

Listing 8.2 Generate definition for Script Artifact Type

```
// output filename
public static final String filename = "RR_ScriptArtifact.tosca";

/**
* Create ScriptType xml description
*
* @param cr
* @throws JAXBException
* @throws IOException
*/
public static void init(Control_references cr) throws JAXBException,
IOException {
      File dir = new File(cr.getFolder() + Control_references.Definitions);
      dir.mkdirs();
      File temp = new File(cr.getFolder() + Control_references.Definitions + filename);
      if (temp.exists())
      temp.delete();
      temp.createNewFile();
      OutputStream output = new FileOutputStream(cr.getFolder()
      + Control_references.Definitions + filename);

      JAXBContext jc = JAXBContext.newInstance(Definitions.class);

      Definitions shema = new Definitions();

      Marshaller marshaller = jc.createMarshaller();
      marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
      marshaller.marshal(shema, output);
      cr.metaFile.addFileToMeta(Control_references.Definitions + filename,
          "application/vnd.oasis.tosca.definitions");
}
```

Listing 8.3 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;
}
```

Listing 8.4 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 cr
* @param source
* @throws FileNotFoundException
* @throws IOException
* @throws JAXBException
*/
public abstract void proceed(String filename, String source) throws
    FileNotFoundException, IOException,
JAXBException;
}
```

Listing 8.5 Create TOSCA node for bash language

```
      public String createTOSCA_Node(String packet, String source) throws IOException,
          JAXBException{
if(created_packages.contains(packet+"+"+source))
return packet;
created_packages.add(packet+"+"+source);
packet = getNodeName(packet, source);
RR_NodeType.createNodeType(cr, packet);
RR_ScriptArtifactTemplate.createScriptArtifact(cr, packet);
RR_PackageArtifactTemplate.createPackageArtifact(cr, packet);
RR_TypeImplementation.createNT_Impl(cr, packet);
return packet;
}
```

Listing 8.6 File parsing for Bash + apt-get

```
public void proceed(String filename, String source)
throws IOException, JAXBException {
String prefix = "";
for (int i = 0; i < Utils.getPathLength(filename) - 1; i++)
prefix = prefix + "../";
if (cr == 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) {
// split string to words
String[] words = line.replaceAll("[;&]", "").split("\\s+");
// skip space at the beginning of string
int i = 0;
if (words[i].equals(""))
i = 1;
// look for apt-get
if (words.length >= 1 + i && words[i].equals("apt-get")) {
// apt-get found
if (words.length >= 3 + i && words[1 + i].equals("install")) {
// replace "apt-get install" by "dpkg -i"
System.out.println("apt-get found:" + line);
isChanged = true;
for (int packet = 2 + i; packet < words.length; packet++) {
System.out.println("packet: " + words[packet]);
//                                cr.AddDependenciesScript(source, words[packet]);
cr.getPacket(language, words[packet], source);
}
}
newFile += "#//References resolver//" + line + '\n';
} else
newFile += line + '\n';
}
br.close();
if (isChanged) {
// references found, need to replace file
// delete old
File file = new File(filename);
file.delete();

// create new file
FileWriter wScript = new FileWriter(file);
wScript.write(newFile, 0, newFile.length());
wScript.close();
}
}
```

Listing 8.7 Ansible proceeding

```
      public void proceed(Control_references cr) throws FileNotFoundException,
IOException, JAXBException {
      if (cr == 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);
      }
}

/**
* proceed given file
*
* @param filename
* @param cr
* @param source
*         of file, example - archive
* @throws FileNotFoundException
* @throws IOException
* @throws JAXBException
*/
public void proceed(String filename, String source)
throws FileNotFoundException, IOException, JAXBException {
      for (PacketManager pm : packetManagers)
      pm.proceed(filename, source);
}

/**
* Handle ZIP package
*
* @param zipfile
* @throws FileNotFoundException
* @throws IOException
* @throws JAXBException
*/
private void proceedZIP(String zipfile) throws FileNotFoundException,
IOException, JAXBException {
      boolean isChanged = false;
      // String filename = new File(f).getName();
      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));

}
```

Bibliography

[13] *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. 17).

[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. 27).

[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. 17).

[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. 17).

[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. 23).

[OAS] OASIS. *Organization for the Advancement of Structured Information Standards*. URL: https://www.oasis-open.org/ (cit. on p. 24).

[OAS13] OASIS. “Topology and Orchestration Specification for Cloud Applications Version 1.0.” In: *OASIS Committee Specification 01. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.* Mar. 18, 2013 (cit. on p. 25).

[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 p. 21).

[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. 17).

[Stu13] I. U. Stuttgart. *OpenTOSCA - Open Source TOSCA Ecosystem*. 2013. URL: http://www.iaas.uni-stuttgart.de/OpenTOSCA/ (cit. on pp. 17, 26).

[tec] techopedia. *Definition - What does Cloud App mean?* URL: https://www.techopedia.com/definition/26517/cloud-app (cit. on p. 21).

[wika] wikipedia. *Ansible (software)*. URL: https://en.wikipedia.org/wiki/Ansible_(software) (cit. on p. 30).

[wikb] wikipedia. *Bash (Unix shell)*. URL: https://en.wikipedia.org/wiki/Bash_(Unix_shell) (cit. on p. 30).

All links were last followed on March 17, 2008.

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