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 even 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 Systems, in other words, a layer between the external operations of cloud systems and Cloud Service provider's API. With the help of TOSCA, it's possible to describe several methods of interaction with different APIs, what allows to automate the rapid deployment/roll-up on the sites of different providers. The University of Stuttgart implemented this Specification in the Runtime Environment named OpenTOSCA. Description of a Cloud System is stored in the **C**loud **S**ervice **AR**chive (CSAR), which contains all components necessary for a cloud system life-cycle.

Often, cloud systems are described so that during they deployment, additional packages and programs need to be downloaded via the Internet. Even with a single platform, this can significantly slow down the deployment of cloud system, and if it's necessary to deploy a large number of servers, each of them will download a large amount of data during the deployment, this can significantly increase both time and money consumption.

During this work an framework will be implemented, which will eliminate external dependencies in a CSAR, resupply they with all necessary for deployment packages and also change the internal structure to display the achieved self-containment. For example, all commonly used "apt-get" commands, which download and install packages, must be removed. Appropriate package must be downloaded and integrated into CSAR structure. Furthermore, all dependencies needed for that package must be resolved too.

This Document concern about Concept and Architecture for mentioned framework. Besides some aspects of implementation will be described.

Contents

List of Figures

List of Tables

List of Listings

List of Algorithms

List of Abbreviations

CH **C**SAR **H**andler. 38

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 use Cloud applications. And that are not only big company but also small firms. [Bun14; Bun17] One of the most important reason for development of cloud applications is the saving of resources. It is much easier and often cheaper to rent a part of someone else's big mainframe, then to maintain your 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 coagulation of cloud applications.

One common problem of cloud application *affection*. The transfer of a cloud application configured to work with the API of one provider, to work with another provider and another API is a difficult, but important task. The ability to quickly transfer cloud application to more suitable provider, is a key to developing competition and reducing the cost of maintenance.

Topology and **O**rchestration **S**pecification for **C**loud **A**pplication (TOSCA) [13] provides a opportunity to solve this problem. TOSCA defines the language, which allows to describe cloud application and they management portable and inter-operable. The use of TOSCA allows to simplify and automate the management of cloud applications by different providers. By 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 of the application

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 needed to deploy the cloud system, 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 megabytes of pre-known data. If there is one deployed server, this may mean a few seconds of delay, but when deploying a large number of servers,

the costs can increase significantly.

Another problems with external dependencies are security and stability. To ensure the security of information, some enterprises restrict Internet access for internal networks. In other networks, 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.

As part of this work is necessary to implement a framework for resolving external dependencies in CSARs. This framework should analyze the CSAR, identify dependencies to external packages and resolve them by downloading this packages (as well as all other dependent packages) and adding them to the CSAR's structure. The simplest example is to find all the commands like "apt-get install package", delete this command, download the package and all packages necessary to this package, and add them to CSAR. This framework must be easily expanded, since it is impossible to predict and describe all possible types of external dependencies. The Result of the framework is a CSAR, which contains additional to original structure, like all the packages necessary for the deployment of the cloud application, with the minimum possible level of access to 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), CSAR archive (section 2.2.1), 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 concept as well as architecture of the framework are illustrated and explained.

Chapter 5 – Implementation: In 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

2.1 Cloud Computing and Cloud application

2.1.1 Definitions

Unfortunately, the is no generally accepted definition of cloud computing that describes all possible situations. But in scientific community the definition put forward by **N**ational **I**nstitute of **S**tandards and **T**echnology (NIST) is often 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)
Cloud application is an application that is executed according to a cloud computing model. [tec]

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 you to effectively use resources, distributing the load to a system from several physical servers and shifting the maintenance to the providers. If your service uses a single one physical server and this server will be disabled, then entire service will be completely unavailable too. But if you use cloud computing with a hundred of physical servers, then disabling one of them will not carry such serious consequences. In addition, the user does not need to maintain a team of administrators for the event of various problems.

Usually the user does not have direct access to the infrastructure (servers and operating systems), he uses only provided interface. An interface provides a set of methods to communicate with cloud infrastructure. Each provider defines his own set of methods, depending on his area of specialization. On the one hand this specialization makes it easier to work with the provider, on the other hand it becomes more difficult to switch to work with 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 new ways to enable portable automated deployment and management of composite applications. TOSCA describes the structure of composite applications as topologies containing their components and their relationships. Plans capture management tasks by orchestrating management operations exposed by the components. [BBKL14] TOSCA application is a cloud application described according to TOSCA standard.

Usage

TOSCA can be used not only for describing all stages of a cloud application life-cycle, but alse serve as a layer between the cloud application and provider's interfaces, allowing to implement a single application suitable for working with different providers.

Structure

TOSCA specification provides a language to describe service components and their relationships using a service topology, and it provides for describing the management procedures that create or modify services using orchestration processes. The combination of topology and orchestration in a Service Template describes 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.) when the applications are ported over alternative cloud environments. [OAS13]
Descriptions of the main components of TOSCA used in this work will be provided.

- Service Template. This component contains an Information about structure (TopologyTemplate) and Plans of cloud application.

- Plans 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 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 nodes 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 nested Source Element and Target Element elements).

- Relationship Type defines the semantics and any properties of the relationship.

- Artifact represents the content needed to realize a deployment such as an executable (e.g. a script, an executable program, an image), a configuration file or data file, or something that might be needed so that another executable can run (e.g. a library). Artifacts can be of different types, for example EJBs or python scripts. The content of an artifact depends on its type. Typically, descriptive metadata will also be provided along with the artifact. This metadata might be needed to properly process the artifact, for example by describing the appropriate execution environment. TOSCA distinguishes two kinds of artifacts: implementation artifacts and deployment artifacts.

- Implementation Artifact represents the executable of an operation of a node type. An example will be provided in section 2.4

- Deployment Artifact represents the executable for materializing instances of a node.

- Artifact Type is describes an type of a artifact.

- 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.

Types and Templates defining a TOSCA application are stored in definition documents.

2.2.1 CSAR

A **C**loud **S**ervice **AR**chive (CSAR) is used to store the TOSCA application. This 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 TOSCA runtime environment.

Structure

The root folder must contain a "Definitions" and a "TOSCA-Metadata" folders. The "Definitions" folder contains definition documents, 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

Terms

Input **C**loud **S**ervice **AR**chive (CSAR) is a CSAR, which can contain external references and will be processed by the framework.
Output CSAR is a SCAR, which was processed by the framework and doesn't contain external references.

2.3 OpenTOSCA

OpenTOSCA provides an open source ecosystem for TOSCA applications. The Open-TOSCA ecosystem consists out of three parts: [Stu13]

- OpenTOSCA **Container**, a TOSCA runtime environment

- **Winery** , a graphical modelling TOSCA tool.

- **Vinothek**, a self-service portal for the applications available in the container

The runtime environment ant Winery will be provided in more details.

Runtime environment

The runtime enables fully automated plan-based deployment and management of applications in the CSAR format. 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 runtime and therefore visual interface is available in 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 on figure 2.2.
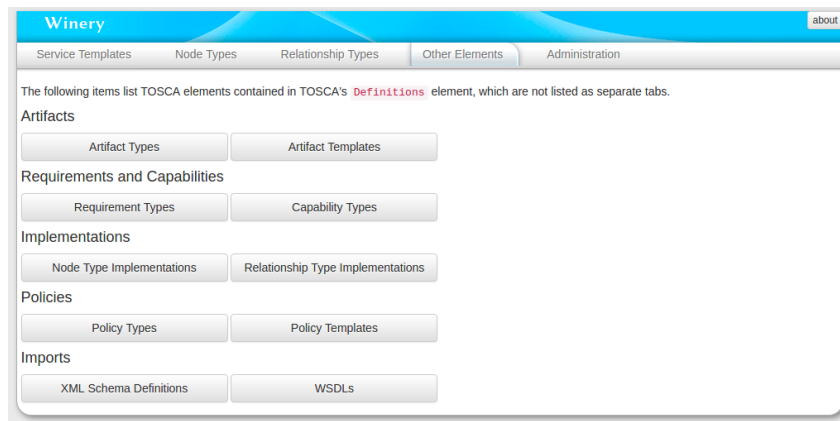
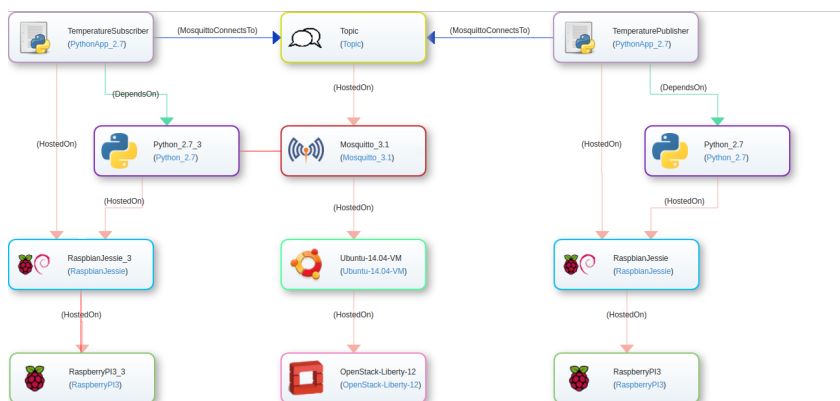**Figure 2.1:** Visual interface for *Winery*.

Figure 2.2: TOSCA topology presented by *Winery*.

2.4 Package management

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

Package managers

Package Manager is a set of software tools that automate the process of installing, updating, configuring and removing of computer programs. 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 manage 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 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 results 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 the user can also 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 of obtaining a dependency list 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 $required$ leads to the presence of cycles in dependency tree, which differs them from normal tree graph structures.

2.5 Package management automation

To automate the package management, various environment are used. Those environments 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 program is to Resolve References, further the RR will be used as an abbreviation. A framework should be developed to eliminate external dependencies in TOSCA topology represented in CSAR file. The framework should be easy extendable to provide the ability to eliminate a large number of dependency types. At first a minimal configuration will be developed, that finds all the bash scripts which use the apt-get package manager. All package installation commands should be removed (apt-get install package). Then both the package itself and all the packages from his dependency tree should be downloaded. It is also necessary to update the topology of the TOSCA, by adding nodes and dependencies. Common definitions should be added, like Relationship Types and Artifact Types. In the TOSCA language new nodes can be defined by Node Types, Node Type Implementations and Artifacts, and then instantiated by Node Templates. Relations will be instantiated by Relationship Templates. These Templates must be added to right Service template, where nodes containing external references are instantiated. To find the Service Templates and Node Types corresponding to a certain artifact, it will be useful to use preprocessing of entire TOSCA topology. References chain can be build:

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

After implementing a bash language, it should be easy to add additional script languages and package managers, like $aptitude$ for bash or new languages like $shef$ or $ansible$. To proof the correctness of the corresponding TOSCA topology a winery described in section 2.3 can be used.

Stages of CSAR processing

Here example steps are provided, representing how the framework should work.

- Begin
  To start the work our program need to became input CSAR name, output CSAR name, and architecture of target hardware. Then input CSAR need to be extracted.

- Preprocessing
  During preprocessing stage, RR need to analyze internal references and build dependency three. Furthermore, common Tosca definitions for artifacts and relations between packages need to be added.

- Processing with Languages
  Each file from CSAR need to be processed for each described Language.

- Processing with Packet Managers
  If file belongs to the Language, he will be processed by Packet Manager Handler to find and resolve external references. Package name from this reference will be provided forward.

- Package handling
  Using a package name the right package must download, TOSCA definitions created and recursively repeats for all dependent packages, creating Dependency Tree.

- Topology handling
  Using information about internal references and dependencies TOSCA Topology will be updated by creating Node and Reference Templates.

- End
  Meta-file should be updated and all data packed back to the CSAR.

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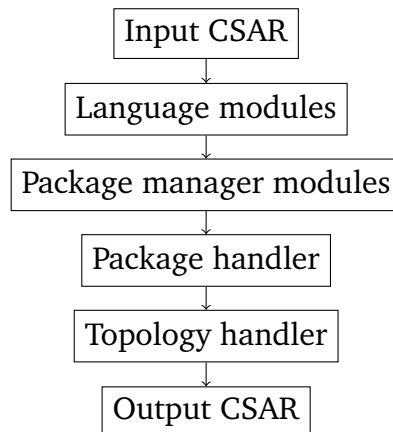
4 Concept and Architecture

4.1 Concept

In this section main concepts of this work will be described. In general, the structure of framework can be described by the diagram 4.1.
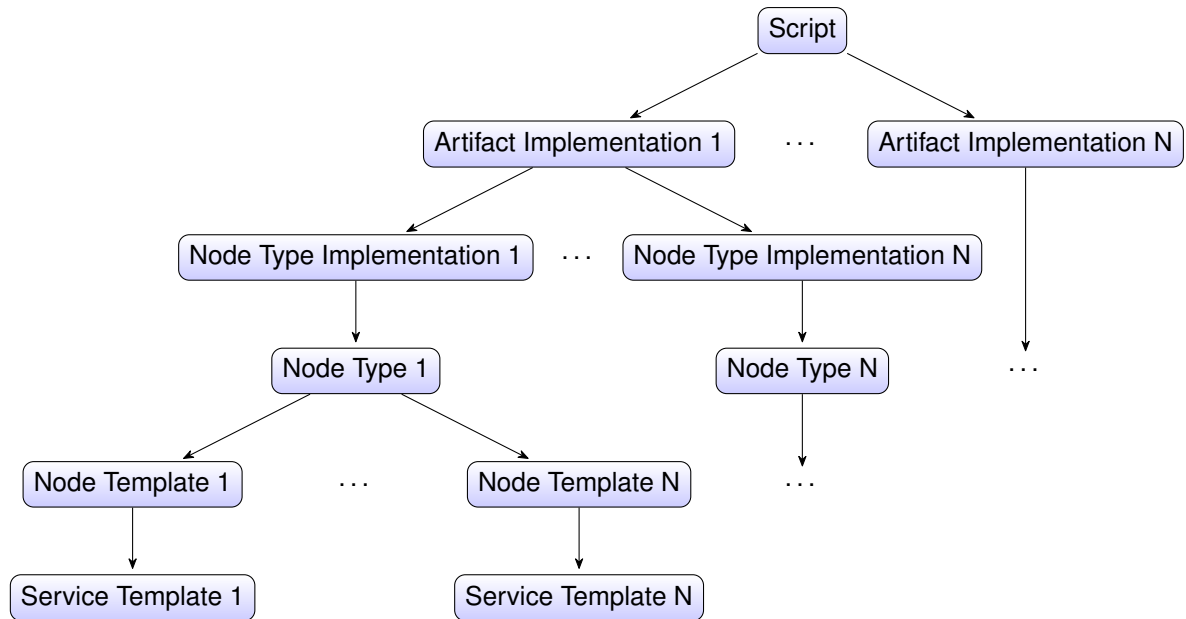
4.1.1 Analysis existing TOSCA-Topology

To properly display new packages in TOSCA topology, it is necessary to add to each node containing an external reference a reference to newly created node resolving the external reference. Besides Service Templates must be found where those nodes are instantiated. As was already pointed, according to TOSCA standard a simple sequence can be build:

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

Now consider this bindings in more detail.

Input CSAR
↓
Language modules
↓
Package manager modules
↓
Package handler
↓
Topology handler
↓
Output CSAR

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

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

- $script \rightarrow Artifact\ Implementation$
  A script can be referenced by several Artifact Implementations (Despite the fact that this is a bad practice).

- $Artifact\ Implementation \rightarrow Node\ Type\ Implementation$
  The same way an artifact can be used by several Node Type Implementations.

- $Node\ Type\ Implementation \rightarrow Node\ Type$
  Node Type Implementation can describe implementation of only one Node Type

- $Node\ Type \rightarrow Node\ Template$
  Each Node Type can have a lot of Node Templates

- $Node\ Template \rightarrow Service\ Template$ But each Node Template will be defined only in one Service Template.

Thus structure can be described not by a sequence, but by a tree with the script as a root, and Service Templates as leaves (Example in figure 4.2).

Of course it is possible to move in opposite direction, starting from a Node and moving toward scripts, but this method brings additional complexity. Node Type can have several Node Type Implementations, but which one will be used can be determined only during deployment. The method presented above can uniquely determine Node Templates and Service Template for a given script. Of course it is not guaranteed that found Node

---

Algorithmus 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
```

Type Implementation will be used during deployment, but we can't do anything with this. The following steps can be executed during a preprocessing, to identify necessary dependencies.

- Find all Artifact Implementations to build dependency from script name to Artifact Implementation.

- Find all Node Type Implementation. Since those definitions contains both the Node Type and used Artifact Implementations, the dependency from script to Node Types can be build.

- Find all Service Templates, and in them contained Node Templates. So the necessary dependencies can be achieved (script to Node Template and script to Service Template).

Find external dependencies

Analysis for external references. Unfortunately it is impossible to identify all possible external dependencies, even within one language and one package manager (example in algorithm 4.1). Since this work is aimed at creating a common tool that is easily expanded and supplemented, initially only basic usage of package managers will be considered, but complimented when necessary. Ease of adding modules to framework will proof 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 by writing one line. When processing of this combination will reach an acceptable level, new languages and package managers will be added.

4.1.2 Representing downloaded packages in TOSCA-Topology

A package node denote to 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 artifacts or relations. This can be done once at preprocessing stage.

- The package node definition will be represented by a Node Type. There will be described that this node must be installed.

- Downloaded data and they installation script will be referenced by Artifact Template.

- Node Type Implementation will combine all artifacts.

- Node Template will instantiate a package in the corresponding Service Templates. To determine corresponding Service Template a preprocessing described in Analysis existing TOSCA-Topology will be used.

- Reference Template will provide topology information, allowing the observer (user or runtime environment) to determine, for which nodes the package must be installed. References will be created both from origin Node Template to the package's Node Template and between Node Templates of created packages.

4.1.3 Determining architecture of a final Platform

The most acute way is the question about the architecture of the device where deployment scripts will be executed. Unfortunately, it is impossible to analyze the structure of any CSAR and give an unambiguous answer to the question, on which architecture which command will be executed. There are many pitfalls here. A single Service Template can use several physical devices with different architectures. The same Implementation Artifacts can refer to different Node Types, instantiated on different platforms. So one simple Implementation Artifact with bash script containing "*apt-get install python*" command when deployed on different devices (for example with arm, amd64 and i386 architectures) will result in the loading and installation of 3 different packages. For an end user, the ability to use such a simple command is a huge relief, but for the framework it can greatly complicate analysis. The following methods of architecture selection were designed.

- Deployment environment analysis
  The script analyzing the system where it is running (for example using $uname\ -a$ command) and depending on the result, installing the package corresponding to the architecture.

- Unified architecture
  An Unified architecture, predefined by the user at framework startup.

- Artifact specific architectures
  An architecture can be defined for each artifact.

Analysis of methods

Unfortunately, the Deployment environment analysis, which at first glance seems to be the most reliable solution, brings many additional problems. Packages for different platforms can differ not only by architecture, but also in the versions and the list of dependencies. As a consequence, a complete chaos will be produced when trying to display these different packages with different versions in the TOSCA-topology. The only robust solution would be to create for each installed package a set of archives (one archive for each architecture), containing the entire dependency tree for the given package. But this approach contradicts one of the ideas of this work: each package should be mapped in topology.

The Artifact specific architectures carries additional complexity to the user of the framework. The user will be obligate 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 most simple and easy to implement. If it will be necessary, this method can be easy expanded to Artifact specific architectures method (be replacing the user input to chose an unified architecture at start, by choosing an architecture for each artifact.) and to Deployment environment analysis (By downloading packages for all available architectures and adding the architecture determination to installation script.).

4.1.4 Extensibility

Framework should handle different script languages, each of them should support a various package managers. This principle can be illustrated by a figure 4.3.
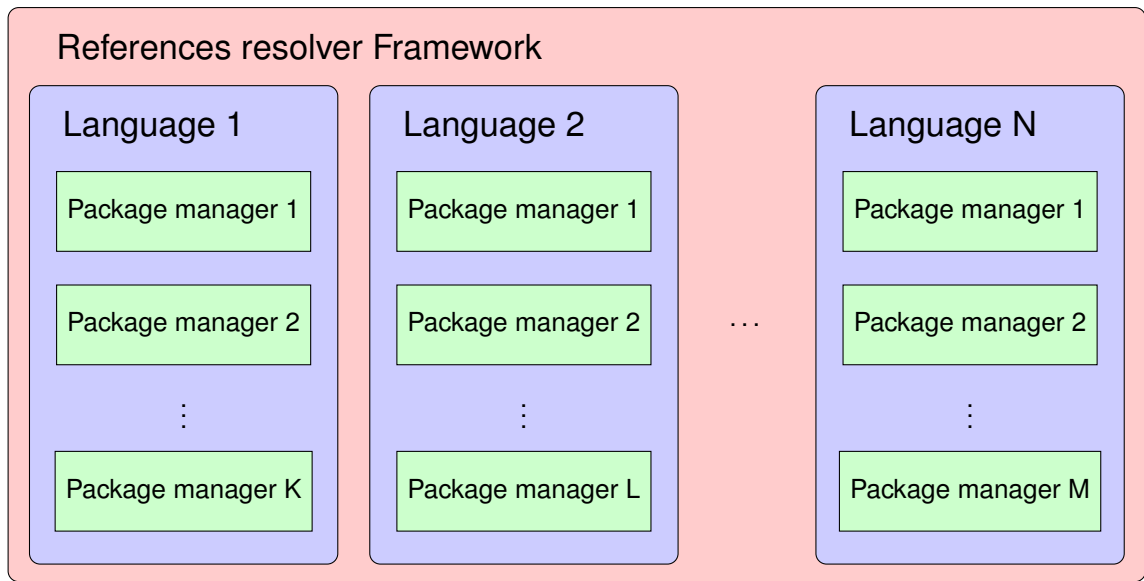
Figure 4.3: Example scheme representing several languages and package managers

4.1.5 Result's checking

Checking the results of the work 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 the output CSAR. To test for overall correctness, it is possible to use winery tool from OpenTOSCA. This tool for creating and editing archives is also great for visualizing results. Checking the deployment of the CSAR is a much more complicated task, for which OpenTOSCA runtime environment can be used. But it's also necessary to create proper environment for deploying the cloud application.

4.2 Architecture

This section will present the architecture of the framework and a description of its modules. The main modules of the Framework are: Language modules, Package manager modules, Package handler module and Topology handler module.

4.2.1 CSAR handler

CSAR **H**andler (CH) is a module to provide access to CSAR and to maintain CSAR consistency. It describes process of adding new files (to handle the matadata), decompression, architecture processing, etcetera.
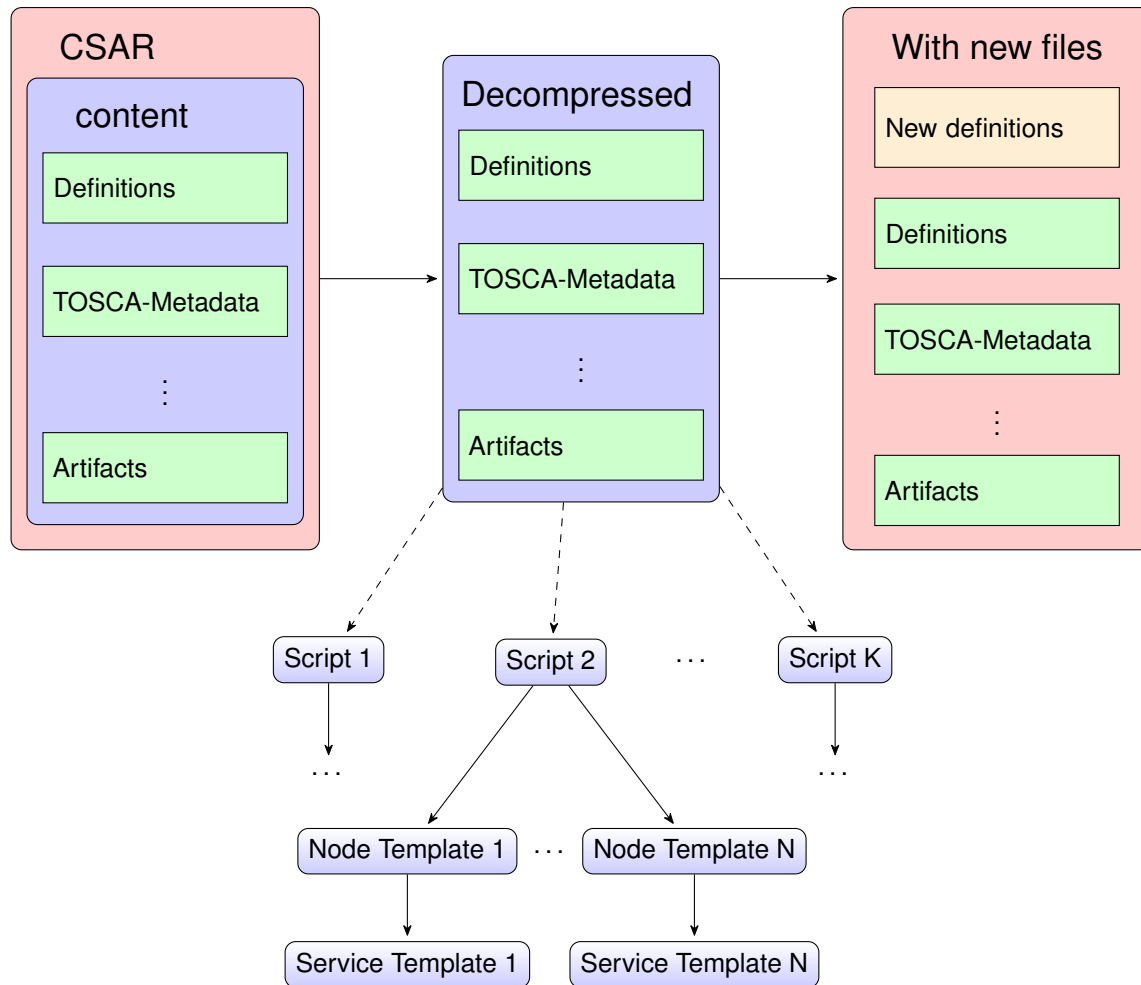
**Figure 4.4:** Preprocessing: decompression, adding files and generating dependencies

4.2.2 References resolver

This is a main module, which execution can be divided into three stages: preprocessing, processing, finish.

At the begin the preprocessing will be executed for decompression, adding files and generating dependencies (generation is described in section 4.1.1). Figure 4.4 illustrates the stages of preprocessing.

Then during the processing, any 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..
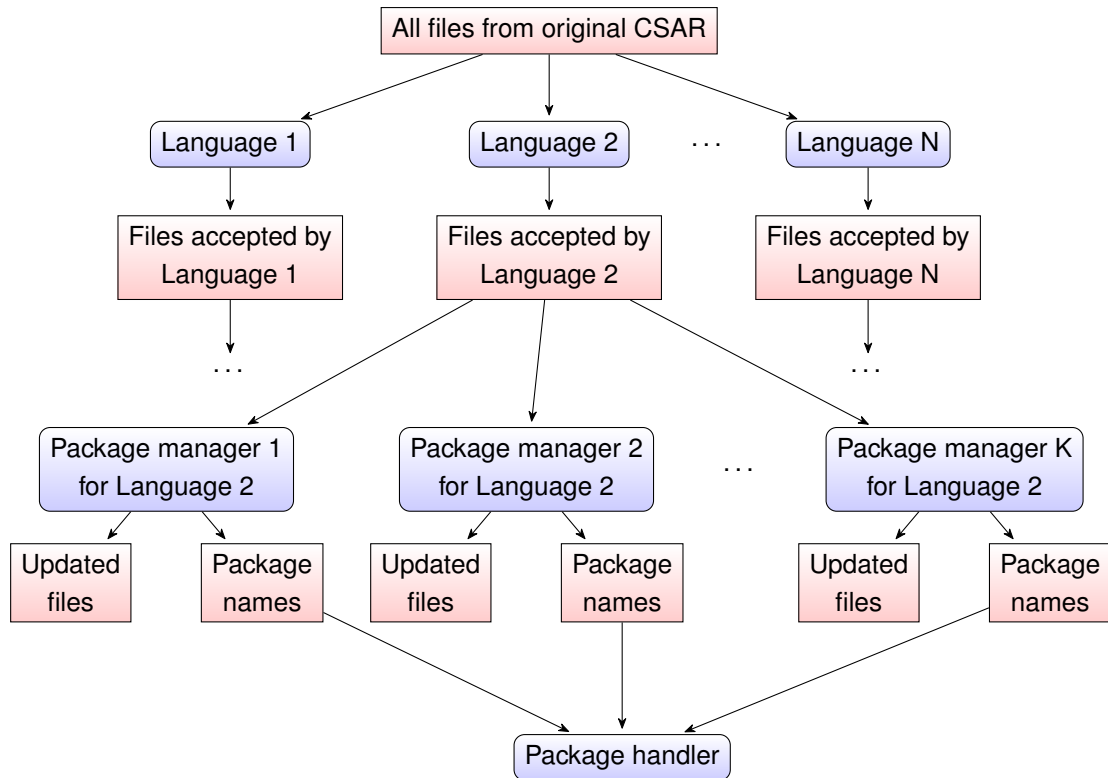
Figure 4.5: Data flow scheme between language modules, package manager modules and package handler.

4.2.3 Language modules

As already mentioned above, language modules analyze each file and check its belonging to the specific language. Files not belonging to the language are filtered out. The remaining files are transferred to this language'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

Package managers modules should find external references, resolve them and transmit package name to package handler module, described in next section. Figure 4.5 illustrates data flow between languages, package managers and package handler.

4.2.5 Package Handler

Package Handler become package name and should download installation package, transfer package name to Topology handler and recursively repeat the actions for all depended packages.

4.2.6 Topology Handler

The Topology Handler adds package to the topology. This includes adding new files and updating existing files. Necessary steps were described in section 4.1.2.

5 Implementation

This chapter provides implementation of the framework and his modules and stages, which was described in chapter 4. For the implementation was chosen a Java language, because of his simplicity and strength.

5.1 Global modules

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

CSAR handler

CH handle common information about CSAR content. It handles:

- Temp extraction folder.
- List with files to proceeded.
- Meta-file entry.
- Architecture of target platform.

All this data are encapsulated into CH, to access them public function of this class should be used.

Utils

This class provides useful methods, used by many other modules.

- Create file with content.
- Get path deep.
- Adapt name for OpenTOSCA.

This functions doesn't belong to specific module and can be used in many cases.

Listing 5.1 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);
/**
* Unzip it
*
* @param zipFile
*        input zip file name
* @param outputFolder
*        output folder
*/
static public List<String> unZipIt(String zipFile, String outputFolder);
```

Zip handler

This is a small module with strait functionality. It serves to pack-unpack zip archives, which are used by CSARs. For handle with archives it was decided to use common Java package $java.utils.zip$. The functions of archiving, unarchiving were implemented in one class zip with public methods $zipIt$ and $unZipIt$, which are provided in listing 5.1

5.2 References resolver

This module is executed into three stages.

Preprocessing

At the preprocessing stage a CSAR will be unarchived, common TOSCA definitions generated and internal dependencies found.

Listing 5.2 Processing stage of references resolver

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

Generating TOSCA Definitions

For generating common glstosca definitions a $javax.xml.bind$ package was chosen. Descriptive descriptions for common definitions were created.

- DependsOn and PreDependsOn definitions describe Relationship Types (glsTOSCA Structure) for according relations between packages (Dependencies).

- Package Artifact Type defines a deployment Artifact Type for package installation data.

- Script Artifact Type defines a implementation Artifact Type for package installation script.

- Ansible Playbook Type defines a implementation Artifact Type for package installation via Ansible playbook.

example can be found in listing 8.1

Finding internal references

Topology Handler mainly uses internal references from Analysis existing TOSCA-Topology. Therefore, this two modules were combined to one class $Topology\_Handler$. At the preprocessing stage, he analyses all origin definitions to build internal relations, as was described in section 4.1.1. To read origin definition a package $org.w3c.dom$ was used.

processing

During this stage all described language modules are started.

Finishing

To finish the work changed data should be packed back to CSAR. Function $zipIt$ from class zip, defined in listing 5.1 is used.

5.3 Search for external references

This section will describe the finding external dependencies in the original artifacts. For this purpose serve Language modules and Package manager modules. Since the framework is initially oriented to easy extensibility, abstract classes for Language modules and Package manager modules will be described. New language and package manager modules can be added by implementing these methods.

Language Model

To describe common functionality of different language modules, the abstract class Language was defined in listing 8.3. Common components for all language models are:

- Language name

- set of package manager modules

- Extensions of files

Common functions are:

- $getName$ returns this language's name.

- $getExtensions$ returns this language's extensions.

- $proceed$ checks all original files and transfer results to package manager modules.

- $getNodeName$ returns how should TOSCA node installing package with this language be named.

- $createTOSCA\_Node$ creates TOSCA definitions for new package. Since created TOSCA definitions depends from original script's language, these languages must provide method for creating the definitions.

Package Handler Model

As with languages, an abstract class is defined at first. This definition contains only one function $proceed$ (listing 8.4) that finds and eliminates external references, as well as passes the package names to package handler.

Bash implementation

The processing of popular Bash language was implemented. As a sign of belonging to the Bash language, the file extension (".sh" and ".bash") and the first line ("#!/bin/bash") are used. All files that satisfy this conditions are passed to package managers modules, in our case - $apt\text{-}get$. TOSCA node containing a package and bash install script is defined by Node Type, Node Type Implementation, Package Artifact, Script Artifact. This definitions are created my $createTOSCA\_Node$ method presented in listing 8.5. now consider it in more details. To avoid creating of the same nodes, we store names of created nodes in $created_packages$ list, and check it before start. Then language specific node name is generated and TOSCA definitions for this name are created.

Apt-get Bash implementation

The apt-get package manager module processing is fairly simple line-by-line file parser that searches for lines starting with the "$apt\text{-}get\ install$" string and passes this command's arguments to package handler's public function $getPackage$ described later. 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 oft packed to archives, 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 this operations. As a sign of the ansible language, the "$.yml$" extension is used, since this files doesn't contain specific header.

Creating a TOSCA node for this language is a complicated operation. The basic moments are:

- Analyze original files to determine an ansible configuration (set of options like username or proxy).

- It can be necessary to complete a configuration using user input.

- Create installation data (executable yml file and folder with package)

- Pack installation data

- Create TOSCA definitions describing installation data

Listing 5.3 *getPackage* definition

```
/**
* Download package and check its dependency
*
* @param packet
*        package name
* @param cr
*        CSAR manager
* @param depth
*        dependency level to be checked
* @param listed
*        list with already included packages
* @return list of packages
* @throws JAXBException
* @throws IOException
*/
getPacket(Language language, String packet, List<String> listed, String source, String
    sourcefile);
```

Apt implementation

Since the package installation using *ansible* language and *apt* package manager can be described in many different ways, the processing of is complicated too. It's worth mentioning that the processing uses a simple state machine and regular expression from *java.util.regex*.

5.4 Package Handler

Package handler is an interface for interacting with the package manager of the operating system. Allows to load packages and to determine the type of dependency between them.

Package downloading

This operation is performed using one recursive function *getPacket* defined below in listing 5.3. This function downloads packages for full dependency three, calls corresponding Language to create nodes and Topology Handler to update topology. Arguments will be described shortly.

- *Language* is used to coll right *createTOSCA_Node* function.

- *packet* defines package-name to download.

- *listed* holds a list with packages already presented in the dependency tree. No need to download them, but dependencies will be created.

- *source* defines parent-element in dependency tree. For root package that will be a source script file, for other packages - depending package.

- *source file* is source script file name. And is used to generate package name and by Languages to get additional information for node creating.

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

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 Service Templates. For this purpose the Finding internal references stage is executed during preprocessing stage.

Update Service Templates

To update Service Templates two functions are provided.

- *addDependencyToPacket* generates dependency between two package nodes generated by the framework.

- *addDependencyToScript* generates dependency between original node and package node generated by the framework.

To update existing TOSCA definition the *org.w3c.dom* and *org.xml.sax* packages are used.
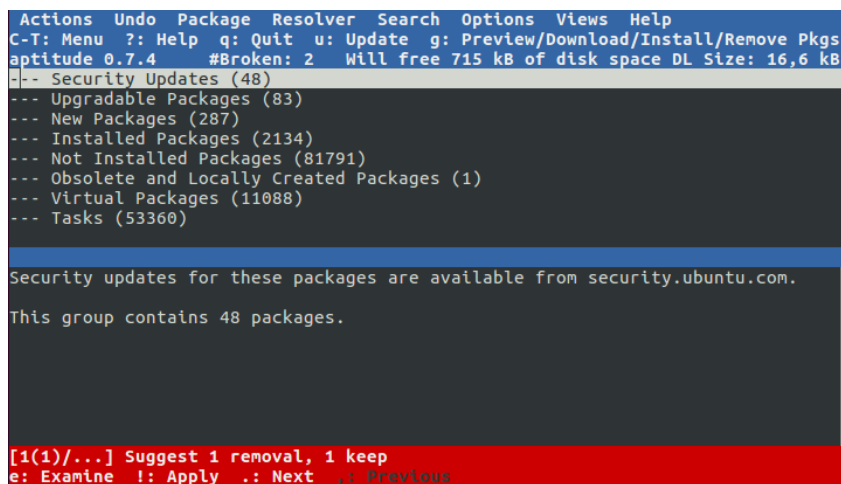
6 Add new package manager module

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

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.
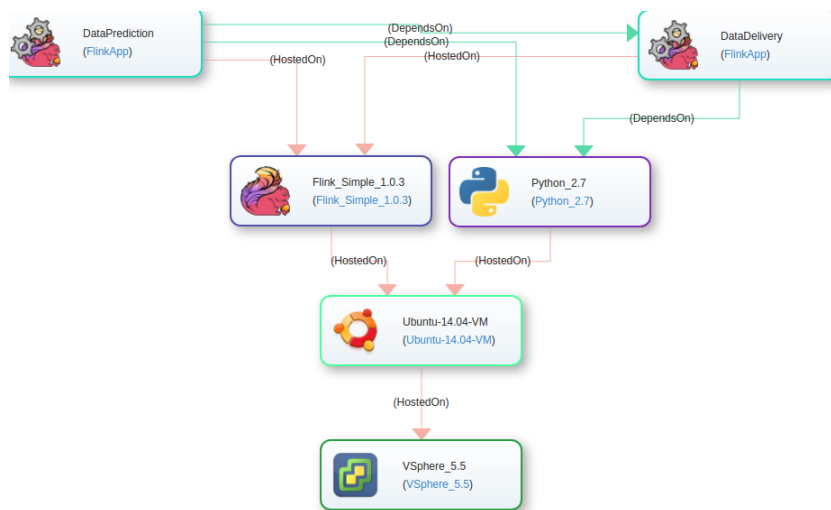
Figure 7.1: Source CSAR represented by $Winery$.

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
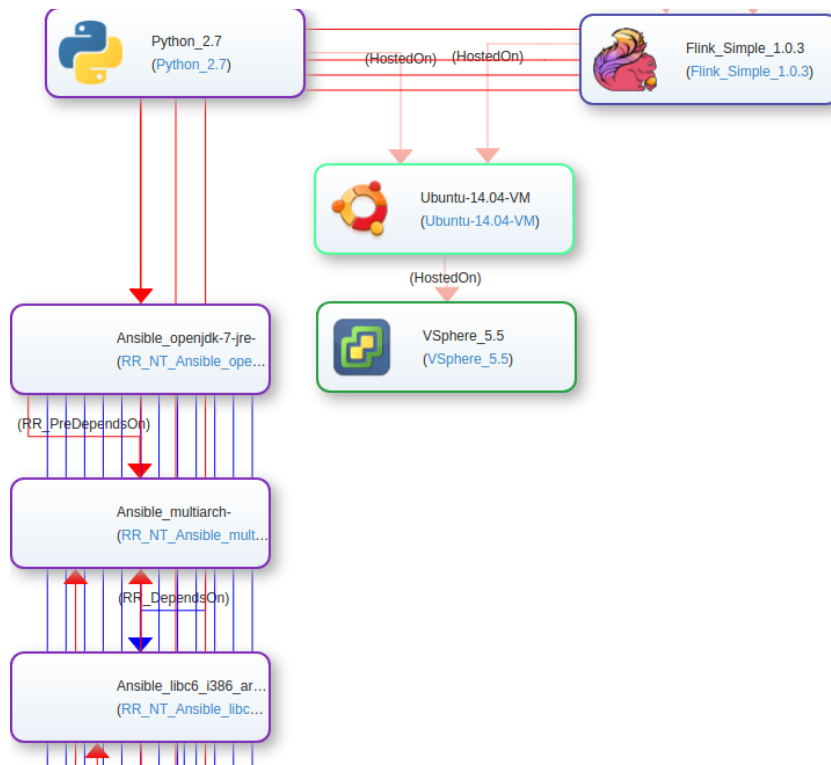

Figure 7.2: Output CSAR represented by $Winery$.

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.
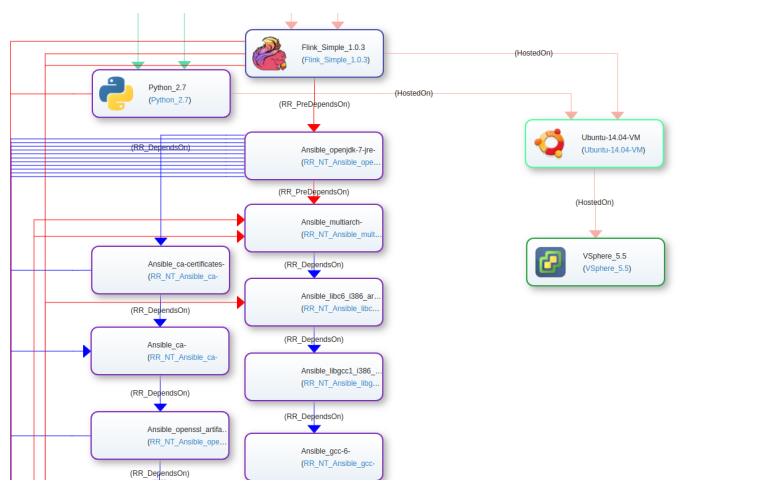
Listing 7.1 Check bash installation script

```
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.

Figure 7.4: Ansible playbook execution process

8 Summary

Listings

Listing 8.1 Generate Script Artifact Type

```
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

```java
// 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

```java
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

```java
    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

```java
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. 26).

[BBKL14]   T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. In: *Advanced Web Services*. New York: Springer, Jan. 2014. Chap. TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. ISBN: 978-1-4614-7534-7. DOI: 10.1007/978-1-4614-7535-4_22 (cit. on p. 23).

[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. 23).

[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. 24).

[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. 29).

[wikb]   wikipedia. *Bash (Unix shell)*. URL: https://en.wikipedia.org/wiki/Bash_(Unix_shell) (cit. on p. 29).

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