Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis

Dynamic Cloud Provisioning based on TOSCA

Anshuman Dash

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	M.Sc. Kálmán Képes
Commenced:	May 24, 2017
Completed:	October 24, 2017
CR-Classification:	D.2.11, D.2.9, D.2.12, D.2.13

Abstract

Cloud computing, today, is a ubiquitous paradigm. Its features such as availability of a practically infinite pool of computing resources, on demand, by using a pay-per-use model has resulted in its adoption by the industry for the realization of modern, sophisticated, and highly scalable IT applications. Such applications are often comprised of various components and services offered by different cloud service providers. This, in turn, raises two significant challenges- (i) automated provisioning and management, and (ii) interoperability and portability of the applications in a multi-cloud environment. As a result, the Topology and Orchestration Specification for Cloud Applications (TOSCA) standard was introduced by OASIS. This standard provides a metamodel to describe the topology of complex applications along with all the components, artifacts, and services in a single template that allows deploying the application in an interoperable and portable manner. In this Master thesis, we propose a concept that generates small and reusable TOSCA provisioning plans which can be orchestrated to deploy the overall application as opposed to using a monolithic provisioning plan. This goal is achieved in three steps - (i) splitting the application topology into a set of smaller sub-topologies, (ii) generating smaller plans called partial plans for each sub-topology, (iii) and finally orchestrating the partial plans to provision an instance of the application. Additionally, this concept enables the reuse of these plans for tasks such as scaling out individual components of the application. Finally, the feasibility of the proposed concept is demonstrated by a prototypical implementation developed using the OpenTOSCA framework.

Contents

1 Introduction **13**
- 1.1 Problem Statement . . . 15
- 1.2 Motivating Scenario . . . 16
- 1.3 Scope of Work . . . 17
- 1.4 Thesis Outline . . . 18

2 Fundamentals **19**
- 2.1 Important Terms . . . 19
- 2.2 Cloud Computing . . . 20
- 2.3 OASIS TOSCA . . . 23
- 2.4 OpenTOSCA . . . 28
- 2.5 OASIS WS-BPEL 2.0 . . . 31
- 2.6 Topology Splitting . . . 34

3 Related Work **37**
- 3.1 Model Based Provisioning and Management of Cloud Applications . . . 37
- 3.2 Pattern Based Provisioning and Management of Cloud Applications . . . 39
- 3.3 Provisioning and Management of Cloud Applications in TOSCA . . . 42
- 3.4 Topology Splitting and Matching of Cloud Applications . . . 46

4 Concept and Specification **49**
- 4.1 Concept Overview . . . 49
- 4.2 System Overview . . . 50
- 4.3 Metamodel . . . 53
- 4.4 Running Example . . . 58
- 4.5 Topology Splitting . . . 59
- 4.6 Partial Plan Generation . . . 68
- 4.7 Partial Plan Orchestration . . . 69

5 Design and Implementation **73**
- 5.1 Design Overview . . . 73
- 5.2 Implementation . . . 76

6 Conclusion and Future Work **83**

Bibliography **85**

List of Figures

1.1 Illustration to demonstrate the motivation for the problem statement by comparing the current approach for Application Provisioning using OpenTOSCA with the proposed approach . . . 15
1.2 Illustration to demonstrate a sample use case using an application deployment scenario in a multi-cloud environment . . . 17

2.1 Illustration of the components of the NIST Cloud Model (Based on the illustration in [SAMT14]) . . . 21
2.2 Structural Elements of a Service Template and their Relation (Based on the illustration in [TOSCAb13] . . . 24
2.3 Sample Architecture and Processing Flow of the TOSCA Environment (Based on the illustration in [TOSCAb13] . . . 27
2.4 Architecture and Control Flow of the OpenTOSCA Ecosystem (Based on the illustration in [OTOSCa] . . . 29
2.5 Structural components of an Executable BPEL Process Document . . . 32
2.6 Illustration showing the Lifecycle of an Executable BPEL Process . . . 33
2.7 Illustration depicting the classification of Topology Splits . . . 35
2.8 Illustration showing the difference between Vertical and Horizontal Topology Splits . . . 35

3.1 Illustration showing the steps of Plan generation as discussed in [BBK+14] . 43
3.2 Illustration showing the steps of Scale-Out Plan generation as discussed in [KBL17] . . . 45

4.1 Illustration depicting the High Level Concept proposed in the thesis for Dynamic Provisioning of Cloud Applications . . . 49
4.2 Illustration depicting the High Level System Architecture for the concept introduced in Section 4.1 . . . 50
4.3 Syntactic consistency between property and parameter names for a simple topology that deploys a Docker Container on a pre-configured Docker Engine 52
4.4 Illustration showing an Application Template and its components - Topology, Split Definitions, Partial Plans, and Orchestration Map . . . 58
4.5 Illustration showing high-level steps in the Topology Splitting algorithm. . . 59
4.6 Illustration showing the stages involved in the second step of the Topology Splitting algorithm - "Split Topology Vertically" . . . 61
4.7 Illustration showing the stages involved in the third step of the Topology Splitting algorithm - "Split Topology Horizontally" . . . 63

4.8 Illustration showing the steps involved in Partial Plan Generation from a Split Topology 68
4.9 Illustration showing an Orchestration Map that contains the mapping between Nodes, Split Definitions, and Partial Plans 70
4.10 Illustration showing the different stages of the Partial Plan Orchestration Algorithm. 70

5.1 Illustration showing the sample application topology considered for the prototype 73
5.2 Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17] 75
5.3 Illustration showing the UML class diagram for the CSAR model and Topology model used in the prototype implementation (based on the data model in [KEP13]) 77
5.4 Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17] 78
5.5 Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17] 80
5.6 Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17] 80

List of Algorithms

4.1 Check Topology For User Defined Splits 60
4.2 Split Topology Vertically 62
4.3 Extract Vertical Splits 64
4.4 Check Node Dependency 65
4.5 Check Property Dependency 65
4.6 Generate Split Definition 66
4.7 Remove Duplicate Split Definitions 66
4.8 Split Topology Horizontally 67
4.9 Generate Partial Plans 69
4.10 Orchestrate Partial Plans 72

List of Abbreviations

BPEL Business Process Execution Language. 14

BPMN Business Process Model and Notation. 14

CSAR Cloud Service Archive. 14

DSL domain specific language. 14

EPP Executable Provisioning Plan. 43

EPR Endpoint Reference. 31

IA Implementation Artifact. 30

IaaS Infrastructure as a Service. 13

MDD Model Driven Development. 19

OS operating system. 13

PaaS Platform as a Service. 13

PBD Pattern Based Development. 20

POG Provisioning Order Graph. 43

PPS Provisioning Plan Skeleton. 43

QoS quality of service. 13

SaaS Software as a Service. 13

TOSCA Topology and Orchestration Specification for Cloud Applications. 14

VM virtual machine. 13

WAR web archive file. 13

1 Introduction

Cloud Computing, in recent years, has gained widespread popularity within the IT industry because of the lucrative economic benefits it has to offer and the technical ease and flexibility it provides for IT operations and management. Features of the paradigm such as *high scalability, high availability, and pay-as-you-go pricing models* [MG11] have led to a significant shift in the way today applications are developed and deployed. To add to that, the various service models of the cloud paradigm like Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS) [MG11] have further enabled the ease of development and deployment of complex applications within a short span of time. Thereby, providing companies a competitive edge by significantly reducing the time to value and time to market for their products.

As a result of the increasing adoption of the cloud computing paradigm and the ever-growing scale and complexity of the applications that it supports, it has become imperative to automate their provisioning and management processes. Provisioning, in the present context, means providing requested cloud resources to the user on demand. Under the hood, this may involve orchestrating several complex executable steps and managing different files to provision the resource successfully. E.g., provisioning an instance of a simple cloud-based java web-application in an IaaS scenario may involve the following steps - (i) create a virtual machine (VM), (ii) install an operating system (OS), (iii) setup the necessary Java Runtime Environment, and (iv) deploy the web archive file (WAR) file. Manually handling these steps would be not only time-consuming but also prone to human errors. Therefore, automatic provisioning of cloud applications has been one of the focus areas for research in academia as well as the industry.

Traditionally, mostly the cloud providers needed to automate their internal management processes to achieve the necessary quality of service (QoS) for their offerings [BBK+13]. E.g., provisioning a new VM or setting up a PaaS environment like Amazon Beanstalk[1]. However, today, automation of provisioning is a necessity in modern-day applications built for the cloud owing to their ever-growing scale and complexity. These applications are typically comprised of several components and may consume services from different cloud providers. Moreover, the deployment strategies of these applications may vary based on business requirements. E.g., the front-end of a web application may be outsourced on to a Public Cloud like Amazon AWS[2], whereas the backend is deployed on a Private Cloud within the premises of the company. These factors further add to the complexity

[1] https://aws.amazon.com/elasticbeanstalk/
[2] https://aws.amazon.com/

of the provisioning process. An attempt to manually deploy such applications would require extensive technical know-how of the various proprietary interfaces exposed by cloud vendors and their corresponding data exchange formats [BBK+13]. Moreover, the tight coupling between the application and the cloud service providers may further result in the so-called vendor lock-in problem that affects the portability of the application in a multi-cloud environment [OST16]. A multi-cloud environment provides the possibility of provisioning an application across multiple cloud service providers and multiple cloud offering e.g., Private clouds and Public clouds.

Several configuration management tools have been developed, in recent years, to tackle the problem of automatic application provisioning and deployment, like CHEF[3], Puppet[4], Ansible[5], and Saltstack[6]. These tools use proprietary domain specific languages (DSLs) to specify the provisioning and management steps for the applications declaratively. These steps are then converted into executable code that are orchestrated (e.g., using workflows) to provision the application successfully. Additionally, AWS Cloud Formation[7] is a proprietary orchestration based cloud application provisioning service offered by Amazon. It works similarly as the configuration management tools discussed above, but it lets the users define only AWS resources (EC2[8], or S3[9] Buckets) using its JSON compliant DSL. Although these tools solve a part of the problem by automating the provisioning process, they do not address the issue of application portability in a multi-cloud environment or the interoperability of the provisioning specifications that are represented by the proprietary domain specific languages.

Therefore, to address the issues mentioned above, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [TCATTC] standard was introduced. "TOSCA is an OASIS open standard that defines the interoperable description of services and applications hosted on the cloud and elsewhere; including their components, relationships, dependencies, requirements, and capabilities, thereby enabling portability and automated management across cloud providers regardless of underlying platform or infrastructure; thus expanding customer choice, improving reliability, and reducing cost and time-to-value" [OTTC]. TOSCA enables the description of cloud applications as XML [BPS+08] based definition files. These definition files define the overall structure and dependencies between the various components of the application. The structure of the application is defined by a so-called *Topology*. The definition files also contain information regarding the artifacts e.g., WAR files, VM images, and shell scripts that are required to install the various components of the application. The provisioning logic in TOSCA is specified in the form of so-called *Plans* that can be workflows implemented using Business Process Execution Language (BPEL) (2.5) or Business Process Model and Notation (BPMN) [OMG11]. All the definition

[3] https://www.chef.io/chef/
[4] https://puppet.com/
[5] https://www.ansible.com/
[6] https://saltstack.com/
[7] https://aws.amazon.com/cloudformation/
[8] https://aws.amazon.com/ec2/
[9] https://aws.amazon.com/s3/

files, the artifacts, and the plans are then packaged into an archive file called Cloud Service Archive (CSAR). The CSAR file can then be deployed to any environment that supports a TOSCA compliant run-time engine, thereby making the application portable as well as inter-operable.

Since its introduction, many TOSCA based solutions have been developed for cloud aplication provisioning and management. One such implementation is OpenTOSCA[10]. OpenTOSCA is an open source TOSCA compliant ecosystem that provides the end-to-end tools chain to model, provision, and manage cloud applications. The OpenTOSCA runtime supports both Imperative and Declarative processing of application management logic [BBK+14]. For the scope of this thesis, we will limit our discussion to the later. Declarative processing, in the context of the TOSCA standard, refers to the automatic provisioning by interpreting the topology of the application embedded in one the definition files that are packaged in the CSAR. Declarative processing, in OpenTOSCA is supported by the Plan Generator component of the runtime environment that generates BPEL plans for the provisioning and management of the application.

1.1 Problem Statement

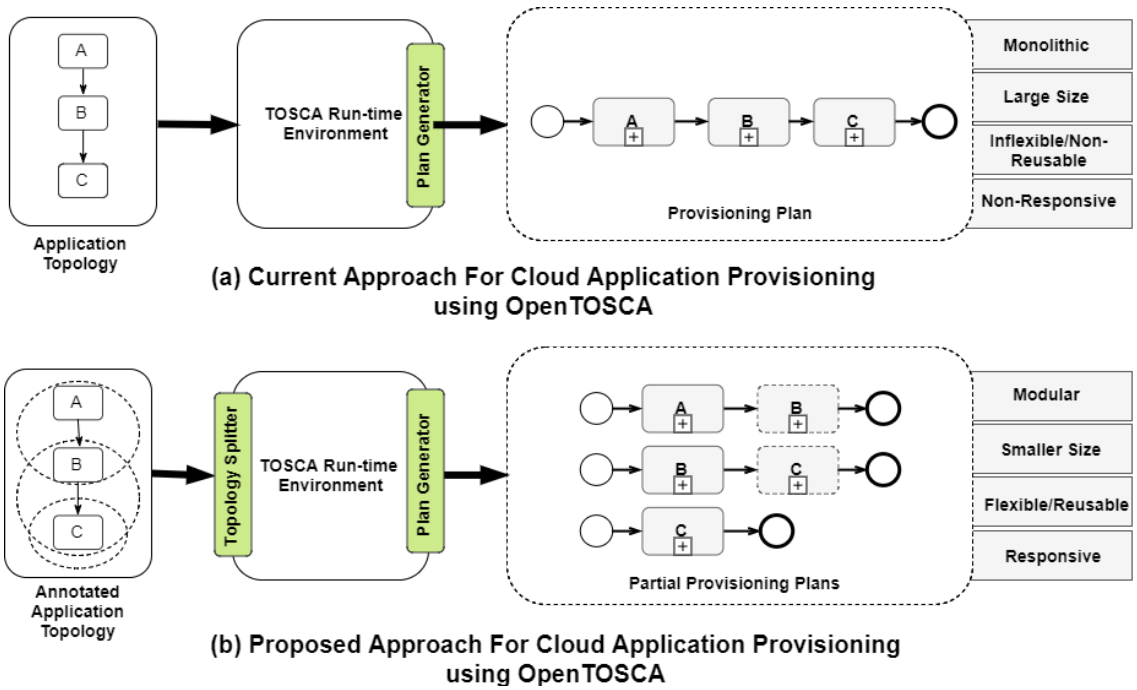

Figure 1.1: Illustration to demonstrate the motivation for the problem statement by comparing the current approach for Application Provisioning using OpenTOSCA with the proposed approach

[10] http://www.opentosca.org/

As mentioned in the previous section, the Plan Builder component of the OpenTOSCA runtime automatically generates BPEL plans by interpreting the application structure and its dependencies from the definition files packaged in the CSAR. Although, this approach aids in solving the problem of automatic provisioning of cloud applications, in its current form, it suffers from a few drawbacks highlighted next.

- Firstly, the BPEL plans generated are monolithic, i.e., the plan builder creates a single plan to provision the entire application including all of its components. As a result of which, with the increase in complexity and size of the cloud application, the complexity and size of the BPEL plan also increases significantly.
- Secondly, execution of BPEL plans in a compliant execution engine like Apache ODE, or WSO2 BPS can be a time and resource-intensive process. Therefore, having large and complex plans may further result in performance overload in situations where the available computing resources are limited, e.g., in IoT devices like Raspberry Pi.
- Thirdly, the plans generated by the plan builder are tightly coupled to the application, i.e., the plans are not generic and therefore cannot be reused for provisioning other applications [BBK+14].
- Finally, OpenTOSCA, in its current form, does not support dynamic provisioning. Dynamic provisioning refers to the ability of the service/application provider to provide resources to the user at runtime. The plans generated by the plan builder only allow for provisioning instances of the complete application. They do not support provisioning parts of the application during runtime which affects the flexibility of the system in a dynamic and scalable environment.

The purpose of this thesis is to explore the possibility of provisioning an application using multiple modular plans instead of a single monolithic plan. This in turn would provide a possible approach to alleviate some of the problems mentioned above that are related to monolithic plans while also acting as a proof of concept for the more complex issue of dynamic provisioning in OpenTOSCA.

1.2 Motivating Scenario

In order to motivate the problem statement, let us consider a simple cloud application as depicted by the Topology in Figure 1.2. The application comprises a PHP Web Application deployed on an Ubuntu VM with an Apache Web Server installed on it. The PHP Application in turn connects to a MySQL DB that is deployed on an Ubuntu VM with a MySQL RDBMS installed on it. Based on the business requirements, it is assumed that the PHP Web Application will be deployed on a *Public Cloud*, while the database will be deployed on a *Private Cloud* for data security. Moreover, the provisioning of the application is handled by a TOSCA run-time environment (e.g. OpenTOSCA) managed by a third party as shown in the figure.

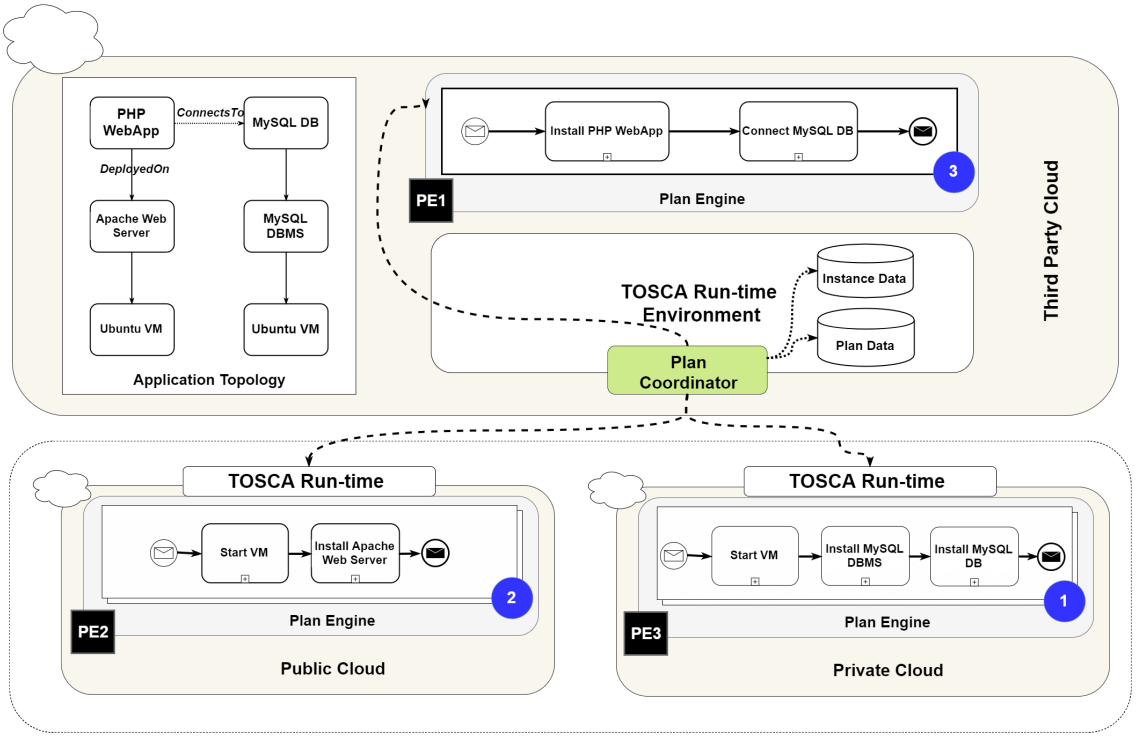

Figure 1.2: Illustration to demonstrate a sample use case using an application deployment scenario in a multi-cloud environment

Traditionally, the TOSCA run-time environment would generate a single plan for the application topology. The plan would then be deployed on the local plan engine (PE1, Figure 1.2) and the tasks will be orchestrated to provision the complete application. But the drawbacks of such an approach has already been detailed in the previous section (1.1). In this example, we consider that the run-time engine is capable of generating modular plans as labeled 1-3 in the figure. These plans are then deployed to the respective environments, e.g. Plan-1, that installs a database on a VM is deployed to PE-3 on the private cloud. Whereas, Plan-2, that installs an Apache Web Server on a VM is deployed to the plan engine (PE-2) of the public cloud. A third plan (P3) that installs a PHP Web Application and connects it to the installed Database instance, is deployed locally at PE1. Then, the Plan Coordinator component of the TOSCA run-time engine orchestrates the modular plans in order to create an instance of the complete application.

1.3 Scope of Work

This Master Thesis deals with the process of generation of modular plans for a given application followed by their orchestration for the successful provisioning of the application in TOSCA. The main contribution of this thesis is the development of an abstract concept for achieving the above mentioned goal. The feasibility of the concept is also shown with the

help of a prototype developed within the OpenTOSCA framework using the OpenTOSCA container APIs.

1.4 Thesis Outline

This master thesis is structured as follows:

Chapter 1 – Introduction: The motivation for the thesis is explained with a basic introduction of the TOSCA standard and OpenTOSCA ecosystem. The problem statement of the thesis is discussed and the scope of the thesis is also outlined.

Chapter 2 – Fundamentals The important terms and technologies are discussed that are necessary to understand the concepts introduced in the thesis.

Chapter 3 – Related Work The various research work related to cloud application provisioning and management, application topology splitting, and generation of scale-out plans in TOSCA are discussed.

Chapter 4 – Concept and Specification The high-level concepts behind the thesis are explained. The general architecture of the approach is outlined and the building blocks are discussed in detail.

Chapter 5 – Design and Implementation The low-level implementation of the concepts and the design of each individual components are described.

Chapter 6 – Conclusion and Future Work In this chapter, a summary of the results of the work is presented as well as an outlook for future work is provided.

2 Fundamentals

In this chapter, we discuss the various terms and technologies that are necessary for the better understanding of the concepts introduced in the thesis. In Section 2.2 we take a look at the cloud computing paradigm and elaborate on its various features followed by a brief overview of the OASIS TOSCA standard in Section 2.3. Then in Section 2.4 we discuss the various components of the OpenTOSCA runtime engine with a focus on the Plan Generator component that is augmented by the concepts introduced in Chapter 4. Finally in Section 2.5 we highlight the essential features of WS-BPEL 2.0 that provide a language for the specification of Executable and Abstract business processes and is used by the OpenTOSCA runtime to generate executable plans.

2.1 Important Terms

Provisioning Provisioning, in general, can be defined as the process of making something available. It is the fourth step in the Operations, Administration, Maintenance and Provisioning (OAMP)[1] management framework. Provisioning can be defined as the configuration and deployment of heterogeneous IT resources. Specifically, cloud provisioning can be described as the configuration, management, and deployment of cloud resources to deliver a cloud application to the user.

Orchestration Orchestration is a term found in close association with Provisioning and can be defined as the automated arrangement, management, and coordination of complex software systems. In the context of cloud computing, orchestration can be viewed as the process of defining the architecture of the service, binding software and hardware components, and finally automating and coordinating workflows to deliver the service.

Service Bus Service Bus is a software component that implements the communication between heterogeneous software systems in a Service Oriented Architecture (SOA).

[1] The OAMP management framework describes the processes, activities, tools, and standards involved in operating, administering, managing and maintaining any system (Source: https://en.wikipedia.org/)

Model Driven Development (MDD) It is a software development paradigm supported by the Model-driven Architecture (MDA) methodology, a software design approach released by the Object Management Group (OMG) [OMG14]. MDA provides a set of guidelines for structuring specifications expressed as models. It starts with a platform-independent model (PIM), continuing through an appropriate domain-specific language (DSL), and then transforming into one or more platform-specific models (PSMs) for the actual implementation platform, e.g., Java™ 2 Platform or.Net, implemented in a general programming language such as Java™, C#, and Python. MDD primarily focuses on Model Transformations and Code Generation. [Yu06].

Pattern Based Development (PBD) It is a software development paradigm that uses accepted solutions to recurring problems within a given context (pattern). Patterns encapsulate a designer's time, skill, and knowledge to solve a software problem. By designing around a particular design pattern, developers have greater flexibility to control the generated code, which may be a constraint of MDD. [Yu06]

2.2 Cloud Computing

The National Institute of Standards and Technology (NIST) has defined Cloud Computing as follows, "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" [MG11]. As illustrated in Figure 2.1, The NIST model of Cloud computing is further comprised of (i) five essential characteristics, (ii) three Service Models, and (iii) four Deployment Models. We will discuss these in details in the subsequent sections.

2.2.1 Characteristics

The NIST Model defines five essentials characteristics of cloud computing as listed below,

1. **On-Demand Self Service:** "A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider." [MG11]
2. **Broad Network Access:** "Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations)." [MG11]
3. **Resource Pooling:** "The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge

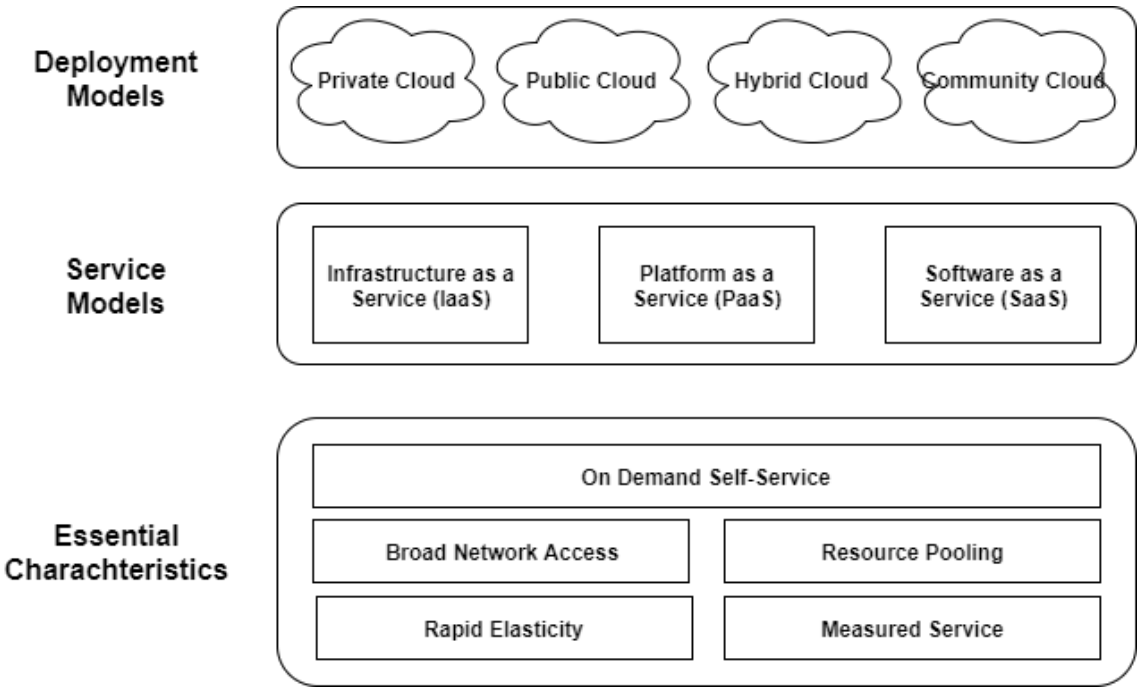

Figure 2.1: Illustration of the components of the NIST Cloud Model (Based on the illustration in [SAMT14])

over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth." [MG11]

4. **Rapid Elasticity:** "Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time." [MG11]
5. **Measured Service:** "Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service." [MG11]

2.2.2 Service Models

The NIST Model defines three Service Models for cloud computing as described below,

1. **Infrastructure as a Service (IaaS):** In this service model, the service provider abstracts the management of the underlying cloud resources such as servers, storage, network through virtualization. The consumers are capable of provisioning fundamental resources such as VMs, OS, storage, and networks which they can deploy and run their applications. Amazon AWS[2], Microsoft Azure[3], and Google Cloud Platform[4] are some of the popular IaaS providers.
2. **Platform as a Service (PaaS):** Unlike IaaS that provides virtualized infrastructure to the consumers, this service model provides complete software platforms and runtime environments. The consumers are capable of deploying applications developed using different languages, libraries, services, and tools that are supported by the provider. The consumer has no control over the underlying cloud infrastructure. Some examples of PaaS solutions are Amazon Beanstalk[5], Google App Engine[6], and Heroku[7].
3. **Software as a Service (SaaS):** This is one of the most commonly found service model in use today for consumer applications. In this service model, the consumers are capable of using a provider's application deployed on the provider's infrastructure. Some of the commonly found SaaS solutions are Google Docs[8], Office 365[9] etc.

2.2.3 Deployment Models

The NIST Model defines four Deployment Models for cloud computing as listed below,

1. **Private Cloud:** In this deployment model, "the cloud infrastructure[10] 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." [MG11]

[2] https://aws.amazon.com/
[3] https://azure.microsoft.com/
[4] https://cloud.google.com/
[5] https://aws.amazon.com/elasticbeanstalk/
[6] https://cloud.google.com/appengine/
[7] https://www.heroku.com/
[8] https://www.google.com/docs/about/
[9] https://www.office.com/
[10] A cloud infrastructure is the collection of hardware and software that enables the five essential characteristics of cloud computing. The cloud infrastructure can be viewed as containing both a physical layer and an abstraction layer. The physical layer consists of the hardware resources that are necessary to support the cloud services being provided and typically includes server, storage, and network components. The abstraction layer consists of the software deployed across the physical layer, which manifests the essential cloud characteristics. Conceptually the abstraction layer sits above the physical layer.[MG11]

Considering the pool of resources are privately owned by a single organization it offers higher control, security, as well as data privacy.

2. **Public Cloud:** This is one of the most commonly found deployment models today. In this model, "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." [MG11]
3. **Community Cloud:** In this model, a group of organizations gets exclusive usage rights on the provisioned cloud infrastructure. These organizations may have shared concerns like mission, security requirements, policy, and compliance considerations. It may exist on or off premise and can be owned by one of the organizations of the community, a third party, or a combination of them.
4. **Hybrid Cloud:** In this deployment model,"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)." [MG11]

2.3 OASIS TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [TCATTC] is an official standard introduced by the Organization for the Advancement of Structured Information Standards (OASIS) consortium. The goal of the standard was to address the growing complexity of Cloud applications and enable a standardized way to design and manage the applications in a portable and inter-operable manner [TOSCAa13]. In the subsequent sections, we discuss the various concepts and components of the TOSCA specification that are most relevant to the understanding of the concepts presented in the Chapter 4 and 5

2.3.1 Structure

The TOSCA specification is an XML based meta-model for defining IT services [TOSCAa13]. It defines both the structure of the service as well as the necessary definitions of the operations for the management of such services [ZBL17]. The structural components of the metamodel (Figure 2.2) are discussed below. It should be noted that some of the components, e.g., Capability and Requirement Definitions, have been omitted from Figure 2.2 for clarity. Moreover, these components are not necessary for the understanding of the concepts in this thesis.

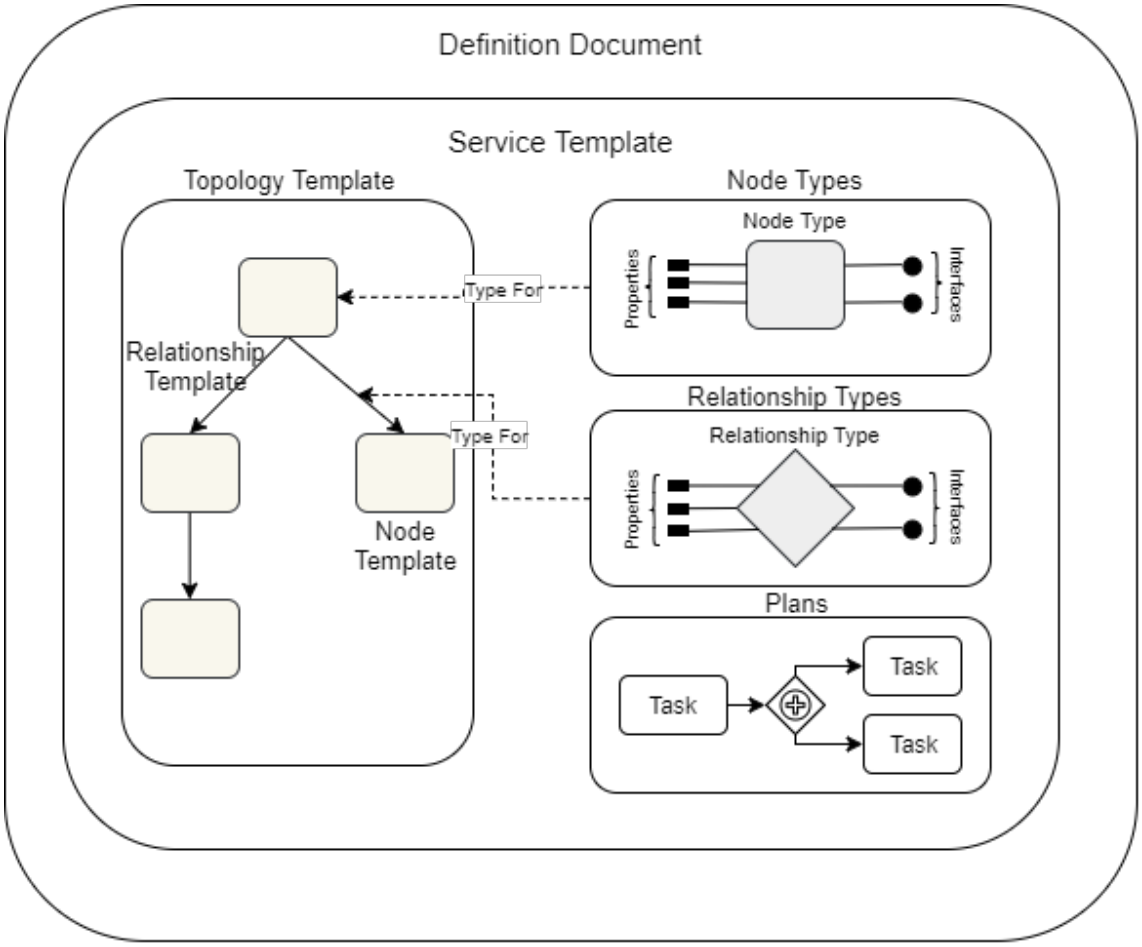

Figure 2.2: Structural Elements of a Service Template and their Relation (Based on the illustration in [TOSCAb13]

Definitions Document A definition document is the most fundamental unit of the TOSCA specification. It is an XML file that encapsulates all the elements needed to define the Service Template for the application e.g. Node Templates, Relationship Templates, Note Type, Relationship Type, etc. A service template also needs to be defined within a definitions document.

Service Template A service is defined by definitions document named Service Template. The service template provides the end to end information that is necessary to create and manage instances of the service in the cloud. The two primary components of the Service Template are the Topology Template and Plans. The Topology Template defines the structure and components of the service, and the Plans define processes that are necessary to deploy, manage, and terminate the service during its lifetime.

Topology Template The Topology Template is a directed graph (not necessarily connected) comprising of nodes and edges. The nodes are represented by **Node Templates** which in turn represent a component of the application, e.g., a PHP application, a Docker container, or a virtual machine. The edges in the Topology Template are represented by **Relationship Templates** which further represent the dependencies between the various Node Templates, e.g., hostedOn, dependsOn or connectsTo.

Node Types and Relationship Types Furthermore, the TOSCA standard promotes reusability of the Node Templates and Relationship Templates through Node Types and Relationship Types which are used to define the semantics of the node and relationship templates respectively [TOSCAa13]. The Node Types define **Properties**, e.g. credentials for a database node, and **Operations** e.g. 'CreateVM', 'StartVM', ShutdownVM via **Interfaces** for a VM node. The same semantics hold for Relationship Types as well.

Implementation Artifacts and Deployment Artifacts The management operations defined within interfaces in the Node and Relationship Types are implemented by artifacts called Implementation Artifacts (IA). They can either be simple Shell Scripts or may be packaged into WAR files. Then, there are some artifacts that implement the business logic of a given node and are represented by Deployment Artifacts (DA). For example, an image file to instantiate a VM or a Node JS script implementing a simple application.

Plans Additionally, the Topology Templates defined in the Service Templates are instantiated using special plans referred to as build plans. These plans defined within the Service Template facilitate the automated management of the application. Based on the activity performed, these plans can broadly be categorized as (i) build plans, (ii) management plans, and (iii) termination plans. These plans orchestrate the management operations defined in the node/relationship types and templates to provision, manage, and terminate instances of the application respectively. TOSCA does not specify a particular process modeling language for the definition of plans, however, recommends to use a workflow language such as the Business Process Execution Language (BPEL) for the Advancement of Structured Information Standards (OASIS) [BPEP07] the Business Process Model and Notation (BPMN) [OMG11].

CSARs Further, TOSCA ensures the portability of applications by packaging the Service Templates, type definitions, artifacts and management plans necessary to provision the application into a self-contained archive file called Cloud Service Archive (CSAR). This archive can be executed by all standard-compliant TOSCA Runtime Environments, e.g., OpenTOSCA (Binz et al.,2013), and, thus, ensures the application's portability and interoperability.

2.3.2 Processing Environment

A TOSCA environment, offered by a Cloud Provider, may contain a set of features that are necessary to process the definitions according to the specification [TOSCAb13]. These features can be grouped into various logical components that in turn are grouped to define the architecture of the TOSCA environment. Moreover, different service providers may choose to include some or all the features in their respective offerings thereby resulting in different TOSCA architectures. A sample architecture is depicted in Figure 2.3 and its components and processing steps are discussed in the following section for a general understanding of the TOSCA processing environment.

To begin with, a (graphical) *TOSCA Modeling Tool* is used to create CSAR files that package the cloud application to be provisioned. Even though this is an optional component, its use is recommended.

Once, the CSAR is created, it can either be processed (i) imperatively, or (ii) declaratively. ***Imperative Processing*** requires that the service template contains the complete topology of the cloud application, and also all of its management behaviors is explicitly defined using plans. On the other hand, ***Declarative Processing*** assumes that it is possible to extract the management behavior of the cloud application from its topology; this requires that the semantics of the node types and the relationship types are defined precisely and that the TOSCA environment is capable of interpreting them correctly.

Further, processing in a TOSCA environment can be broadly divided into two stages - *(i) CSAR Deployment*, and *(ii) Application Instance Management*. These stages are discussed briefly in the following sections.

CSAR Deployment

In this stage, the CSAR file is processed, all its components are extracted, and the environment for the management of the cloud application is set up. The process comprises the following steps,

1. The CSAR file is processed by the *CSAR Processor* such that it can be initially deployed into the environment (step 1, Figure 2.3). This component contains the logic to parse the CSAR file correctly based on the semantics proposed by the TOSCA specification.
2. CSAR Processor interacts with the *Model Interpretor* (in the case of Declarative Processing) to determine the management behavior of the application (step 2, Figure 2.3).
3. The CSAR Processor extracts all the definition files from the CSAR and passes them on to the *Definition Manager* (step 3, Figure 2.3). The definitions manager then stores all the definition files in the *Model Repository* to be retrieved and used later (step 4, Figure 2.3).

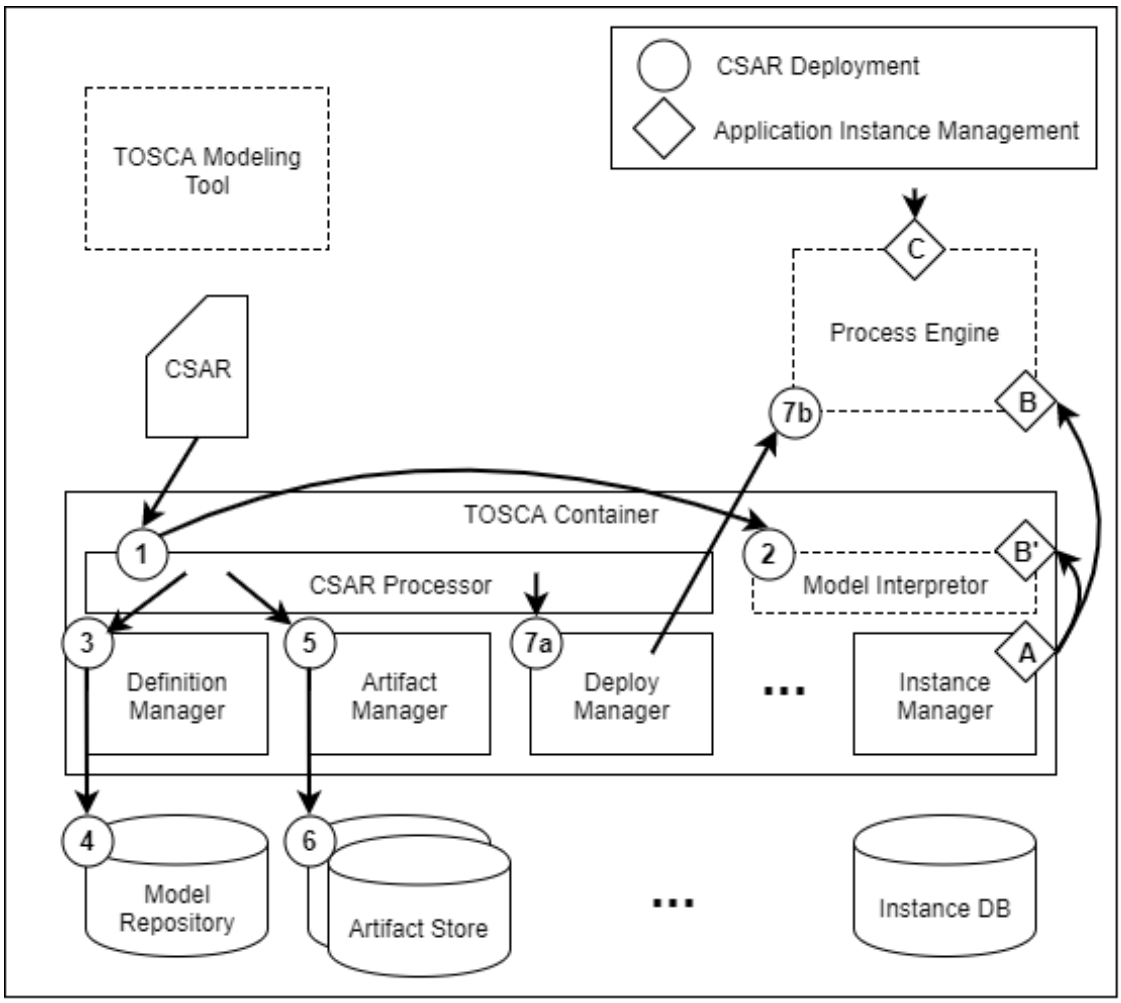

Figure 2.3: Sample Architecture and Processing Flow of the TOSCA Environment (Based on the illustration in [TOSCAb13]

4. Furthermore, the CSAR processor extracts all the implementation and deployment artifacts and passes them to the *Artifact Manager* (step 5, Figure 2.3). The artifact manager then stores them in the appropriate *Artifact Store* (step 6, Figure 2.3), e.g,. storing Virtual Images in virtual image directories, or Docker images in a Docker Repository, etc.

5. Next, the *Deploy Manager* deploys all the implementation artifacts into the environment thus making the executables for all the operations of the node types and relationship types available and ready for use (step 7a, Figure 2.3).

6. In case of imperative processing, Step-2 above is skipped, and an additional step is added at the end. In this step, all the plans available within the CSAR are deployed into the *Process Engine* (step 7b, Figure 2.3). Deploying a plan includes binding the tasks in the plan to the concrete endpoints in the environment which host the operations of the node and relations via the deployed implementation artifacts.

Application Instance Management

Once the environment is set up after the successful deployment of the CSAR, instances of the cloud application can be created, updated, and terminated. The possible steps in this process are listed below,

1. The process of instance creation is handled by the *Instance Manager* (step A, Figure 2.3)
2. It can be done by either invoking deployed *Build Plans* in the Process Engine in case of imperative processing (step B, Figure 2.3), or by interacting with the Model Interpreter in case of declarative processing (step B', Figure 2.3)
3. The instances of the application and its components are stored in the *Instance DB*.
4. Once the instance of the applications are created, the applications can be managed by calling the respective *Management Plans* mentioned in Service Template and deployed in the Process Engine (step C, Figure 2.3) .
5. Finally, when the application is no longer required, it can be decommissioned by either execution the respective *Termination Plan* in the Process Engine, or by interacting with the Model Interpreter.

2.4 OpenTOSCA

OpenTOSCA is a TOSCA compliant ecosystem for cloud application modeling, deployment, and management. It is a research prototype developed by the Institute of Architecture of Application Systems (IAAS)[11] and the Institute for Parallel and Distributed Systems (IPVS)[12] of the University of Stuttgart, Germany. OpenTOSCA comprises of three major components - (i) Modeling Tool (Eclipse Winery), (ii) Runtime Environment (OpenTOSCA Container), and (iii) Administration and Self-Service Portal (OpenTOSCA UI). Figure 2.4 depicts the architecture and control flow of the ecosystem. It can be seen that OpenTOSCA architecture builds on top of the sample TOSCA environment architecture in Figure 2.3. For this thesis an overview of the OpenTOSCA Container is sufficient. However, for the sake of completeness, we provide a brief overview of the other components as well.

2.4.1 Eclipse Winery

Winery is an open source TOSCA Modeling tool [KBBL13]. It provides a UI for modeling Topology Templates and packaging them into CSARs. Winery also provides a management

[11] http://www.iaas.uni-stuttgart.de/
[12] https://www.ipvs.uni-stuttgart.de/

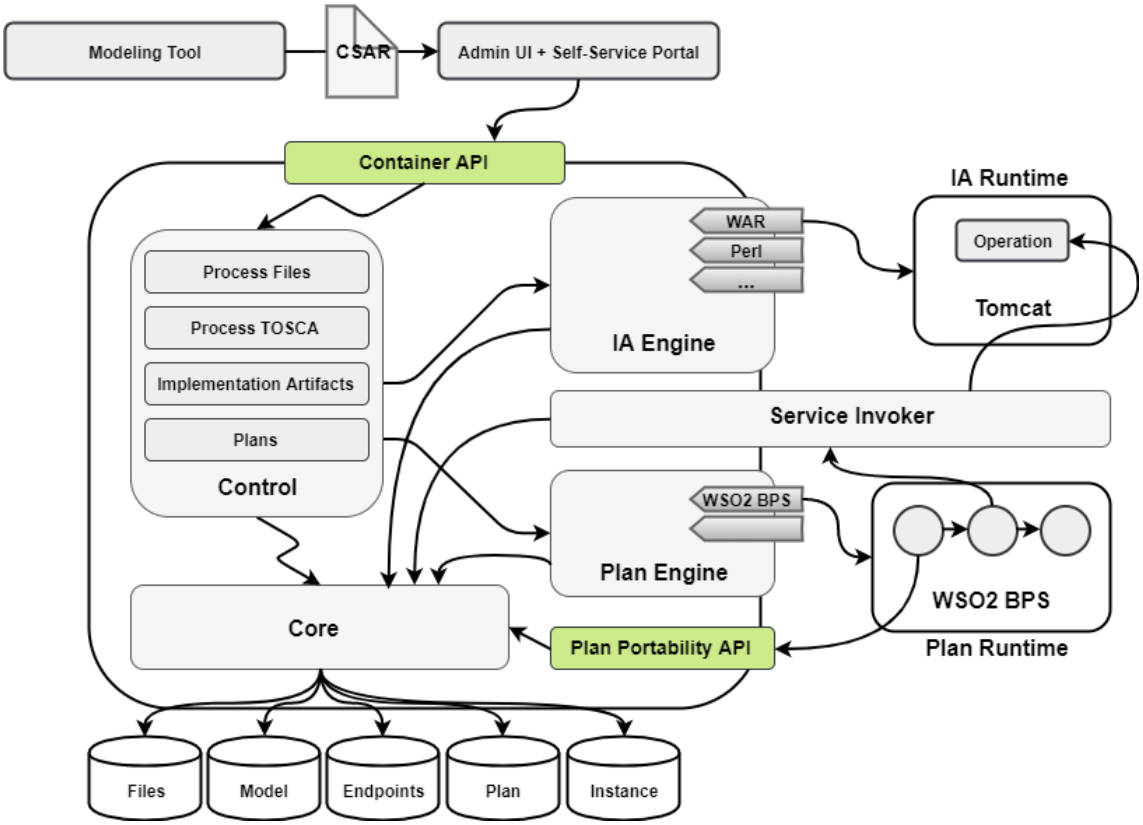

Figure 2.4: Architecture and Control Flow of the OpenTOSCA Ecosystem (Based on the illustration in [OTOSCa]

backend to manage types and artifacts. It also facilitates the import and export of CSARs thereby enabling the creation of new CSARs, and modification of existing ones.

2.4.2 OpenTOSCA UI

The OpenTOSCA UI is a web application developed in NodeJS that combines two features - (i) the Admin UI, and (ii) the Self-Service Portal (formerly known as Vinothek). The Admin UI is used to upload and install a CSAR into the runtime environment, and the Self-Service Portal is used to instantiate, monitor, and manage application instances for the deployed CSARs. The UI interacts with the container via the OpenTOSCA Container API.

2.4.3 OpenTOSCA Container

"The OpenTOSCA Container is the TOSCA Runtime Environment to deploy and manage Cloud applications. It enables the automated provisioning of applications that are modeled using TOSCA and packaged as CSARs" [OTOSCb]. The implementation of the OpenTOSCA container is based on the OSGi framework [OSG08] thereby ensuring extensibility of its

components. As shown in Figure 2.4, *Control, IA Engine, Core, Service Invoker, and Plan Engine* form the core components of the OpenTOSCA container. The *Container API* is used by external components to interact with the container via well-defined REST endpoints. The requests to the API are passed to the control component which manages and orchestrates the other components of the runtime. The Core component offers common services to other components, e. g., managing data or validating XML.

Similar to the concept discussed in (2.3.2), processing in OpenTOSCA can also be broadly divided into two stages - (i) CSAR Deployment, and (ii) Application Instance Management, that are briefly discussed in the following sections.

CSAR Deployment

Once the CSAR is uploaded using the OpenTOSCA UI, the Container API is used to deploy the application into the OpenTOSCA Environment. The following steps outline the control flow of the CSAR deployment process [BBH+13],

1. First, the *Control* component unpacks the CSAR, processes the files, and then store them into the *File Store*.
2. Next, all the TOSCA definition documents (XML files) are loaded, validated for proper semantics, and processed by the Control component. Once verified, these definitions are stored in the *Model* repository.
3. The control component then calls the *Implementation Artifact (IA) Engine*. The IA engine deploys the implementation artifacts referenced in the definition files, e.g., SOAP Web Services implemented as WAR files through a supported plugin into a compatible runtime. OpenTOSCA uses an Apache Tomcat server as IA Runtime for WAR files. Once the plugin has successfully deployed the IA and returned the endpoints of the corresponding management operations, the IA Engine stores them in the *Endpoints* database.
4. Next, the *Plan Engine* checks for available plans within the service template (imperative processing) and processes them. The Plan Engine employs plugins that can support different workflow languages like BPEL or BPMN and their corresponding runtime environments. Presently OpenTOSCA supports BPEL workflows and uses WSO2 BPS[13] as its *Plan Runtime*.
5. In case, no plans were found in step 4, the Plan Engine generates the plans from the given topology template (declarative processing) and then deploys the generated plans to the Plan Runtime.

[13] https://wso2.com/products/business-process-server/

Application Instance Management

Once the CSAR is deployed and the environment is setup successfully, the life-cycle of the application can be managed using the OpenTOSCA UI. The following steps outline the control flow of the process [BBH+13],

1. Upon receiving a request to create an instance of the application, the Container API passes it to the Control component which in turn forwards the request to the Plan Engine.
2. The Plan Engine then invokes the respective Build Plan to provision the application instance. The plans use the *Plan Portability API* to access the Topology and Instance values, e.g., node and relationship property values.
3. Application management (e.g., scaling) and decommissioning are handled similarly by invoking the corresponding Management Plans and Termination Plans respectively.

2.5 OASIS WS-BPEL 2.0

WS-BPEL stands for Web Services Business Process Execution Language. It provides a model and the semantics for orchestration of Business Processes using web-services [BPEP07]. The basic concepts of WS-BPEL can be applied in one of the two ways - (i) abstract, and (ii) executable. For the scope thesis, only Executable BPEL Processes will be considered. Executable Processes can be deployed into a compatible Process Engine (e.g., WSO2 BPS) where they execute and interact with their partners in a consistent way through Web Service resources and XML data. For all purposes of discussion, hereon, the terms Plan, BPEL Process, and Executable Business Process will be used interchangeably. In the following section, we will briefly discuss the relevant aspects of the (i) structure and (ii) lifecycle of an executable BPEL Process.

2.5.1 Structure

As depicted in Figure 2.5, a BPEL Process document comprises of the following components - (i) Partner Links, (ii) Variables, (iii) Correlation Sets, and (iv) Activities. It should be noted that the list of components is not exhaustive and certain components have been omitted for clarity.

Partner Links At an abstract level, partner links define the shape of the communication between the business process and a partner. They are further typed by the so-called *Partner Link Types*. Partner link types characterize the relationship between two services by defining

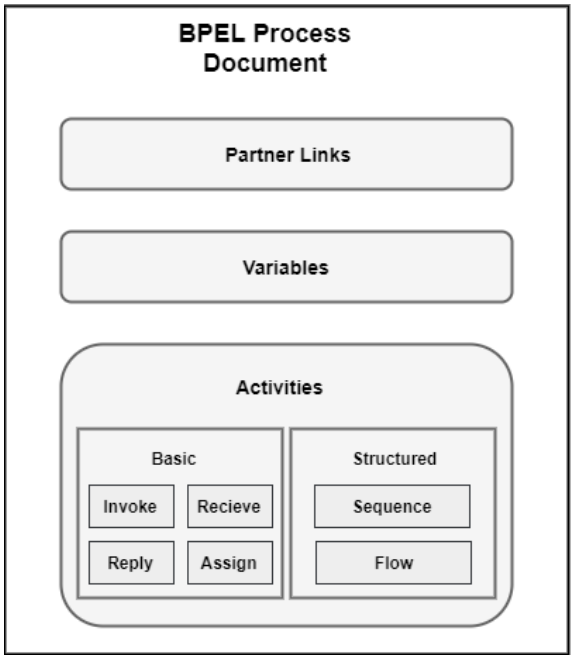

Figure 2.5: Structural components of an Executable BPEL Process Document

the roles played by each of them and specifying the *portType*[14] defined by each service to receive messages within the conversation. However, the concrete services provided by a partner can be dynamically determined using the notion of an Endpoint Reference (EPR) [BPES07].

Variables A variable is the persistent data that is required to maintain the state of the business process, e.g., the messages that are received from the partners, of the messages that are constructed to be sent to the partners. Variables can also hold data that are needed for holding state related to the process and are never exchanged with partners.

Activities In WS-BPEL, Activities are the components that perform the processing logic. They can be broadly categorized into the following two types - (i) Basic Activities, and (ii) Structured Activities.

- **Basic Activities** describe the elemental steps of a Business Process behavior. Some of the relevant basic activities are described below,
  - **Invoke:** This activity is used to invoke an operation exposed by a service. It contains the *inputVariable* and *outputVariable* attributes that carry the input and the output messages sent to and received from the invoked operation.

[14] A WSDL portType is a set of abstract operations along with the abstract messages involved, that are exposed by a service.

- **Receive:** A BPEL process can be instantiated only if it contains a $receive$ (or $flow$) activity with the *createInstance* attribute set to *true*. It contains a $variable$ attribute that is used to receive the incoming message data. This is a blocking activity and does not complete unless a matching message is received by the instance of the business process.
- **Reply:** This activity is used to send a response to a request that was previously made using a $recieve$ activity. This activity is only valid in case of a $request - reply$ interaction between the process and the partner(s).
- **Assign:** This activity is primarily used for data assignments within a BPEL process. An $assign$ activity enables copying of data from one variable into another, as well as constructing and inserting new values.

- **Structured Activities** describe the order in which a collection of activities is executed. They describe the creation of a business process by coordinating the control flow, fault handling, and message exchanges using nested activities. Some of the relevant structured activities are described below,
  - **Sequence:** This activity contains one or more nested activities that are executed sequentially. Execution of a sequence activity completes when the last nested activity is successfully executed.
  - **Flow:** This activity contains one or more nested activities that are executed concurrently. Execution of a sequence activity completes when all the nested activities are successfully executed. A $flow$ activity may contain $link$ elements that are used for synchronization of the nested activities.

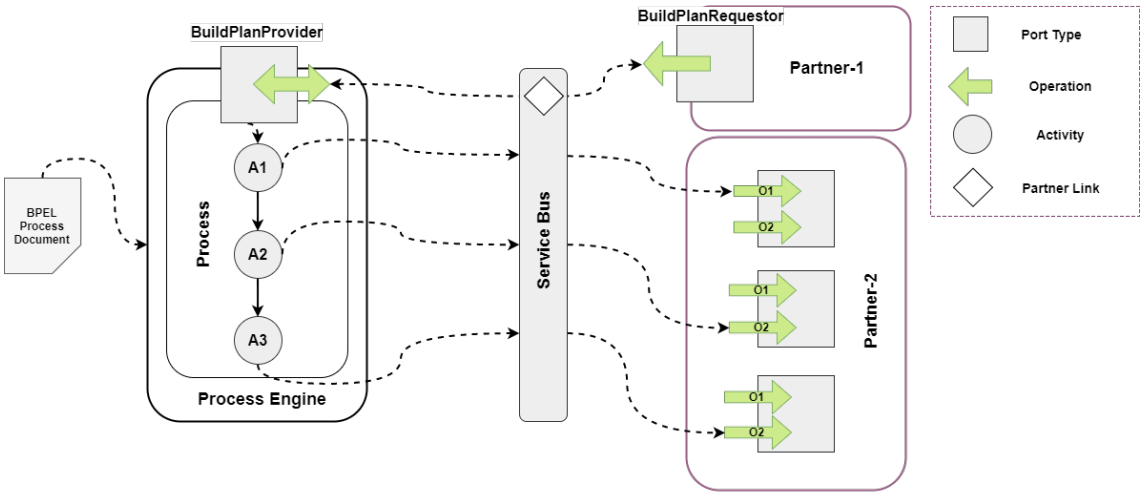

Figure 2.6: Illustration showing the Lifecycle of an Executable BPEL Process

2.5.2 Lifecycle

As depicted in Figure 2.6, a lifecycle of a BPEL Process comprises of the following steps - (i) Process Deployment, (ii) Process Instantiation, and (iii) Process Termination. These steps are discussed below,

Process Deployment In this step, first, the BPEL Process Document is deployed in the Process Engine. Then the actual binding information about the partner services is determined dynamically via assignment of endpoint references which contain the information of the Partner Links defined in the Process Document.

Process Instantiation Once the binding information of services is available, an instance of the process can be implicitly created by receiving messages via the Port Types defined for the Partner Role assigned to the process. As discussed in 2.5.1, a Receive activity with $createInstance$ attribute set to $true$ initiates the instance creation for the process.

Process Termination A BPEL Process can terminate in two ways - (i) *Normal Termination* when the root activity completes execution, or (ii) *Abnormal Termination* when a fault is encountered or an $exit$ activity is used explicitly.

2.6 Topology Splitting

An application topology can be split in two ways - *physically or conceptually*. Therefore, within the context of this thesis, *Splits* are broadly classified into the following two categories - *Physical Splits and Conceptual Splits*. These categories are further classified into two sub-categories viz. *Vertical Splits and Horizontal Splits*. This classification is depicted graphically in Figure 2.8. For the algorithms presented in Section 4.5.2 and for all purposes of discussion hereon, the term *Split* would refer to a *Conceptual Split* unless otherwise mentioned.Moreover, a *Conceptual Split* is formally represented by a *Split Definition* (Definition 4.3.17, Section 4.3.1). A brief description of each category is provided below.

2.6.1 Physical Split vs. Conceptual Split

Physical Splits as the name suggests, physically alter the given topology. The topology file is split into multiple smaller topology files, each containing a subset of the original topology elements. The split topology files are then iteratively provided as input to the Plan Generator component (4.6) in order to generate the partial plans. These partial plans can then be orchestrated in the correct order to provision the application.

Conceptual Splits on the other hand preserve the original topology file. Instead of generating multiple granular topology files, splits are defined in the same topology file by

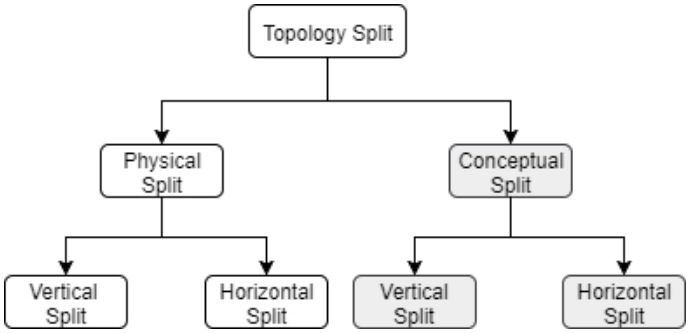

Figure 2.7: Illustration depicting the classification of Topology Splits

annotating topology elements similar to [SBKL17] or by defining *Split Definitions* (4.3.1). Split definitions are basically meta-data contain the information about individual splits, e.g, the nodes and relations contained within the split. The annotated topology is provided as an input to the the Plan Generator component. The Plan Generator interprets the split definitions to generate the partial plans that are then orchestrated for provisioning the application.

2.6.2 Vertical Splits vs. Horizontal Splits

Vertical/Horizontal Splits depend on the relationships within a topology.We primarily consider two base Relationship Types in this thesis - *HostedOn and ConnectsTo*.

Vertical Split refers to a split in the topology across a *ConnectsTo* Relationship Type or its derivatives. This is depicted in Figure 2.8(b).

Horizontal Split refers to a split in the topology across a *HostedOn or DependsOn* Relationship Type or their derivatives. This is depicted in Figure 2.8(a).

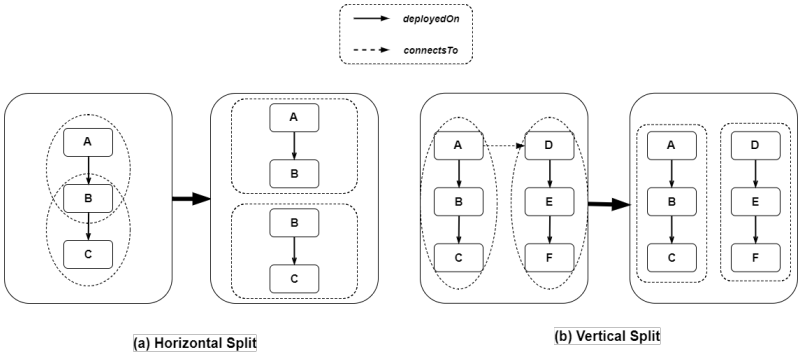

Figure 2.8: Illustration showing the difference between Vertical and Horizontal Topology Splits

3 Related Work

In this chapter we discuss the related research work done that is relevant to the concepts presented in this thesis. First, we explore some of the model (3.1) and pattern (3.2) based approaches for provisioning and management of cloud applications, followed by a look at some of the work done in the area of provisioning using the TOSCA standard (3.3). Finally, in Section 3.4 we take a look at a topology splitting and matching technique for deploying applications in a multi-cloud environment. Also, for each of the work discussed, we will highlight their limitations, if any, and also underscore their influences on the work done in this thesis.

3.1 Model Based Provisioning and Management of Cloud Applications

3.1.1 Pattern-based Composite Application Deployment

Eilam et al. [EEKS11] in the paper titled Pattern-based Composite Application Deployment propose a model-based approach for cloud application deployment by bridging the gap between high-level non-executable deployment models and low-level executable workflows.

Motivation

Deployment methodologies for provisioning composite enterprise applications can be broadly categorized into *workflow-based* approaches and *model-driven design* [EEKS11]. Although the workflow based approach is more popular than model-driven approaches, it suffers from high sensitivity to topological changes in the application and design complexity. On the other hand, while the model-driven approach shows promise by providing a high-level of abstraction, separation of concerns, and strong semantics, they suffer from the fact that the output of such a model is not directly executable. Therefore, Eilam et al. in their paper propose a method to integrate both the approaches to provide high flexibility in modeling complex applications while automatically generating reusable workflows to deploy them.

Approach

The proposed approach in [EEKS11], on a high level, comprises of defining a ***deployment model*** that represents the desired deployment state of an application and then the automated transformation of the model into an executable workflow. The conversion involves two steps - (i) identifying a set of operations which will affect the desired state, and (ii) determining the partial order in which they can be executed. The operations are represented as ***automation signatures*** which comprise of a model pattern that describes the effect of the operation on components of the given model, and its requirements on related components. E.g., an operation to install a docker engine on a VM may require the IP address of the VM. Automation signatures, therefore, provide the link between the models and the underlying executable operation logic (implemented as scripts or workflows).

The deployment model consists of a ***Topology*** that is used to define the desired state of the application. The Topology further comprises of typed ***nodes*** and ***edges*** that represent ***resources*** which are to be installed. Furthermore, the operations declare automation signatures that define the requirements of their operation and their effects in a semantically rich manner. An automation signature is represented as a 3-tuple $A = (O, P, L)$ where O is the Operation, P is a deployment model known as *pattern*, and L represents the set of relations between the Operation and the Pattern.

To generate a workflow for a given deployment model, the problem is modeled as a set cover algorithm. Firstly, all the ***targeted nodes*** are obtained from the model. Targeted nodes are a subset of the deployment model for which *initstate* $\neq$ *goalstate*. This subset is treated as the universe *U*. Then, a collection *S* is obtained which consists of all possible implementations of automation signatures for the targeted nodes. The algorithm then finds a sub-family $C \subseteq S$ that covers the Universe U, such that none of the sets in C overlap. Finally, the partial order of the operations in C is found by analyzing the requirements expressed in their automation signatures and incorporating available domain knowledge. Once the partial ordering is obtained, they can be flattened to a total order.

Summary and Influences

The workflow generation algorithms proposed in the paper make use of graph covering techniques to determine the optimal set of operations to instantiate the deployment model. This is done by matching the automation signature patterns with the targeted nodes. We will use a variant of the algorithm in Section 4.5.2 to determine dependencies between different nodes in a topology to group them into logical units called *Split Definition*.

3.2 Pattern Based Provisioning and Management of Cloud Applications

3.2.1 Pattern-based Deployment Service for Next Generation Clouds

In the paper Pattern-based Deployment Service for Next Generation Clouds Lu et al. [LSS+13] provide a service to deploy a multi-cloud application using deployment plans that are represented in the form of patterns. A pattern represents an abstract view of the cloud application that in turn represents the logical units of the application and their corresponding mapping to the actual cloud resources. The service not only instantiates the pattern but also allows for runtime updates to the deployed application.

Motivation

The Pattern Based Deployment Service (PDS) proposed in [LSS+13] tries to overcome two challenges - (a) automated application deployment on a two-tiered (smart edges and core) as well as the multi-cloud environment, and (b) dynamic adaptation of the deployed application.

Approach

On a high level, the PDS takes as an input a ***System Pattern Description (SPD)*** file. The SPD defines the details of the application topology and deployment. Once uploaded, the SPD is parsed by the service and a graph is generated that represents the abstract steps required to deploy the application. The generated graph is then dynamically converted into Chef commands which in turn are executed to instantiate the various nodes of the application topology.

The pattern-based deployment service defines its own XML based Domain Specific language (DSL). The DSL identifies the following elements - *topology, container, node, and service*. The ***topology*** represents the structure and relationship between the various components of the application to be deployed. The ***container*** element represents a collection of nodes. A ***node*** represents a virtual machine instance that has several attributes that, e.g., define the cloud on which the node will be deployed and the services that should be deployed on the given node. The ***service*** element defines the various software components that should be installed/deployed on a given node.

Moreover, a topology may have multiple nodes, and a node can have multiple services (e.g., Database, Web Application, etc.). This, in turn, may result in a scenario where the deployment of one service may require configuration information from another service and so on. E.g. The web application service on one node may need the IP address of the node that hosts the database service to establish a connection. The paper defines such dependencies as ***deployment dependencies***. The DPS uses two algorithms to detect and

resolve such dependencies. On a conceptual level, the first algorithm iterates through the topology to extract all the deployment dependencies and the second algorithm defines the correct ordering of the Chef invocations. Finally, a workflow is defined which makes sure that the Chef recipes are executed in the correct order with the right attributes to make sure that the service dependencies are correctly resolved, and the nodes are successfully deployed.

Summary and Influences

As discussed in the previous section, the PDS converts the system pattern (SPD) represented in the form of an XML-based DSL into a workflow of actions. These actions are specific to Chef and their execution in the defined order provisions the application in a multi-cloud/two-tier cloud environment. But, the approach doesn't talk about granular and reusable workflows (Plans). The workflows generated are monolithic and tightly coupled to the topology defined in the SPD thereby suffering from the drawbacks that were highlighted in Section 1.1.

3.2.2 Pattern-based Runtime Management of Composite Cloud Applications

Breitenbücher et al. [BBKL13] in their paper Pattern-based Runtime Management of Composite Cloud Applications, propose a pattern-based approach to decouple high-level management of cloud applications from their low-level management operations. This, in turn, allows for the re-usability of management knowledge in a dynamic and automated manner for different applications.

Motivation

The paper attempts to tackle the complex task of management of distributed cloud applications. The challenge with such applications is the unavailability of a mapping between the high-level tasks such as scaling-out components of the application and the actual operations that implement the tasks above. As a result, often such management tasks tend to be tightly coupled to the applications while making is hard to reuse the management knowledge for other applications. Therefore, the authors in [BBKL13] define a method in which (i) high level tasks are captured by ***Management Patterns***, (ii) a separate management layer, called ***Management Planlets***, provide executable low level Management tasks, (iii) both the layers are integrated seamlessly to provide the overall Management of the applications.

Approach

On a conceptual level, the approach for execution of management tasks for distributed cloud applications in an automated manner is divided into three steps - (i) Applying Management Patterns, (ii) generating Management Plans by orchestrating Management Planlets, and (iii) execution of the generated plans.

In the first step, high-level management tasks such as scaling-out an application are mapped to Management Patterns which represent the high-level knowledge required to perform the desired management tasks. These Management patterns transform the ***Application State Models (ASM)*** of a running application to the so-called ***Desired Application State Models (DASM)***. The ASM reflects the current state of an application. It contains the *Application Topology* which is conceptually similar to a SPD (Section 3.2.1) and a Topology (Section 3.1). The Application topology is a graph that represents the components of the application and the relationship among them. The components, in turn, may have arbitrary key-value pairs, e.g., the IP address of a Virtual Machine component, that hold the runtime information about them. The DASM, on the other hand, represents the desired state of the application. It is represented by an annotated topology where the annotations may mean *creation, removal of components or execution of domain-specific tasks* on the respective component. Further, Management Patterns comprise of primarily two parts - **(i) Target Topology Fragment**, and **(ii) Topology Transformation**. The Target Topology Fragment represents a subset of a topology, i.e., components and relationships on which the Management Tasks of the respective pattern will be applied. This fragment is used to match the components of an ASM to the pattern. Once a match has been found, the Topology Transformation is applied to the ASM to generate the DASM.

In the second step, a Management Plan is generated from the DASM obtained from the previous step by the Plan Generator. The paper describes three levels of Management granularity viz. **Management Plans, Management Planlets, and Management Operations**. Management Plans are Workflows which execute the management tasks in an automated manner. They represent the highest granularity of the management layer, e.g., Migration of an application from a private cloud to a public cloud, and are tightly coupled to the application and its topology. At the second level of granularity is the Management Planlets that are small single entry single exit workflows which implement low-level management tasks such as deploying a docker container on a Virtual Machine, or starting a virtual machine on a public cloud. They provide reusable and self-contained building blocks for generating Management Plans for different applications. These Management Planlets may need various input parameters to execute successfully. These input parameters may be provided by the user or extracted dynamically from the mapped elements of the corresponding ASM and DASM. At the lowest level, the Management Operations, which is provided by the components offer concrete functionalities for management tasks, e.g., copy a file to an operating system. These are the operations that are aggregated by the Management Planlets and orchestrated to execute the Management Plans.

To summarize, the Management Plan is generated by first, finding a set of Management Planlets that execute the operations in the DASM, and then employing a Partial Order Planning (POP) algorithm [POP94] to generate a sequence of execution for the planlets.

In the third step, the generated Management Plan is provided with the required input parameters and executed to achieve the Desired Application State.

Summary and Influences

The concepts mentioned in this section have a significant influence on the concepts proposed in the thesis. To begin with, a concept similar to the annotation based topology transformation will be used to mark the so-called *Split Definitions*. This annotated topology (identical to a DASM) will then be used by the Plan Engine to generate granular plans from the available sets of generic planlets. Then, these granular plans will be orchestrated to determine the optimum sequence of operation to provision the application.

3.3 Provisioning and Management of Cloud Applications in TOSCA

3.3.1 Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA

In their paper Breitenbücher et al. [BBK+14] introduce a method of automated provisioning for cloud applications that combines both the *declarative* and *imperative* management techniques (Section 2.3.2) based on TOSCA topology models. The paper aims to overcome the drawbacks of both the approaches and benefit from their advantages.

Motivation

As already highlighted in Section 2.3.2, the imperative processing approach enables developers to explicitly define the steps of the provisioning in the form of custom plans. This method provides the application developers with a higher degree of control and facilitates customization of complex management tasks. But, it suffers from two significant drawbacks. One, the process of generating custom plans is labor intensive, expensive and error-prone considering heterogeneous management services need to be orchestrated, and script-centric approaches need to be wrapped, and different data formats need to be handled [BBK+13][EEKS11]. Moreover, the provisioning plans generated are tightly coupled with the corresponding application topology. Any changes to the structure of the topology will require the creation of new plans [EEKS11][BBKL13]. Considering application topologies can get complicated, this approach is not suitable for every situation. The declarative processing approach, on the other hand, alleviates this problem by automating

the plan generation process. But it too suffers from the following drawbacks - (i) the TOSCA runtime environment must be able to interpret the components and management logic to infer the provisioning logic, and (ii) the degree of customization is lost due to automation. Therefore, the approach proposed by Breitenbücher et al. combines both the approaches thereby automating generation process with the help of a *Provisioning Plan Generator* while also enabling manual customization and fine tuning of the generated plans.

Approach

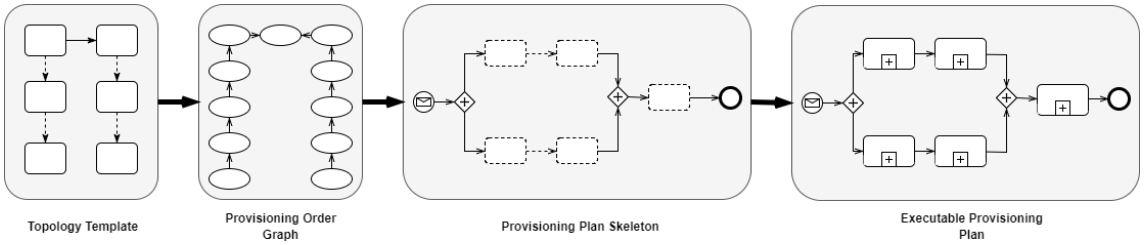

Figure 3.1: Illustration showing the steps of Plan generation as discussed in [BBK+14]

On a conceptual level, the plan generation process works in three steps. These steps are illustrated in Figure 3.1. In the first step, the Topology Template contained in a CSAR is analyzed and a workflow-language independent *Provisioning Order Graph (POG)* is generated. Each vertex in the POG represents a task to provision a component (Node or Relationship Template) of the topology template. Each directed edge defines the temporal provisioning order, i.e., the source vertex needs to be processed before the target vertex. In the second step, a language-specific *Provisioning Plan Skeleton (PPS)* is built based on the POG. The PPS contains *Empty Provisioning Activities* and therefore is not executable. The vertices in the POG are transformed into Empty Provisioning Activities, and the edges define the control flow between the activities. In the final step, the non-executable PPS is converted into an *Executable Provisioning Plan (EPP)* by injecting language specific *Provisioning Subprocess Templates* for individual node and relationship templates.

Summary and Influences

The concepts presented in the previous section for the foundation of the Plan Builder component of OpenTOSCA. Further, a modified version of the Plan Builder component will be used to generate scale-out plans in Section 3.3.2 which in turn will be reused in this thesis for creating modular plans.

3.3.2 Integrating IoT Devices based on Automatically Generated Scale-Out Plans

Képes, Breitenbücher, and Leymann [KBL17] in their paper propose a method to automatically generate *scale-out* plans, based on the TOSCA model, that are capable of adding instances of new elements to the existing topology at run-time.

Motivation

The pervasiveness of IoT applications, today, and their data-intensive nature has led to the use of Cloud Computing resources to design middleware and backend solutions to provide scalability. Moreover, the growing complexity and heterogeneity of IoT devices leads to the complexity of installation, configuration, management, and thus binding of the device to an application. Therefore, to reduce the effort of automatic binding of IoT devices to software, Képes, Breitenbücher, and Leymann propose a TOSCA based approach to mark *Scale-Out Regions* in an application topology. These Scale-Out regions are further converted into *Scale-Out Plans* that create instances of the components inside the marked regions and thereby enabling the automatic installation and configuration of IoT devices.

Approach

Automated plan generation in this context is broadly divided into two steps - (i) defining *Scale-Out Regions* for the topology, and (ii) *Scale-Out Plan* generation.

A **Scale-Out Region** is defined as a connected subset of Node- and Relationship Templates of a given Topology Template, that needs to scaled-out at run-time. The boundary of a scale-out region is defined by the Node Templates annotated with a so-called *Selection Strategy*. A selection strategy defines how a single instance of a Node Template is selected at run-time. Three different selection strategies have been defined in [KBL17] viz. *First Instance , User Provided* and *Workload-based Selection Strategies*. In First Instance Selection strategy, as the name suggests, selects the first found instance of a Node Template. Workload-based selection strategy selects a Node Template instance based on workload. Finally, the User Provided selection strategy selects a Node Template instance that is manually provided by the user.

A **Scale-Out Plan** creates a single instance of the Node- and Relationship Templates circumscribed by the Scale-Out Region, and connects them to the selected instances at the boundary of the scale-out region based on the annotated scaling strategies. Further, Scale-Out Plan Generation is divided into three steps (similar to 3.3.1) - *(i) Scale-Out Order Graph Generation, (ii) Scale-Out Plan Skeleton Transformation, (iii) Scale-Out Plan Completion*.

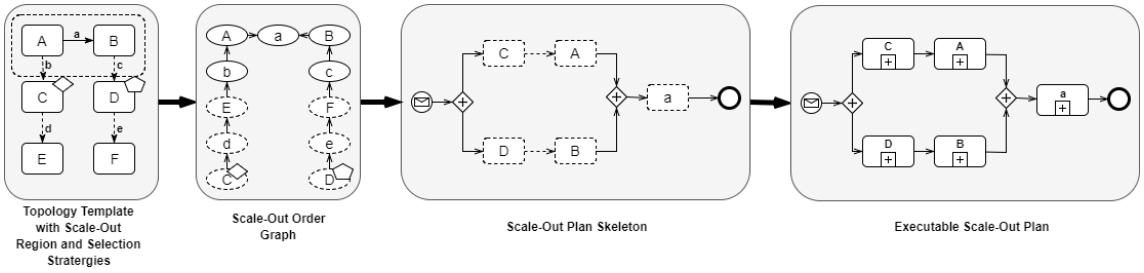

Figure 3.2: Illustration showing the steps of Scale-Out Plan generation as discussed in [KBL17]

1. ***Scale-Out Order Graph Generation:*** In this step, a *Scale-Out Order Graph (SOG)* is generated from the Topology Template that is annotated with the Scale-Out Region and Selection Strategies (Figure 3.2(b)). This step is carried out in two phases. In the first phase, the Provisioning Order Graph Generation algorithm is reused from Section 3.3.1. For each Node and Relationship template marked in the region, an abstract *Provisioning Activity* is added to the SOG. In the next phase, *Recursive Selection Activities* and *Strategic Selection Activities* are added. A Strategic Selection Activity represents the fact that a Node Template instance has to be selected based on the annotated Selection Strategy. Whereas, a Recursive Selection Strategy represents the fact that instances are selected for Node Templates that aren't marked with a selection strategy but are connected to a marked Node Template.
2. ***Scale-Out Plan Skeleton Generation:*** This step is similar to the Provisioning Plan Skeleton Generation algorithm described in Section 3.3.2 where the abstract activities of the POG are replaced with language-specific placeholders.
3. ***Scale-Out Plan Skeleton Completion:*** In this step, a set of *Provisioning Logic Providers* and *Instance Selection Logic Providers* are used to inject executable code into the placeholders to generate an executable plan.

Summary and Influences

The work in the thesis is built on the concepts of this paper. The modular Plans are simply scale-out plans that are generated by first annotating the topology. The Scale-Out Regions and User Defined Selection Strategy are used to define the splits in the topology. Then the scale-out plans created for the marked regions are orchestrated to provision the complete application.

3.4 Topology Splitting and Matching of Cloud Applications

3.4.1 Topology Splitting and Matching for Multi-Cloud Deployments

In the paper Topology Splitting and Matching for Multi-Cloud Deployments, Saatkamp et al. [SBKL17] propose a method to automate the deployment of a cloud application model by splitting the topology following a manually specified distribution in the Business layer. The distribution reflects the changes to the existing topology due to strategic decisions within a company's IT.

Motivation

As we have seen in sections 3.1 and 3.2 cloud applications can be modeled using *deployment models*. To be more specific, as discussed in 3.2.2 applications are represented on a high-level using *topology models*. A topology is a directed graph that consists of nodes which represent a particular component of the application, and the edges represent the relationship between them. Although such a model-driven approach makes it easier to design deployment strategies while abstracting low-level deployment logic, a strategic decision on the business layer, e.g., outsourcing a part of the application from on-premise to a public cloud could make things more complex. The process of adaptation of the topology to such changes if done manually may be not only time-consuming but also error-prone. The solution proposed in [SBKL17] therefore automates the process of splitting the topology to reflect the changes followed by automated deployment of the updated topology.

Approach

In order to achieve the claims discussed in the previous sub-section, Saatkamp et al. introduce the ***Split and Match Method***. The method consists of six steps as follows,

In the first step, the topology model for the application is designed including all the components and the relationships between them. In the second step, *target labels* are added to the application-specific components (nodes). The target labels represent the target on which the corresponding component will be deployed, e.g., *onPrem* for deploying on an on-premise cloud infrastructure, or *AWSPaaS* to deploy the component on a public cloud PaaS offering like AWS Beanstalk. In the third step, the topology model along with the labeled components is checked for whether a splitting is possible. In the fourth step, the topology is split based on the labels. This may result in the duplication of middleware and infrastructure components and corresponding relations of the original topology. Additionally, the target labels of application-specific components are also propagated down to the middleware and infrastructure components. In the fifth step, the split topology is matched against *Provider Repositories* based on their respective labels. The Provider Repositories store information about various components that a provider supports, e.g., an *AWSPaaS* provider repository might contain information about components like *Elastic Beanstalk* and *Amazon RDS*. These

supported components are stored as *component templates*. The labeled components in the split topology are then iterated from bottom to top and are either removed or replaced by a component template based on whether a matching component is found in the repository. In the final step, the resulting topology from step-5 is deployed using a deployment system.

Summary and Influences

The above approach splits the topology vertically which eventually will generate a single monolithic deployment plan. Therefore we reuse the concepts with minor modifications to split the topology vertically as a pre-requisite to Horizontal Splitting. Moreover, the formal definition introduced in the paper will be reused and built upon to formalize the algorithms presented in this thesis.

4 Concept and Specification

The following chapter outlines the concepts presented in this thesis. In Section 4.1 we provide an overview of the core concept followed by the high-level system architecture in Section 4.2. We then introduce a metamodel in Section 4.3 that will be used to formalize the algorithms introduced in the subsequent sections. Then, in Section 4.4 we present a sample application that is used as a running example. For the rest of the chapter we take a closer look at the core components introduced in the System Architecture (4.2) - Topology Splitter (4.5), Plan Generator (4.6), and the Plan Orchestrator (4.7).

4.1 Concept Overview

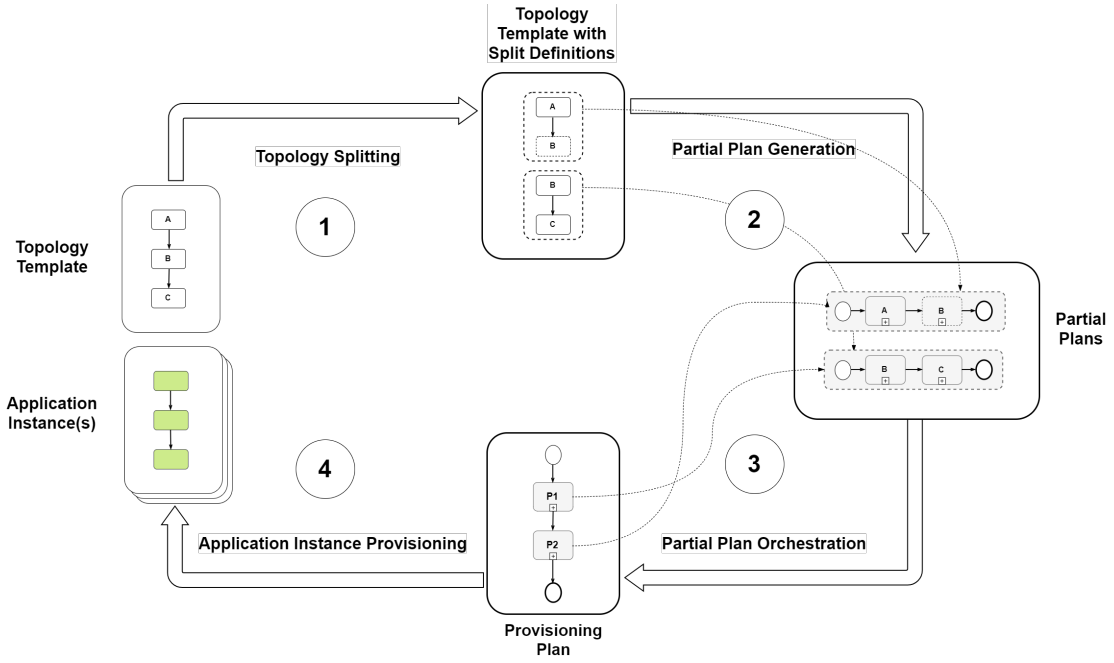

Figure 4.1: Illustration depicting the High Level Concept proposed in the thesis for Dynamic Provisioning of Cloud Applications

As already highlighted in Section 1.3, the primary objective of this thesis is the proposal of a concept to declaratively generate granular plans instead of a single large plan for provisioning a cloud application. The concept can be broadly divided into four steps as illustrated in Figure 4.1. In the first step, the topology of the cloud application is split into

a set of sub-topologies by annotating the elements with a so-called *Split Definition*. Each sub-topology, defined by a *Split Definition*, contains a set of logically grouped nodes and their corresponding relationships. The process of generation of split definitions from a given topology is referred to as *Topology Splitting* (1, Figure 4.1). In the second step, the split definitions are parsed, interpreted and converted into *Partial Plans*. There is a strict one-to-one correlation between split definitions and partial plans, i.e., only one partial plan is generated for every split definition. Each partial plan is responsible for provisioning the sub-topology within the corresponding split definition. The process of generating partial plans from the corresponding split definition is termed as *Partial Plan Generation* (2, Figure 4.1) and is discussed in details in Section 4.6. In the third step, the partial plans are coordinated in the correct order to define the provisioning logic for the cloud application. Each partial plan may take some parameters necessary for provisioning the corresponding components and generate a reference to the instance of the provisioned component(s) as output.This instance reference is used by the next partial plan in the overall provisioning order as an input to access the already provisioned resources. This step is referred to as *Partial Plan Orchestration* (3, Figure 4.1). In the fourth step, the overall provisioning order derived from orchestrating the partial plans is executed to provision an instance of the application. This step is referred to as *application instance provisioning* (4, Figure 4.1).

For the scope of this thesis, only steps 1-3 will be discussed in detail along with the corresponding algorithms in the subsequent sections. In the following section (4.2) we present the high-level system architecture and define the components responsible for executing the processing steps discussed above.

4.2 System Overview

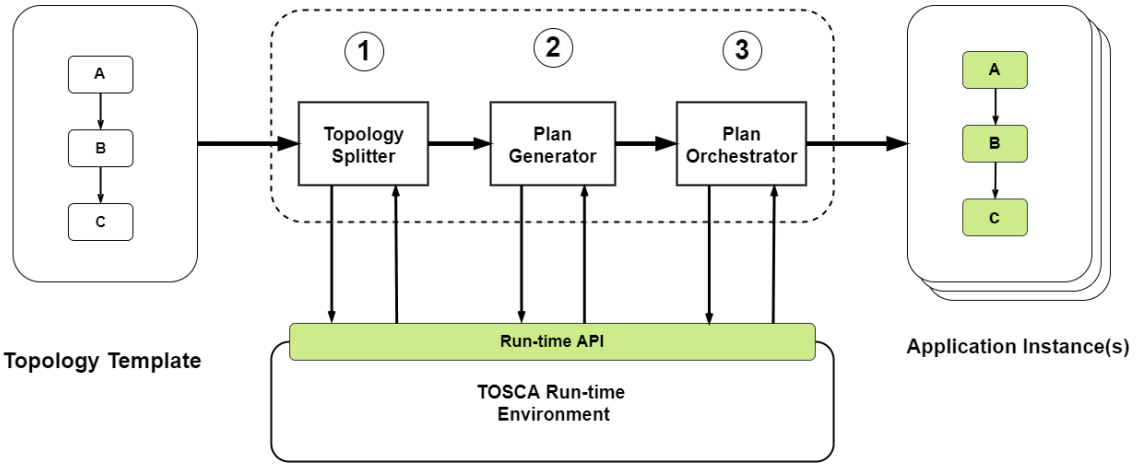

Figure 4.2: Illustration depicting the High Level System Architecture for the concept introduced in Section 4.1

The overall system architecture, as illustrated in Figure 4.2 comprises of three primary components - *(i) Topology Splitter, (ii) Plan Generator, and (iii) Plan Orchestrator*. The

Topology Splitter component takes as input the topology template of a cloud application and generates a set of split definitions. This component abstracts a sequence of well-defined steps, as discussed in Section 4.5.2, that interpret the topology, finds logical groups of dependent components (nodes and relationships) based on well-defined semantics, and finally annotates the components to define the split definitions. The topology splitter can be mapped to step-1 in Figure 4.1. Then, the *Plan Generator* component takes as input the annotated topology generated by the topology splitter and creates a set of executable partial plans. This component abstracts a sequence of well-defined steps, as discussed in Section 4.6.2, that parses, interprets and processes the split definitions defined in the topology to generate the partial plans (step 2, Figure 4.1). Finally, the *Plan Orchestrator* component derives the order of partial plan execution from metadata embedded within the split definitions and orchestrates them to create an instance of the application (step 3, Figure 4.1).

The three components described above interact with an existing TOSCA run-time environment for all the low-level TOSCA specific tasks like storage and processing of topology templates, plans, and associated resources (e.g., IAs and DAs), etc. For this thesis, we will use the OpenTOSCA runtime environment as discussed in Section 2.4.

4.2.1 Prerequisites

In the following section, we list a set of prerequisites in the form of assumptions and requirements that are necessary for the concepts introduced in this chapter to work successfully. These prerequisites are required to formally define the scope of the thesis as outlined in Section 1.3.

Assumptions

Assumption - 1: Only a subset of Node Types are considered It is assumed that only a limited set of Node Types are sufficient to explain concepts introduced in this chapter. The algorithms can further be extended to other Node Types without any modification. The set of used *Node Types* are depicted in Figure 4.4.

Assumption - 2: Only a subset of Relation Types are considered It is assumed that only a limited set of Relationship Types are considered to explain concepts introduced in this chapter.The algorithms can further be extended to other Relation Types without any modification. A collection of Relation Types relevant within the scope of this thesis are depicted in Figure 4.4.

Assumption - 3: Syntactic consistency between Property and Parameter names for this thesis, it is assumed that the names of Plan Input Parameters, Node Properties, and Operation Parameters are syntactically same if they represent the same value. As an example, a simple application topology is depicted in Figure 4.3 that deploys a Docker Container on a pre-configured Docker Engine. It can be seen that the names of the

input parameters of the provisioning plan, the node properties, and the node operation parameters are consistent. The mapping is depicted with dotted arrows. This assumption is a prerequisite to find dependencies between nodes in a topology as a part of the Topology Splitting algorithm discussed in Section 4.5.2.

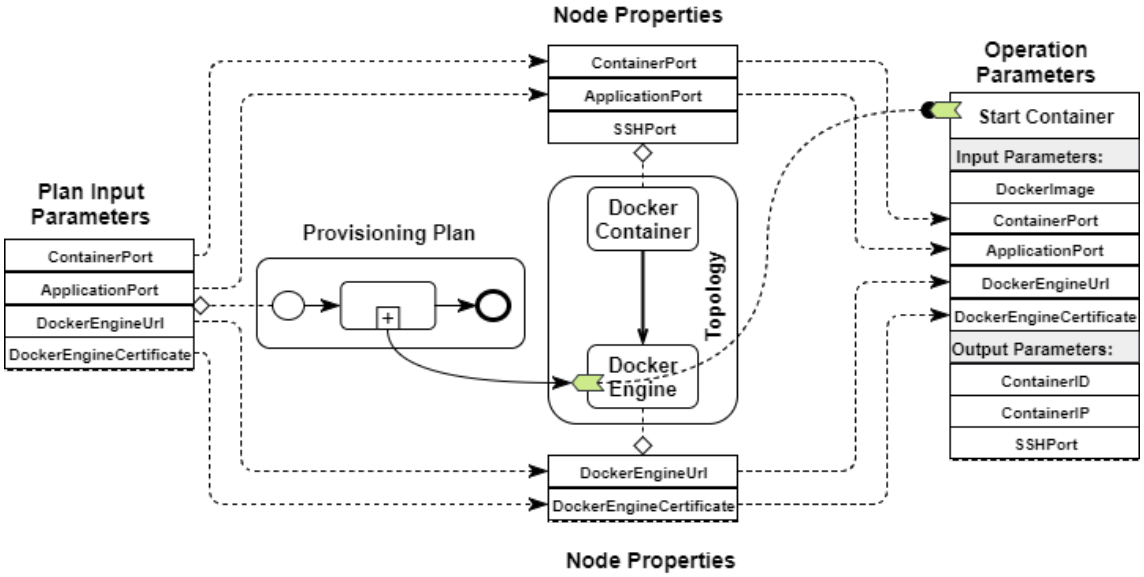

Figure 4.3: Syntactic consistency between property and parameter names for a simple topology that deploys a Docker Container on a pre-configured Docker Engine

Requirements

In their paper titled Merging Models Based on Given Correspondences [PB03] Pottinger and Bernstein define a set of *Generic Merge Requirements (GMRs)*. We will extend and adopt a subset of these requirements similar to [AWMT12] to establish the requirements for the concepts proposed in this thesis.

Requirement - 1: Element Preservation Each element (in this case a Node) in the input (Topology Template), must have at least one corresponding element in the output (Topology Splits). In other words, no node from the input topology is discarded in the splitting process.

Requirement - 2: Relationship Preservation Similar to requirement-1, each relationship between two nodes in the input topology template must have a corresponding relationship between the same nodes in at least one of the topology splits. In other words, no relationship between nodes is discarded in the splitting process.

Requirement - 3: Property Preservation All the nodes in the Split Templates must preserve their properties. In other words, no properties must be discarded, or their semantics altered in the splitting process.

Requirement - 4: Extraneous item prohibition No additional nodes, relationship or parameters must be introduced in the Splitting or Orchestration process that was not part of the original Topology

In the following section, we list a set of formal definitions that will be used to present the algorithms in this chapter.

4.3 Metamodel

In this section, the metamodel introduced by Saatkamp et al. [SBKL17] is extended and reused to define the concepts presented in this thesis. Formal definitions for the various components of an application topology that are relevant in the context of the thesis are given. These formalizations can then mapped to a standard that supports similar semantics for the elements (e.g., TOSCA) and the algorithms can be implemented in any programming language supported by the corresponding run-time engine (e.g., JAVA for OpenTOSCA) .

4.3.1 Definition of Application Templates

- **Definition 4.3.1 (Property)** *A Property is a simple (key, value) pair that can represent different values based on the context. E.g., it can describe the runtime instance value for a Node (Node Property) or a Relation (Relation Property) in a given Topology, or the parameters consumed or generated by an Operation (Operation Input/Output Parameters) or a Plan (Plan Input/Output Parameter) in an Application Template.*

  Let $Prop$ represent the set of all key-value pairs, then $\rho \in Prop$ can be defined by the following tuple:

$$\rho = (key, val) \mid key, val \in \Sigma^{+} \tag{4.1}$$

- **Definition 4.3.2 (Application Template)** *An Application Template is defined as a file that comprises the following components - Topology, Partial Plans, Split Definitions, and an Orchestration Map.*

  Let A represent the set of all application templates, then $a \in A$ can be defined by the following tuple:

$$a = (t_a, P_a, S_a, M_a) \tag{4.2}$$

  Where, t_a, P_a, S_a, M_a represent the topology, partial plans, split definitions, and orchestration map contained in the application template *a* respectively.

  - **Definition 4.3.3 (Topology)** *Topology can be defined as a directed and possibly disconnected graph that comprises of Nodes and Relations. The topology represents the structure of an application where the components are represented by the nodes, and the relationship between the components is depicted by the relations. Each*

Node and Relation within a Topology are typed where the Node Type and Relation Type provide the semantics for the respective nodes and relations.

Let *T* represent the set of all Topologies, then $t_a \in T$ can be defined by the following tuple:

$$t_a = (N_{t_a}, R_{t_a}, NT_{t_a}, RT_{t_a}, L_{t_a}, type_{t_a}, label_{t_a}, source_{Rel}, target_{Rel}, source_{Node}, target_{Node}) \tag{4.3}$$

Where, $N_{t_a}, R_{t_a}, NT_{t_a}, RT_{t_a}$ represent the set of all nodes, relations, node types and relation types defined within the topology t_a respectively.

* **Definition 4.3.4 (Node)** *A node represents a single component of a topology, e.g., Virtual Machine, Web Server, etc. Each node may further comprise of Node Properties, which are key-value pairs and store the runtime instance values for each node. These Node Properties can either be mapped to manually provided parameters or dynamically generated at run time. Nodes can also contain Node Operations which can be executed for provisioning/management of the nodes. Node Operations may belong to a given Node or inherited from the respective Node Type.*

  Let N_{t_a} represent the set of all Nodes in t_a, then *n* $\in N_{t_a}$ can be defined by the following tuple:

$$n = (\Psi_n, \Theta_n) \tag{4.4}$$

  Where,

  - Ψ_n: $\{\rho \mid \rho \in Prop\}$, i.e set of all key-value pairs that define valid *Node Properties* for n
  - Θ_n: $\{\theta \mid \theta \in Ops\}$, i.e. set of all valid *Node Operations* for n and also inherited from the *Node Type* of n.

* **Definition 4.3.5 (Relation)** *A relation represents the relationship between any two nodes of a topology, e.g., a node maybe 'hostedOn' or 'connectesTo' another node. Each relation may further comprise of Relation Properties that are key-value pairs and store the runtime instance values for each relation. Relations can also contain Relation Operations which can be executed for provisioning/management of the relations. Relation Operations may belong to a given Relation or inherited from the corresponding Relation Type.*

  Let $R_{t_a} \subseteq N_{t_a} X N_{t_a}$ represent the set of all Relations in t_a, then *r* $\in R_{t_a}$ can be defined by the following tuple:

$$r = (\Psi_r, \Theta_r) \tag{4.5}$$

  Where,

- Ψ_r: $\{\rho \mid \rho \in Prop\}$, i.e set of all key-value pairs that define valid *Relation Properties* for r
- Θ_r: $\{\theta \mid \theta \in Ops\}$, i.e. set of all valid *Relation Operations* for r and also inherited from the *Relation Type* of r

* **Definition 4.3.6 ($Source_{Node}$)** *'$source_{Node}$' represents the mapping which defines the source node n for each relation r in t_a.*

$$source_{Node} : N_{t_a} \rightarrow R_{t_a}, n_{t_a} \rightarrow r_{t_a}$$

* **Definition 4.3.7 ($Target_{Node}$)** *'$target_{Node}$' represents the mapping which defines the target node n for each relation r in t_a.*

$$target_{Node} : R_{t_a} \rightarrow N_{t_a}, r_{t_a} \rightarrow n_{t_a}$$

* **Definition 4.3.8 ($Source_{Rel}$)** *'$source_{Rel}$' represents the mapping which defines the incoming relation r for each node n for each in t_a.*

$$source_{Rel} : R_{t_a} \rightarrow N_{t_a}, r_{t_a} \rightarrow n_{t_a}$$

* **Definition 4.3.9 ($Target_{Rel}$)** *'$target_{Rel}$' represents the mapping which defines the outgoing relation r for each node n for each in t_a.*

$$target_{Rel} : N_{t_a} \rightarrow R_{t_a}, r_{t_a} \rightarrow n_{t_a}$$

* **Definition 4.3.10 (Node Type)** *A Node Type provides extensibility to nodes. They define the semantics of all the nodes that inherit from the given Node Type.*

  Let NT_{t_a} represent the set of all Node Types in t_a, then $nt \in NT_{t_a}$ defines the semantics of a node that derives from this node type.

* **Definition 4.3.11 (Relation Type)** *A Relation Type provides extensibility to relations. They define the semantics of all the relations that inherit from the given Relation Type.*

  Let RT_{t_a} represent the set of all Relation Types, then $rt \in RT_{t_a}$ defines the semantics of a relation that derives from this relation type.

* **Definition 4.3.12 (Target Label)** *A target label is a string value that specifies the vertical split that a node in the topology is part of.*

  Let L_{t_a} represent the set of all Target Labels in t_a, then $l \in L_{t_a}$ specifies the vertical split a node is part of.

* **Definition 4.3.13 (Type)** *'type' represents the mapping which assigns to each Node and Relation in t_a its Node Type and Relation Type.*

$$type_{t_a} : NT_{t_a} \cup RT_{t_a} \rightarrow N_{t_a} \cup R_{t_a}$$

* **Definition 4.3.14 (Label)** *'label' represents the mapping which assigns each Node in t_a to a Target Label l.*

$$label_{t_a} = \{(n, l) | n \in N_{t_a}, l \in \Sigma^+ \cup \{\perp\}\}$$

* **Definition 4.3.15 (Operation)** *An Operation represents the semantics of an executable function that can be that can be used for provisioning/management of the respective Node (Node Operation) or Relation (Relation Operation).*

  Let Ops represent the set of all Operations, then $\theta \in Ops$ can be defined by the following tuple:

$$\theta = (\Phi_{in_\theta}, \Phi_{out_\theta}) \tag{4.6}$$

  Where,

  · Φ_{in_θ}: $\{\rho \mid \rho \in Prop\}$, i.e., the set of all *Operation Input Parameters* for θ.

  · Φ_{out_θ}: $\{\rho \mid \rho \in Prop\}$, i.e., the set of all *Operation Output Parameters* for θ.

– **Definition 4.3.16 (Partial Plan)** *A Partial Plan can be defined as a set of operations that are executed automatically in a specific order to instantiate a part of the complete application. The partial plans consume a set of Plan Input Parameters and on successful completion generate a set of Plan Output Parameters*

  Let P_a represent the set of all Partial Plans in application template a, then $p \in P_a$ can be defined by the following tuple:

$$p = (\Phi_{in_p}, \Phi_{out_p}) \tag{4.7}$$

  Where,

  * Φ_{in_p}: $\{\rho \mid \rho \in Prop\}$, i.e., the set of all *Plan Input Parameters* for *p*

  * Φ_{out_p}: $\{\rho \mid \rho \in Prop\}$, i.e., the set of all *Plan Output Parameters* for *p*

– **Definition 4.3.17 (Split Definition)** *A Split Definition is semantically equivalent to a 'Scale-Out Region' [KBL17] discussed in Section 3.3.2. Additionally, only 'User Provided Selection Strategy' is used for marking the boundary of a split definition.*

  Let S_a represent the set of all Split Definitions in application template a, then $s \in S_a$ can be defined by the following tuple:

$$s = (N_{split},\ R_{split},\ N_{strat})$$

  Where,

  * $N_{split} \subseteq N_{t_a}$, is the set of all nodes in split that need to be provisioned.
  * $R_{split} \subseteq R_{t_a}$, is the set of all relations in the split that need to be provisioned.
  * $N_{strat} \subset N_{t_a} \mid N_{split} \cap N_{strat} = \emptyset$, is the set of nodes that define the boundary of the split. These nodes are assumed to be provisioned before the other nodes in N_{split}. Additionally, their instance information is used as an input for the partial plan generated for the split definition s, to provision the nodes in N_{split}

– **Definition 4.3.18 (Orchestration Map)** *An Orchestration Map represents a set of tuples that store the mapping information between Nodes, the Split Definition the nodes belong to, and the corresponding Plan generated for the Split Definition.*

  Let M_a represent an Orchestration Map in the application template a, then $m \in M_a$ it can be formally represented by the following expression:

$$m = (n, s, p) \mid n \in N_{t_a},\ s \in S_a,\ p \in P_a \tag{4.8}$$

4.4 Running Example

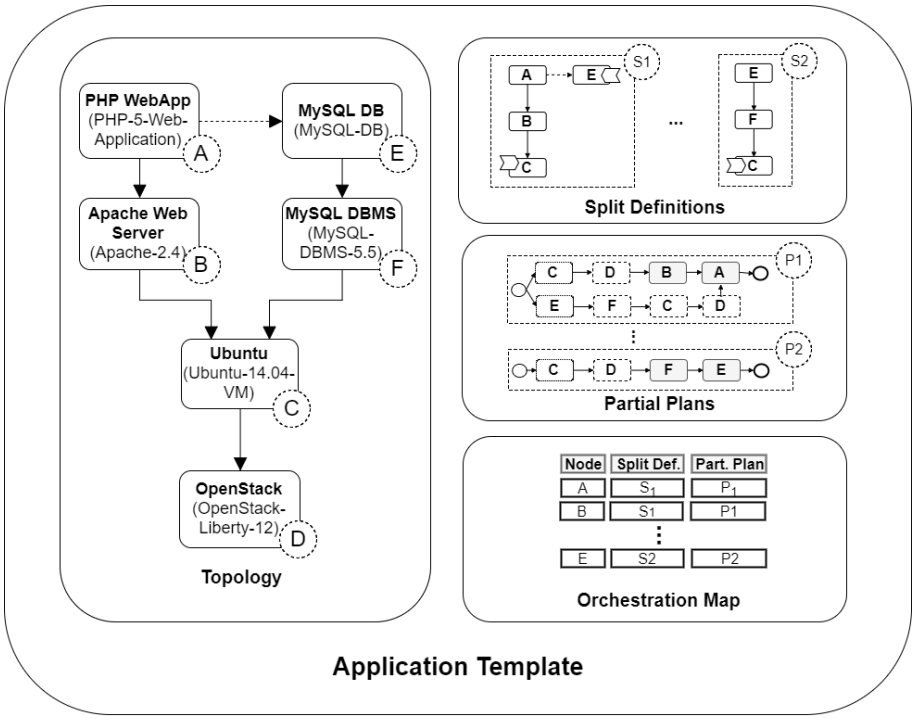

Figure 4.4: Illustration showing an Application Template and its components - Topology, Split Definitions, Partial Plans, and Orchestration Map

A sample *Application Template* is depicted in Figure 4.4 that will be used as a running example for the algorithms discussed in the subsequent sections of this chapter. The *Topology* within the application template depicts a simple stateless PHP Web Application deployed on an Apache Web Server. The application connects to a MySQL Database deployed on a MySQL Database Management System. Finally, all the components are deployed on an Ubuntu Virtual Machine hosted on OpenStack Liberty that forms the infrastructure layer. Additionally, the *Split Definitions (S1,S2, ...,Sn)* and *Partial Plans (P1, P2,...,Pn)*, and the *Orchestration Map* are also depicted.

For clarity, we will use the letters A-F, as shown in the diagram, to represent the nodes of the given topology for the rest of the chapter.

4.5 Topology Splitting

In this section, we will first discuss the high-level steps involved in the topology splitting process (4.5.1). Next, in Section 4.5.2, the detailed algorithm(s) for each of the steps will be presented while adhering to the defined assumptions (4.2.1) and requirements (4.2.1).

4.5.1 Overview

Topology Splitting is the first step in the process of generating granular plans that can be used for dynamic provisioning of the application. The process can be broadly divided into three steps as shown in Figure 4.5.

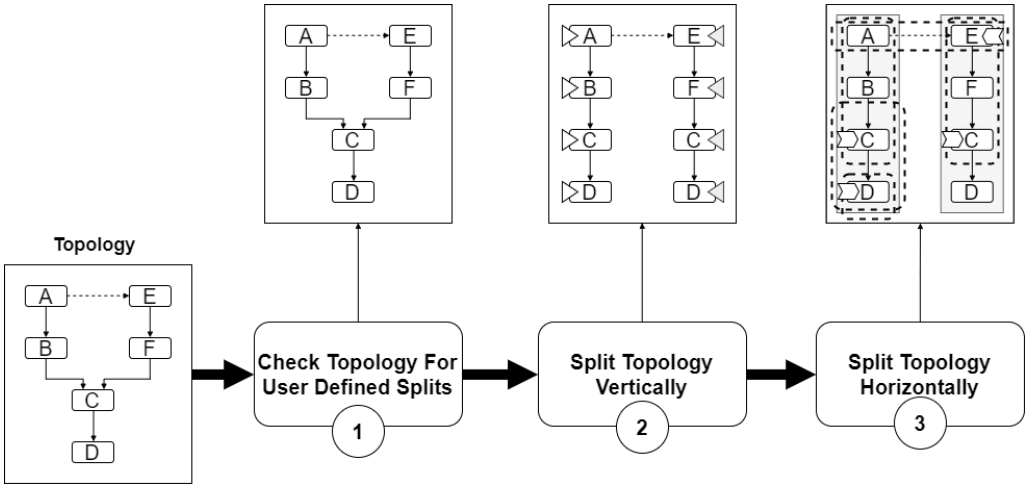

Figure 4.5: Illustration showing high-level steps in the Topology Splitting algorithm.

Check Topology For User Defined Splits In this step, the topology is checked for any existing user-defined splits that were added manually. In case splits already exist, the topology is discarded, and the process exits. Otherwise, the topology is passed to the next step.

Split Topology Vertically In this step, the topology is split vertically using the concepts discussed in Section 3.4.1 [SBKL17]. This step acts as a precursor to the next step in the sense that it generates a topology with clearly defined boundaries that makes it easier for generating horizontal splits. In other words, it generates separate branches for nodes connected via the *connectsTo* relation.

Split Topology Horizontally In this step, the vertically split topology generated in the previous step is used to create *Split Definitions*. As already discussed previously, split definitions represent horizontal splits in the topology and are used by the plan generator component to generate partial plans.

4.5.2 Algorithm

In this section, we take a deeper look into each of the steps mentioned in previous section (4.5.1) and take a look at the corresponding algorithms.

1. Check Topology For User Defined Splits

This step takes an *Application Template* $a \in A$ as an input and returns an empty set $\emptyset$ if the topology has existing splits. If no splits were discovered, then the *Application Template* a is returned as is. A *Topology* t_a in a is said to be split if any of the sets - *Partial Plans, Split Definitions, or Orchestration Map* are non-empty (Line-5, Algorithm 4.1).

Algorithmus 4.1 Check Topology For User Defined Splits

```
1:  procedure CHECKTOPOLOGYFORUSERDEFINEDSPLITS(a ∈ A)
2:      P_a ← π_2(a) //Partial Plans in a
3:      S_a ← π_3(a) //Split Definitions in a
4:      M_a ← π_4(a) //Orchestration Map in a
5:      if (P_a ≠ ∅ ∨ S_a ≠ ∅ ∨ M_a ≠ ∅) then
6:          return ∅
7:      else
8:          return a
9:      end if
10: end procedure
```

2. Split Topology Vertically

This step is based on the *Split Topology* algorithm introduced by Saatkamp et al. [SBKL17] in their paper titled *Topology Splitting and Matching for Multi-Cloud Deployments*. This step takes as input the *Application Template* $a \in A$ that is returned by the previous step and returns a vertically split topology. This is achieved in two stages as shown in Figure 4.6.

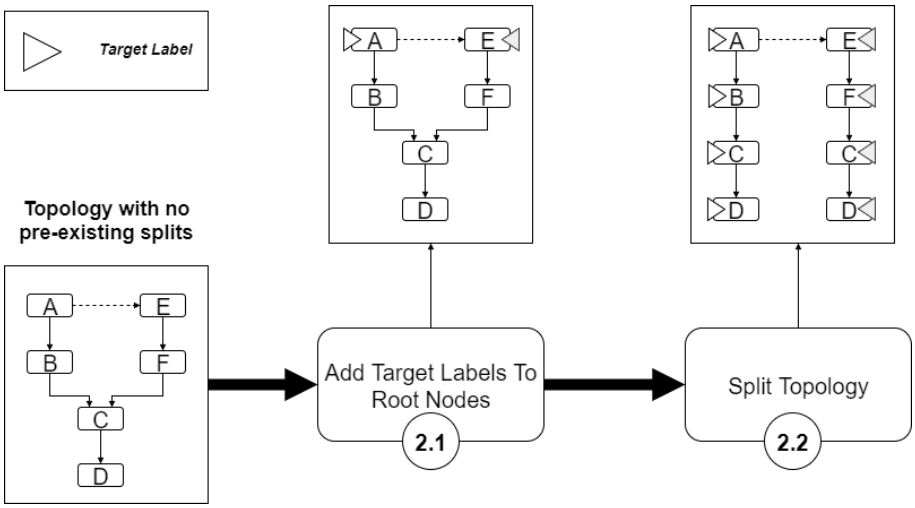

Figure 4.6: Illustration showing the stages involved in the second step of the Topology Splitting algorithm - "Split Topology Vertically"

2.1. Add Target Labels To Root Nodes

This stage is equivalent to the *'Add Target Labels'* step in the *'Split and Match'* method (Saatkamp et al.) discussed in Section 3.4.1. But, unlike the manual approach followed in the paper, we use an automated approach to add labels to nodes (Lines 4-10, Algorithm 4.2). In this approach, first we determine the *Root Nodes* (application-specific nodes) of the topology t_a and then assign a *Target Label* to them. *Root Nodes*, in a topology, are the nodes that do not have any incoming $hostedOn$ relation and *Target Labels* are simply unique string values that represent *vertical splits*.

It should be noted that, just adding *Target Labels* to the root nodes satisfies the conditions for a *valid split* [SBKL17] which is a pre-requisite for the *Split Topology* algorithm.

2.2. Split Topology

This stage is equivalent to the *'Split Topology'* step in the *'Split and Match'* method (Saatkamp et al.) discussed in Section 3.4.1. We re-use the algorithm from the paper [SBKL17] and therefore omit the details in the following discussion.The algorithm is abstracted by the

$Splitting()$ method (Line 12, Algorithm 4.2) that takes the annotated topology t_a from the previous step as input, and generates a vertically split topology t_s.

Algorithmus 4.2 Split Topology Vertically

```
1: procedure SplitTopologyVertically(a ∈ A)
2:     t_a ← π_1(a) //The Topology in a
3:     N_{t_a} ← π_1(t_a) //The set of all Nodes in t_a
4:     //Step-2.1: Add Target Labels To Root Nodes
5:     for all (n_i ∈ N_{t_a}) do
6:         //Check if node n_i is a root node in the topology t_a
7:         if (source_Rel(n_i) = ∅ ∨ ∀r_i ∈ source_Rel(n_i), type(r_i) ∈ RT_{t_a} − {HostedOn})
   then
8:             //Add a unique label to the node n_i that doesn't already exist in label_{t_a}
9:             label_{t_a} ← label_{t_a} ∪ (n_i, l') | l' ∈ Σ^+ − L, L = {π_2(n, l) | (n, l) ∈ label_{t_a}}
10:        end if
11:    end for
12:    //Step-2.2: Split Topology (Saatkamp et al. [SBKL17])
13:    return t_s ← Splitting(t_a)
14: end procedure
```

3. Split Topology Horizontally

This step takes two inputs - (i) the *Application Template* a that is returned in *Step-1: Check Topology For Splits*, and (ii) the *Vertically Split Topology* t_s generated in *Step-2: Split Topology Vertically*. It then iteratively extracts *Vertical Splits* from t_s, find *Node Dependencies* within each vertical split, and generates *Split Definitions* based on the node dependencies. Finally, it removes duplicate split definitions generated, if any, and then returns the updated application template. This is achieved in four stages as shown in Figure 4.6. In the following sections, we describe the individual steps of the process and at the end provide the overall algorithm.

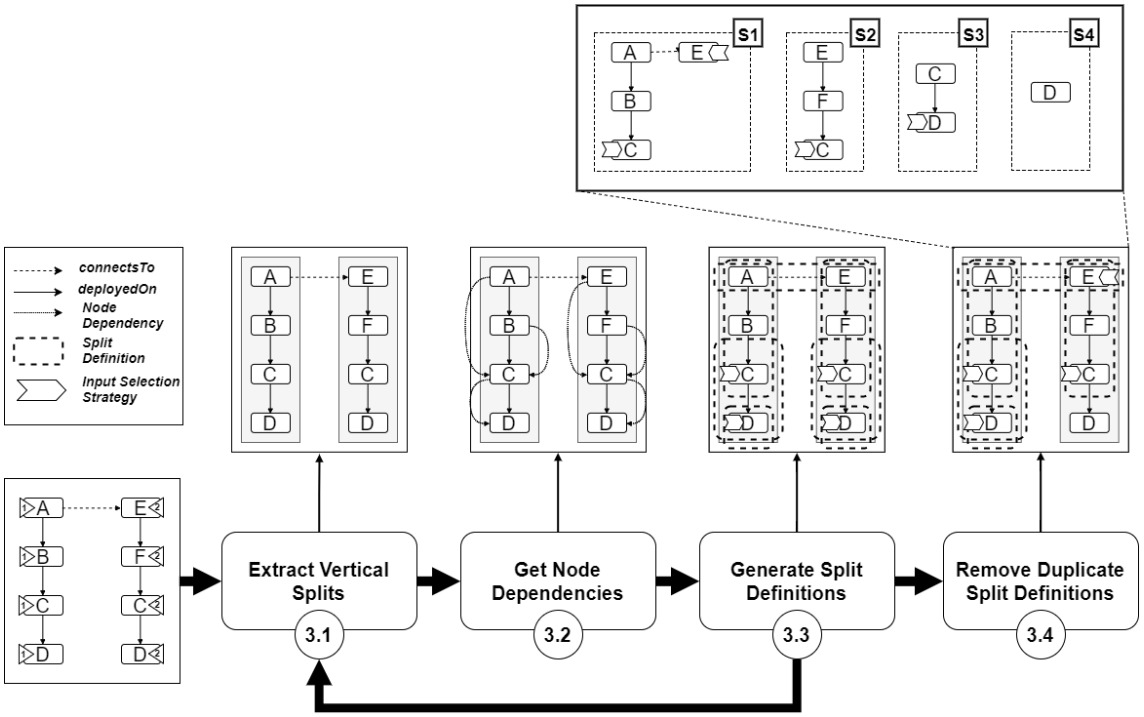

Figure 4.7: Illustration showing the stages involved in the third step of the Topology Splitting algorithm - "Split Topology Horizontally"

3.1. Extract Vertical Split

This stage is represented by the procedure defined in Algorithm 4.3. It takes as input a root node of a vertical split in the topology t_s and returns an ordered set of all nodes in the split. This is achieved by recursively traversing the vertical split by following the outgoing infrastructure edge ($hostedOn$) for each node (Line-6, Algorithm 4.3) and adding them to an ordered set. The algorithm terminates when the leaf node is reached, i.e., a node with no outgoing relations (Line-3, Algorithm 4.3). The ordered set is represented by N_{vert}.

Algorithmus 4.3 Extract Vertical Splits

```
1: procedure ExtractVerticalSplit(n ∈ N_ts) //n is a root node in the vertically split topology
2:     N_vert[0] ← n //Array of nodes in a vertical split (index starts at 0)
3:     i ← 1
4:     while (target_Rel(n) ≠ ∅ ∧ ∀r_i ∈ target_Rel(n), type(r_i) ∈ {HostedOn}) do
5:         n ← target_Node(target_Rel(n))
6:         N_vert[i] ← n
7:         i ← i + 1
8:     end while
9:     return N_vert
10: end procedure
```

3.2. Get Node Dependencies

22nce the ordered list of nodes N_{vert} is obtained, it is iterated recursively (Lines 10-30, Algorithm 4.8) and the nodes are checked for dependencies. The Node dependencies can be of the following two types - *Artifact Dependency and Property Dependency*. Two nodes are said to have an Artifact Dependency when the first node needs to transfer/deploy a specific artifact to/on the second node, e.g., A web application node may deploy a WAR file on a VM node as a part of the provisioning process. Property Dependency, on the other hand, determines if any Node Properties of the first node that are required for its provisioning are dependent on the output parameters of one or more Node Operations of the second node. In other words, if the provisioning of the first node is dependent on the Node Operation(s) of the second node they are said to have a Property Dependency.

The logic for checking *Node Dependencies* is represented by the procedure in Algorithm 4.4. First, artifact dependency is checked (Line-3, Algorithm 4.4) followed by the property dependency (Lines 7-11, Algorithm 4.4). The implementation logic for checking *Artifact Dependency* has been omitted intentionally and can be handled based on the specific implementation used (e.g., OpenTOSCA). On the other hand, the implementation logic for checking *Property Dependency* is represented by the procedure in Algorithm 4.5. If a single match is found between any property of the source node and an output parameter of an operation of the target node (Lines 2-8, Algorithm 4.5), then a boolean $true$ is returned.

Note: Once a node dependency is determined between a source node and a target node, the target node is added to a *Node Dependency List* $N_{dependency}$ (Line 13, Algorithm 4.8).

Algorithmus 4.4 Check Node Dependency

```
1: procedure CHECKNODEDEPENDENCY(n_i, n_j ∈ N_vert)
2:     //Check Artifact Dependency
3:     if (HASARTIFACTDEPENDECY(n_i, n_j)) then
4:         return true
5:     end if
6:     //Check Property Dependency
7:     for all (θ ∈ Θ_{n_j}) do
8:         if (HASPROPERTYDEPENDECY(Ψ_{n_i}, Φ_{out_θ})) then
9:             return true
10:        end if
11:    end for
12: end procedure
```

Algorithmus 4.5 Check Property Dependency

```
1: procedure HASPROPERTYDEPENDENCY(Ψ_n, Φ_{out_θ})
2:     for all (ψ ∈ Ψ_n) do
3:         for all (φ ∈ Φ_{out_θ}) do
4:             if (ψ.key = φ.key) then
5:                 return true
6:             end if
7:         end for
8:     end for
9:     return false
10: end procedure
```

3.3. Generate Split Definitions

The implementation logic for this step is represented in Algorithm 4.6. It takes as input a dependency list $N_{dependency}$ for a node $n \in N_{vert}$ and returns a split definition, s. First, all the nodes (except the leaf node) $n \in N_{dependency}$ are iteratively added to the N_{split}. Then, for each node in $N_{dependency}$ the corresponding outgoing relation R_{out} is added to R_{split}. Finally, the leaf node is added to the N_{strat} component of the split definition s.

Algorithmus 4.6 Generate Split Definition

```
1: procedure GENERATESLITDEFINITION(N_dependency)
2:     N_split ← ∅
3:     R_split ← ∅
4:     N_strat ← ∅
5:     for all (n_i ∈ N_dependency) do
6:         if (target_Rel(n_i) ≠ ∅) then
7:             N_split ← N_split ∪ {n_i}
8:             R_split ← R_split ∪ target_Rel(n_i)
9:         else
10:            N_strat ← N_strat ∪ {n_i}
11:        end if
12:    end for
13:    return s ← (N_split, R_split, N_strat)
14: end procedure
```

3.4. Remove Duplicate Split Definitions

This is simply a cleanup step in the process of topology splitting. Because certain nodes might have been duplicated to generate the vertical splits [SBKL17], there is a possibility of generation of duplicate *Split Definitions*. Two split definitions are said to be equal if they have the same elements in $N_{split}, R_{split},\ and\ N_{strat}$ respectively. Therefore, the duplicate definitions are removed to avoid generation of duplicate partial plans. This step is represented by the *RemoveDuplicateSplitDefinitions()* method (Algorithm 4.7).

Algorithmus 4.7 Remove Duplicate Split Definitions

```
1: procedure REMOVEDUPLICATESLITDEFINITIONS(S_a)
2:     for all (s_i ∈ S_a) do
3:         for all (s_j ∈ S_a − {s_i}) do
4:             if (π1(s_i) = π1(s_j) ∧ π2(s_i) = π2(s_j) ∧ π3(s_i) = π3(s_j)) then
5:                 S_a ← S_a − {s_j}
6:             end if
7:         end for
8:     end for
9:     return S_a
10: end procedure
```

The overall algorithm for splitting the topology horizontally is now presented below,

Algorithmus 4.8 Split Topology Horizontally

```
1: procedure SplitTopologyHorizontally(a, t_s)
2:     S ← ∅
3:     N_{t_s} ← π_1(t_s)
4:     for all (n_k ∈ N_{t_s}) do
5:         N_vert ← ∅ //Array of nodes in a vertical split
6:         //Check if node n_k is a root node in the topology t_s
7:         if (source_Rel(n_i) = ∅ ∨ ∀r_i ∈ source_Rel(n_i), type(r_i) ∈ RT_{t_a} − {HostedOn}) then
8:             N_vert ← ExtractVerticalSplit(n_k)
9:             splitDepth ← |N_vert|
10:            for (i ← 0; i < splitDepth;) do
11:                N_dependency ← ∅
12:                for (j ← i; j < splitDepth; j ← j + 1) do
13:                    N_dependency = N_dependency ∪ N_vert[j]
14:                    if (CheckNodeDependency(N_vert[i], N_vert[j])) then
15:                        break
16:                    end if
17:                end for
18:                //Create Split Definition for hostedOn relation
19:                S_hostedOn ← GenerateSplitDefinition(N_dependency)
20:                S ← S ∪ S_hostedOn
21:                i ← i + |N_dependency|
22:            end for
23:            //Create Split Definition for connectsTo relation
24:            N_dependency = {n_k, target_Node(target_Rel(n_k) | ∀r_i ∈ target_Rel(n_k), type(r_i) ∈ {ConnectsTo}}
25:            S_connectsTo ← GenerateSplitDefinition(N_dependency)
26:            S ← S ∪ S_connectsTo
27:        else
28:            continue
29:        end if
30:    end for
31:    S ← RemoveDuplicateSplitDefinitions(S)
32:    π_3(a) ← S
33:    return a
34: end procedure
```

4.6 Partial Plan Generation

In this section, we first provide an overview of the Plan Generator component (4.6.1) and compare it with the concepts for generating scale-out plans discussed in Chapter 3, Section 3.3.2. Next, in Section 4.6.2, the detailed algorithm for the plan generation step will be presented while adhering to the defined assumptions (4.2.1) and requirements (4.2.1).

4.6.1 Overview

As discussed in Section 4.2, the *Plan Generator* component takes an annotated topology as input and generates a set of executable *partial plans*. Annotated topology, in this context, refers to an application topology for which *Split Definitions* were generated by the *Topology Splitter* component (4.5). These split definitions are parsed, interpreted, and processed by the plan generator component, and a single partial plan is generated per split definition.

The concepts highlighted in Chapter 3, Section 3.3.2 (Képes, Breitenbücher, and Leymann [KBL17]) for scale-out plan generation are reused for the Plan Generator component. As already discussed, the process of scale-out plan generation consists of two main steps - (i) defining Scale-Out Regions for the topology, and (ii) Scale-Out Plan generation. The first step is similar to the *Topology Splitting* discussed in the previous section, where a *Split Definition* can be considered as a special case of a *Scale-Out Region*. In case of a Split Definition, only the *User Provided Selection Strategy* will be used to define the boundary of the scale-out region. Furthermore, the concepts for partial plan generation is the same as the concepts for scale-out plan generation (3.3.2).

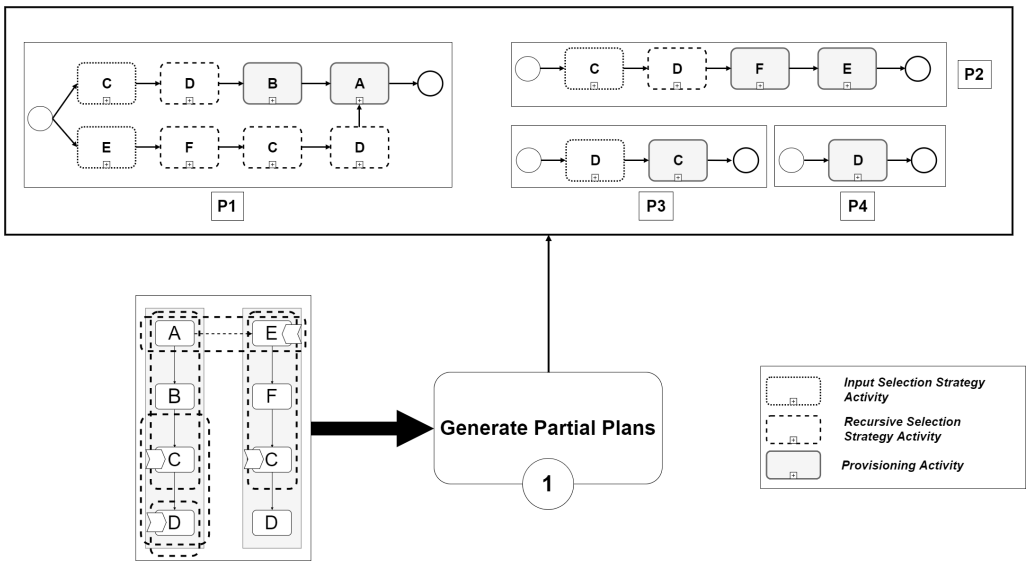

Figure 4.8: Illustration showing the steps involved in Partial Plan Generation from a Split Topology

4.6.2 Algorithm

Algorithm 4.9 provides a high level method for partial plan generation based on the formalizations introduced in Section 4.3.1. The method takes an *Application Template* as input and returns an updated *Application Template* with the generated *Partial Plans*. For each split definition in the application template a partial plan is generated by calling the *GeneratePlan()* method (Line 9, Algorithm 4.9). This method takes a split definition and the application topology as input. The *GeneratePlan()* method abstracts the sale-out plan generation algorithm presented in [KBL17] and therefore the details are omitted for brevity. Finally, the set of all partial plans is added to the application template, and the updated template is returned.

Algorithmus 4.9 Generate Partial Plans

```
1:  procedure GeneratePartialPlans(a)
2:      P ← ∅
3:      S_a ← π_3(a)
4:      for all (s_i ∈ S_a) do
5:          t_a ← π_1(a)
6:          p ← GeneratePlan(s_i, t_a) //According to [KBL17]
7:          P ← P ∪ {p}
8:      end for
9:      π_2(a) ← P
10:     return a
11: end procedure
```

4.7 Partial Plan Orchestration

In this section, we first provide an overview of the Plan Orchestrator component (4.7.1). Next, in Section 4.7.2, the detailed algorithm for the plan orchestration step will be presented while adhering to the defined assumptions (4.2.1) and requirements (4.2.1).

4.7.1 Overview

As discussed in Section 4.2, the *Plan Orchestrator* component derives the order of partial plan execution and orchestrates them to create an instance of the application. This step makes use of an *Orchestration Map* (Figure 4.9) that holds the mapping between the *Nodes*, *Split Definitions* and the corresponding *Partial Plans*. In order to determine the order of partial plan execution we use a modified version of the *Provisioning Order Graph (POG)* generation algorithm discussed in Section 3.3.1 (Breitenbücher et al. [BBK+14]). The orchestration process is carried out in five steps as depicted in Figure 4.10.

Node	Split Definition	Partial Plan
A	S_1	P_1
B	S1	P1
C	S_3	P_3
D	S_4	P_4
E	S_2	P_2
F	S_2	P_2

Orchestration Map

Figure 4.9: Illustration showing an Orchestration Map that contains the mapping between Nodes, Split Definitions, and Partial Plans

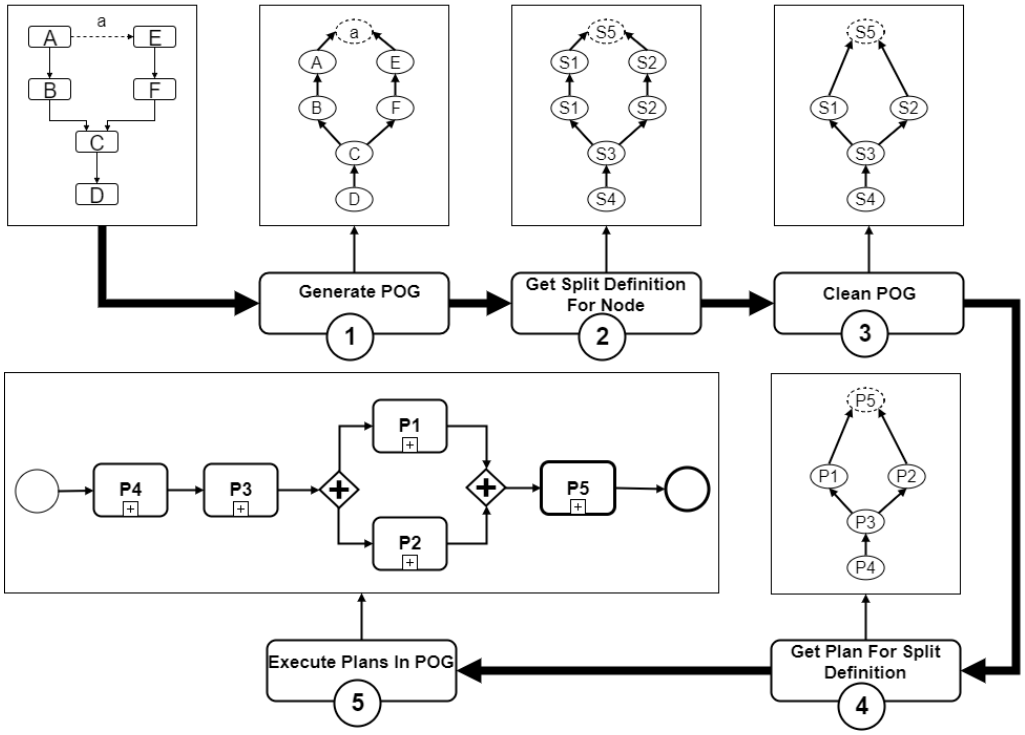

Figure 4.10: Illustration showing the different stages of the Partial Plan Orchestration Algorithm.

4.7.2 Algorithm

Algorithm 4.10 provides the method for partial plan orchestration. It takes as input, the updated *Application Template* obtained from the previous step and generates a workflow for execution of the *Partial Plans* contained in the template.The steps involved in the process (Figure 4.10) are discussed next.

1. Generate POG

In this step, an abstract *Provisioning Order Graph (POG)* [BBK+14] is generated from the given topology t. To generate the *POG*, an abstract activity is inserted into the graph for each *Node* and *Relation* in the *Topology*. Then, the control flow is determined by the semantics of the *Relation*. E.g., In case of a $hostedOn$ relation, the target node is provisioned first and then the source node. Whereas, in case of a $connectsTo$ relation, both the source and target nodes need to be provisioned first before the relation between them can be made. Therefore, a POG represents a partial order of provisioning for the nodes and relations in the topology.

To generate a POG from a given *Topology* we reuse the algorithm proposed by Breitenbücher et al. [BBK+14]. The details of the algorithm are omitted for brevity and are abstracted by the $GeneratePOG()$ method (Line 5, Algorithm 4.10).

Note: The abstract activities for *Relations* have been omitted in the figure for clarity and are handled in the same way as the nodes.

2. Get Split Definition For Node

In this step, the *POG* is traversed recursively and a mapping is created between each vertex and the corresponding *Split Definition* using the *Orchestration Map* (Lines 6-10, Algorithm 4.10). This step is based on the assumption that an Orchestration Map is available and maintained throughout the lifecycle of the topology splitting process.

3. Clean POG

As already discussed in Algorithm 4.8 (Line 23), a split definition may contain multiple nodes based on the elements in the *Node Dependency List,* $N_{dependency}$. Therefore, multiple nodes maybe mapped to the same split definition in the orchestration map. As a result of which, multiple consecutive edges in the *POG* may also be mapped to the same split definition (2, Figure 4.10). Therefore, in this step, duplicate mappings are removed from the *POG*. This is achieved by recursively iterating the POG from the source towards the sink, and keeping only the first vertex from the set of duplicate vertices intact while deleting the rest (3, Figure 4.10). This step is abstracted by the $CleanPOG()$ method (Line 11, Algorithm 4.10).

4. Get Plan For Split Definition

In this step, the *clean POG* is traversed, and a mapping is created between each vertex and the corresponding *Partial Plan* using the *Orchestrtaion Map* (Lines 13-17, Algorithm 4.10).

5. Execute Plans In POG

In this step, the *POG* is traversed from source to sink, and each mapped partial plan is executed. This can either be achieved programmatically through code written in any programming language, e.g., JAVA or automatically using workflows. In each step of the execution a *Partial Plan* is executed, and the output of the execution (which includes the instance id of the provisioned elements) is passed as input parameters to the next *Partial Plan*. These steps are abstracted by the $ExecutePlansInPOG()$ method (Line 12, Algorithm 4.10).

Algorithmus 4.10 Orchestrate Partial Plans

```
 1: procedure ORCHESTRATEPARTIALPLANS(a)
 2:     t_a ← π_1(a)
 3:     M_a ← π_4(a)
 4:     //Step-1: Generate POG
 5:     G_POG ← GENERATEPOG(t_a) //Breitenbücher et al. [BBK+14]
 6:     map_{η→s} ← {(η, s) | η ∈ G_POG, s ∈ S_a}
 7:     map_{η→p} ← {(η, p) | η ∈ G_POG, p ∈ P_a}
 8:     for all (η_i ∈ G_POG) do
 9:         //Step-2: Get Split Definition For Node
10:         s ← {s' = π_2(m) | m ∈ M_a, π_1(m) = η_i}
11:         //Create a mapping between the vertex and the split definition
12:         map_{η→s} ← map_{η→s} ∪ (η_i, s)
13:     end for
14:     //Step-3: Clean POG
15:     CLEANPOG(G_POG)
16:     for all (λ ∈ map_{η→s}) do
17:         //Step-4: Get Plan For Split Definition
18:         p ← {p' = π_3(m) | m ∈ M_a, π_2(m) = π_2(λ)}
19:         //Create a mapping between the vertex and the partial plan
20:         map_{η→p} ← map_{η→p} ∪ (π_1(λ), p)
21:     end for
22:     //Step-5: Execute Plans in POG
23:     EXECUTEPLANSINPOG(G_POG)
24: end procedure
```

5 Design and Implementation

5.1 Design Overview

In this section, we discuss the prototype that was used to test the feasibility of the concepts introduced in Chapter 3 and provide the low-level architecture for its implementation.

5.1.1 Prototype

The prototype was developed using the OpenTOSCA framework discussed in Section 2.4. The *Topology Splitter and Plan Orchestrator* components were developed and integrated with the existing framework, while the *Plan Generator* component of the OpenTOSCA container was reused as is for partial plan generation. A simple application topology was used (Figure 5.1) for the POC. The topology was designed with the Winery Topology Modeler, and then packaged and exported into a CSAR using the functionality provided by the tool. The CSAR was then imported into the OpenTOSCA container with the help of the OpenTOSCA UI web interface.

Once the CSAR is successfully imported, the *Topology Splitter* component gets a reference of the *Service Template* contained in the CSAR and processes the topology embedded in it to generate the *Split Definitions*. The semantics of *Scale-Out Region* [KBL17] are reused to embed the split definitions within the Service Template (see Section 5.2.2). Once the split definitions are generated, the Service Template is repackaged into the CSAR and passed to the Plan Generator component.

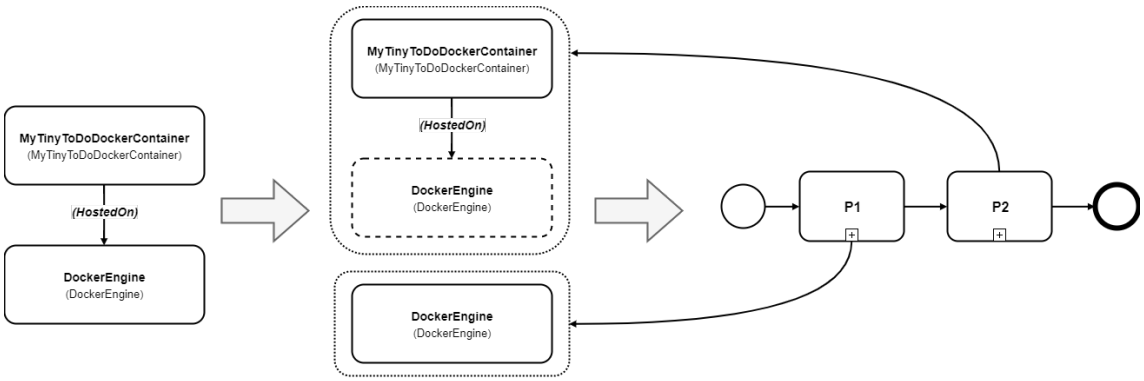

Figure 5.1: Illustration showing the sample application topology considered for the prototype

The Plan Generator component then imports the CSAR, parses the Service Template for the Split Definitions (scale-out regions), generates BPEL based partial plans (scale-out plans) [KBL17] based on the Split Definitions, and adds them to the CSAR. The CSAR is then repackaged and exported back into the OpenTOSCA container. The container is then responsible for deploying the plans into the compatible plan engine (WSO2 BPS) integrated within the runtime environment.

The Plan Orchestrator component that is developed external to the OpenTOSCA container using the OpenTOSCA client API is the used to get a reference to the deployed application. The metadata related to the partial plans is obtained, and their execution order is derived. For the POC, the execution order of the plans was embedded within the plan names, e.g., Plan01, Plan02, etc., i.e., Plan01 is executed before Plan02 and so on. Once the execution order is obtained, the plans are executed iteratively using the client API. At each step of the iteration, the output of the execution of a plan is provided as input to the next partial plan. This is done until all the partial plans are executed, and an instance of the application is created, thereby proving the possibility of the creation of an application instance by orchestrating smaller partial plans instead of a single monolithic plan.

5.1.2 Architecture

The low-level architecture for the implementation of the prototype is depicted in Figure 5.2 and the three components of the prototype - *Topology Splitter, Partial Plan Generator, and Partial Plan Orchestrator* are highlighted. It can be seen that the architecture is divided into two parts - (i) Internal, and (ii) External. Internal, in this case, refers to the OpenTOSCA container where all the components directly interact with the OpenTOSCA core. On the other hand, External components interact with the OpenTOSCA Core via the OpenTOSCA Client API that uses a set of well-defined REST endpoints exposed by OpenTOSCA container for communication. For the prototype, the *Topology Splitter, and Partial Plan Generator* are developed as internal components, whereas the *Plan Orchestrator* is built as an external component.

Internal Component

The architecture of the OpenTOSCA Plan Generator [KEP13] was reused for the design of the Internal component of the prototype. The OpenTOSCA Plan Generator is implemented as an OSGi bundle that can be plugged into the OpenTOSCA container. The relevant components of the architecture are discussed briefly in the following section.

Topology Splitter The Topology Splitter component interacts with the OpenTOSCA core via the Integration layer to import/export CSAR files. It further, consumes the components of the CSAR using a CSAR Model via the Facade.

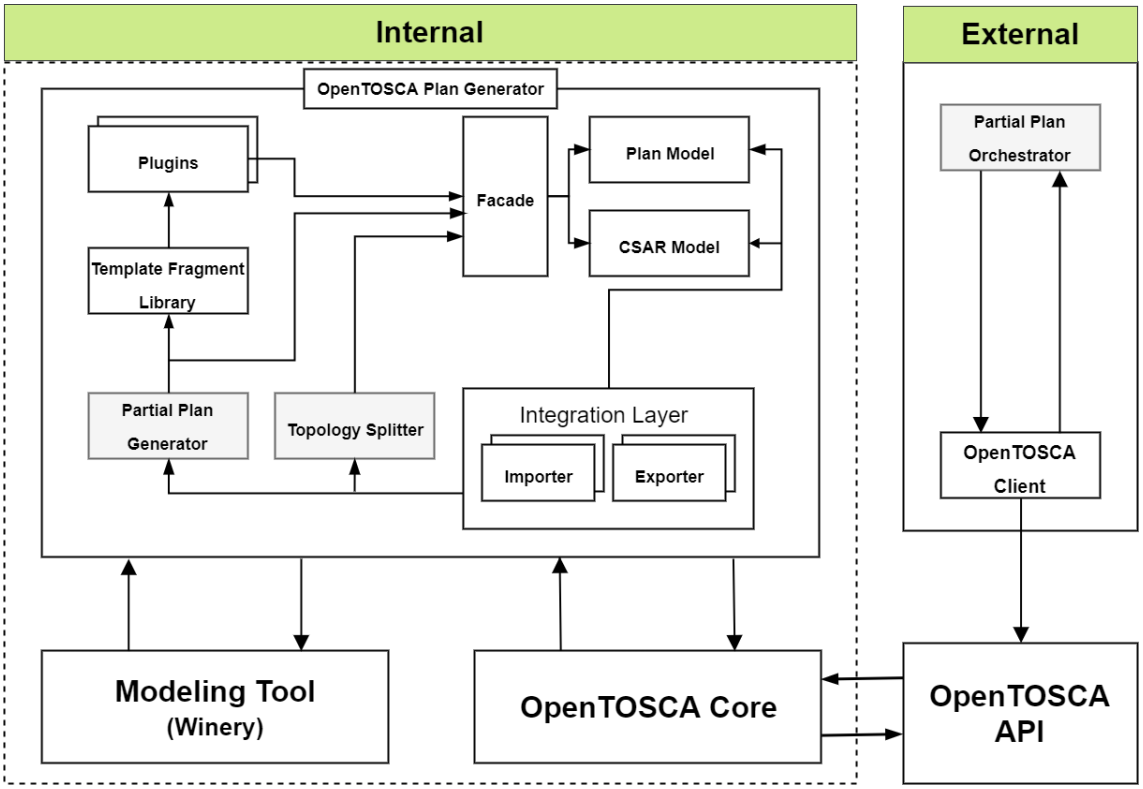

Figure 5.2: Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17]

Partial Plan Generator The Partial Plan Generator component is the same as the *Plan Builder* component [KEP13]. This component implements the provisioning plan generation algorithm described in Breitenbücher et al. [BBK+14] and the *scale-out plan* generation algorithm in [KBL17].

Integration Layer The integration layer [KEP13] is used to make the OpenTOSCA Plan Generator component generic and pluggable. The *Importer and Exporter* components of the Integration layer can be used to provide the logic for integrating the plan generator with different implementations of TOSCA runtime.

Facade The Facade [KEP13] encapsulates the low-level operations on the various models used in the implementation. E.g. The CSAR model replicates a CSAR file and provides access to its components like the Service Templates, Node Types, Node Templates, etc.

External Component

The OpenTOSCA Client [] is reused to design the external component of the prototype. The client is developed in JAVA and provides an abstraction for the low-level REST APIs exposed by the OpenTOSCA container.

Partial Plan Orchestrator The Partial Plan Orchestrator uses the API exposed by the OpenTOSCA Client to access an application and all its components (e.g., Service Template, Plans, etc.) deployed within the OpenTOSCA container. It then orchestrates and executes the partial plans via the APIs.

OpenTOSCA Client The OpenTOSCA Client abstracts the low-level REST APIs exposed by the OpenTOSCA container by providing a strongly typed data model and a JSON API Client. It comprises of models for different components, e.g., the Application (CSAR), Service Instance, Node Instance and Relationship Instance. It also provides APIs to execute plans embedded in the Service Template that is used by the Partial Plan Orchestrator component.

5.2 Implementation

In the following section, the low-level implementation of the prototype is discussed. As already highlighted previously, the OpenTOSCA container is used as the preferred TOSCA runtime environment for the development of the prototype. In Section 5.2.1 we first highlight the abstract data model used to access the various TOSCA elements programmatically followed by the implementation details of the *Topology Splitter* component in Section 5.2.2. In Section 5.2.3 we briefly discuss the relevant parts of the OpenTOSCA Plan Generator (*Partial Plan Generator*) component in the context of the POC, and finally take a look at the OpenTOSCA Client API and its usage in developing the *Partial Plan Orchestrator* in Section 5.2.4.

5.2.1 TOSCA Data Model

Figure 5.3 represents the UML diagram for the CSAR model and the Topology model used for the prototype. The *AbstractDefinition* class is the parent class for all the definition files contained within a CSAR that are imported by the *Importer* component of the *Integration Layer* (5.1.2). The choice of such an abstract model instead of using JAXB was deliberate to enable the importer to adapt it to different environments [KEP13]. Additionally, such an abstract model allows for easy traversal through the topology that is essential to identifying the *Infrastructure Nodes* and *Infrastructure Edges* and thereby defining the *Split Definitions*. It must be noted that only a part of the data model is shown in the figure as the complete TOSCA data model is out of the scope of this implementation.

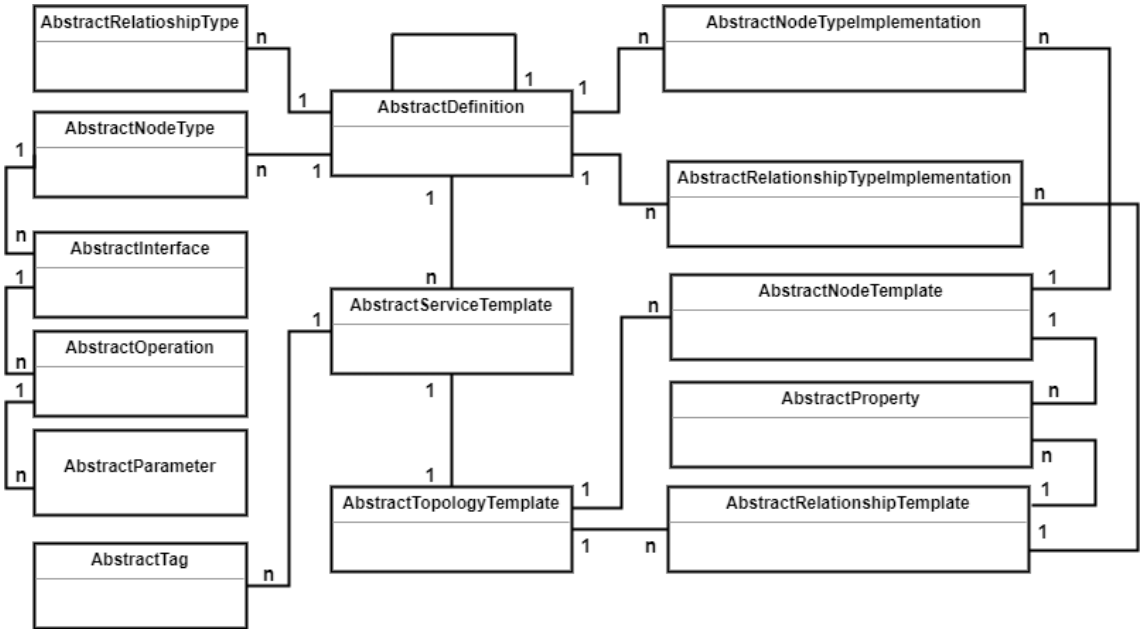

Figure 5.3: Illustration showing the UML class diagram for the CSAR model and Topology model used in the prototype implementation (based on the data model in [KEP13])

5.2.2 Topology Splitter

As already discussed in Section 5.1.2, the Topology Splitter is implemented as an *Internal Component* of OpenTOSCA and is a part of the Plan Generator component. The Topology Splitter directly interacts with the Integration Layer to import the CSAR file from the runtime environment and then uses the CSAR model to access the elements within it. Figure 5.4 represents the UML class diagram for the Topology Splitter. The *Abstract Topology Splitter* supports extensibility by providing the option for different implementations of the class. Furthermore, the *TopologySplitDefinition* class models a *scale-out region* [KBL17]. As already discussed, we extend the concept of scale-out regions to implement Split Definitions. Listing 5.1 shows the Split Definitions for the sample application shown in Figure 5.1 implemented using the *Tag* element of a TOSCA Service Template similar to scale-out regions in [KBL17]. The set of all split definitions is defined by the key-value pair in Line 3, Listing 5.1 where they are listed as comma separated values. Furthermore, each Split Definition is also represented by a key-value pair (Lines 5-6, Listing 5.1) where the *key/name* represents the name/id of the Split Definition, and the *value* represents the Nodes and Relations that are part of the split, and also the Nodes that are to be strategically selected at runtime by using the *User Provided Selection Strategy* [KBL17].

```
1  <tosca:ServiceTemplate name="MyTinyToDo_Bare_Docker" id="MyTinyToDo_Bare_Docker">
2        <tosca:Tags>
3           <tosca:Tag name="splitdefinitions" value="split01,split02"/>
4           <tosca:Tag name="split01" value="DockerEngine;na;na"/>
5           <tosca:Tag name="split02"
                value="MyTinyToDoDockerContainer;HostedOn;UserProvided[DockerEngine]"/>
6        </tosca:Tags>
```

Listing 5.1: Snippet of a TOSCA Service Template with marked regions defining the Split Definitions for the prototype

Finally, the $SplitTopology()$ method of the *TopologySplitter* class, takes the name of the CSAR file and its Root Definition as input and generates a Service Template with the Split Definitions embedded in the *Tags* element. The Service Template is then repackaged into the CSAR and exported back to the runtime environment.

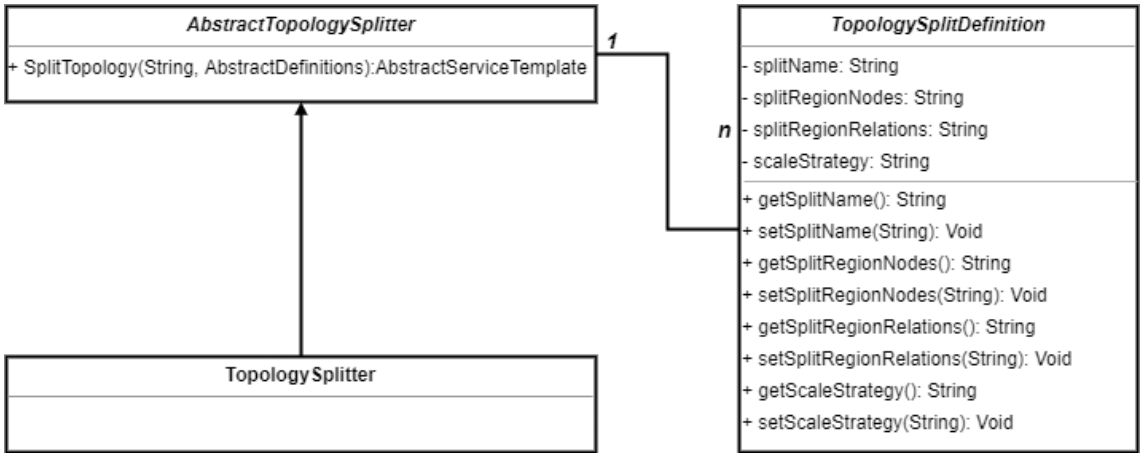

Figure 5.4: Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17]

5.2.3 Partial Plan Generator

As already highlighted previously, we reuse the OpenTOSCA Plan Generator [KEP13] component in this prototype. The Plan Generator component is capable of generating *Partial Plans* by parsing the *Split Definitions* embedded in the Service Template similar to *scale-out plans* from *scale-out regions* [KBL17]. The implementation details are therefore omitted for brevity.

Listing 5.2 shows the snippet of the Service Template with details of the Plans Generated by the Partial Plan Generator. It can be seen that one Partial Plan is created for each Split Definition. Furthermore, each Partial Plan has an *Output Parameter* that maps to the *InstanceID* of the element(Node/Relation) provisioned by it (Lines 13 and 30, Listing 5.2). Also, as a result of using *User Provided Selection Strategy* for the Docker Engine Node in '*split02*'

(Line 5, Listing 5.1), the Partial Plan $'MyTinyToDo\_Bare\_Docker\_PartialPlan\_split02'$ corresponding to the Split Definition expects the InstanceID of the Docker Engine Node as an input (Line 25, Listing 5.2). This feature enables the orchestration of the partial plans.

```
1  <Plans>
2        <Plan id="MyTinyToDo_Bare_Docker_PartialPlan_split01">
3              <InputParameters>
4                    <InputParameter name="instanceDataAPIUrl" type="String" required="yes"/>
5                    <InputParameter name="OpenTOSCAContainerAPIServiceInstanceID"
                         type="String" required="yes"/>
6                    <InputParameter name="CorrelationID" type="String" required="yes"/>
7                    <InputParameter name="DockerEngineURL" type="String" required="yes"/>
8                    <InputParameter name="DockerEngineCertificate" type="String"
                         required="yes"/>
9                    <InputParameter name="ProvisionedInstance_DockerEngine" type="String"
                         required="yes"/>
10             </InputParameters>
11             <OutputParameters>
12                   <OutputParameter name="CorrelationID" type="String" required="yes"/>
13                   <OutputParameter name="DockerEngine_InstanceID" type="String"
                         required="yes"/>
14             </OutputParameters>
15             <PlanModelReference reference=
                    "Plans/MyTinyToDo_Bare_Docker_PartialPlan_split_scaleout_dockerengine.zip"
                    />
16       </Plan>
17       <Plan id="MyTinyToDo_Bare_Docker_PartialPlan_split02">
18             <InputParameters>
19                   <InputParameter name="instanceDataAPIUrl" type="String" required="yes"/>
20                   <InputParameter name="OpenTOSCAContainerAPIServiceInstanceID"
                         type="String" required="yes"/>
21                   <InputParameter name="CorrelationID" type="String" required="yes"/>
22                   <InputParameter name="ApplicationPort" type="String" required="yes"/>
23                   <InputParameter name="ContainerSSHPort" type="String" required="yes"/>
24                   <InputParameter name="csarEntrypoint" type="String" required="yes"/>
25                   <InputParameter name="DockerEngine_InstanceID" type="String"
                         required="yes"/>
26                   <InputParameter name="ProvisionedInstance_MyTinyToDoDockerContainer"
                         type="String" required="yes"/>
27             </InputParameters>
28             <OutputParameters>
29                   <OutputParameter name="CorrelationID" type="String" required="yes"/>
30                   <OutputParameter name="DockerContainer_InstanceID" type="String"
                         required="yes"/>
31             </OutputParameters>
32             <PlanModelReference reference=
                    "Plans/MyTinyToDo_Bare_Docker_PartialPlan_split_scaleout_dockercontainer.zip"
                    />
33       </Plan>
34 </Plans>
```

Listing 5.2: Snippet of a TOSCA Service Template with the Partial Plans generated for the Split Definitions in Listing 5.1

5.2.4 Partial Plan Orchestrator

As already discussed in Section 5.1.2, the Partial Plan Orchestrator is implemented as an *External Component* of OpenTOSCA and uses the *OpenTOSCAContainerAPIClient* to interact with the OpenTOSCA Container. Figure 5.5 shows the Data Model used by the client. The *Application* class represents a successfully Deployed CSAR in the OpenTOSCA runtime environment. Whereas, the *ServiceInstance* class represents an instance of the Application comprising of instances of all the Nodes and Relationships within the Topology of the Application.

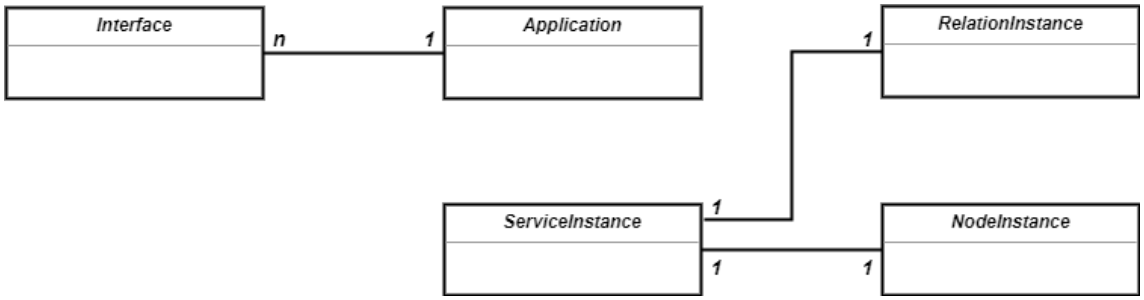

Figure 5.5: Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17]

Further, the UML class diagram of the Partial Plan Orchestrator is represented by Figure 5.6. The *Abstract Partial Plan Orchestrator* supports extensibility by providing the option for different implementations of the class. The figure also lists some of the methods exposed by the OpenTOSCA client that is used by the Partial Plan Orchestrator.

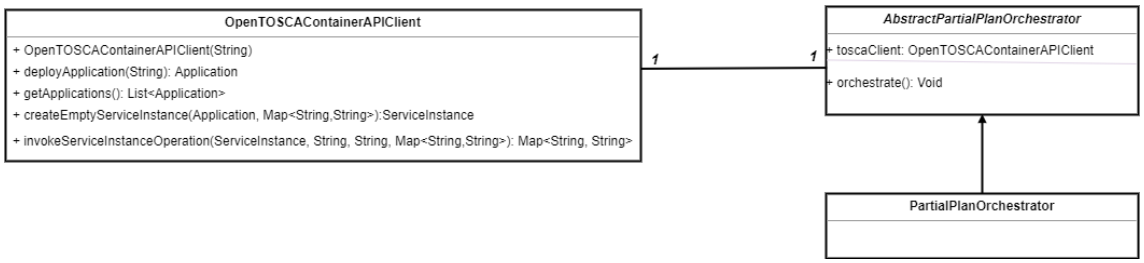

Figure 5.6: Illustration showing the Low-Level Architecture for the concepts based on OpenTOSCA as discussed in [KEP13][KBL17]

The Orchestrator gets an instance of the *Application* deployed in the OpenTOSCA Container along with the generated Partial Plans. Then, it creates an *Empty Service Template* for the Application. An empty service template represents merely a container with no instantiated Nodes or Relationship, in other words, it is a placeholder for the instance of the application. Then, the Partial Plans for the Service Instance are executed iteratively in the order defined by the integer values embedded in their names, e.g., $MyTinyToDo\_Bare\_Docker\_PartialPlan\_split01$ is executed first, followed by

$MyTinyToDo\_Bare\_Docker\_PartialPlan\_split02$. The input parameters provided for provisioning the application instance are mapped to the Input Parameters of both the plans. Upon successful execution of the first plan, its Output Parameters are further assigned to the Input Parameters of the next Plan and so on. Upon successful execution of both the plans, we get an instance of the *MyTinyToDo* application.

6 Conclusion and Future Work

In the course of this master thesis, a concept for dynamic provisioning for cloud applications based on the TOSCA standard was developed. The primary focus was on declaratively generating granular provisioning plans from the topology template of the application, and orchestrating these plans to create instances of the complete application. Concepts from state of the art research were built upon to support and develop the proposed concepts in this thesis. Moreover, the feasibility of the concepts introduced was shown with the help of a prototype designed using the OpenTOSCA ecosystem.

Summary

An extensive study of Related Work was introduced in Chapter-3, where several generic approaches to application provisioning were discussed, and their drawbacks regarding generation of reusable granular plans were highlighted. Furthermore, research work related to Topology Splitting [SBKL17] and Scale-Out Plan Generation [KBL17] were also discussed that laid the foundation for the contributions presented in this thesis.

The first contribution of this thesis is the development of the concepts for *Topology Splitting* which is a pre-requisite for generating granular provisioning plans. Different splitting techniques were highlighted, and the concepts were presented with the help of a sample application topology. The splits in the topology were defined by *Split Definitions* which are a collection of nodes and relations that are dependent on each other based on specific semantics. These semantics were also highlighted, and a detailed algorithm was presented for the approach. Next, the application topology annotated with *Split Definitions* was used by the *Partial Plan Generator* to generate a set of granular plans. The Partial Plan Generator reused the concepts introduced by Képes, Breitenbücher, and Leymann [KBL17] for scale-out plan generation to create the partial plans.

The second contribution of this thesis is the development of the concepts for *Partial Plan Orchestration* that provisions the cloud application. As opposed to the traditional approach of generating and executing a single provisioning plan, the concept of dynamic provisioning dealt with generating multiple smaller plans. The challenge was to be able to execute these plans in the correct order to provide the same output. Therefore, the concept of an *Orchestration Map* was introduced to address this challenge. The map consists of the correlation between the *Elements* in the topology, the *Split Definitions*, and the corresponding *Partial Plans* that provision the elements defined in the split definitions. Next, the order of execution of the partial plans was determined by generating a POG (Breitenbücher et al.

[BBK+14]) from the application topology and mapping the partial plans to the abstract activities of the POG using the Orchestration Map. Then, the application was provisioned by simply executing the POG.

Finally, a prototype was developed based on the OpenTOSCA framework to test the feasibility of the concepts. The *Topology Splitter* component was built as a part of the OpenTOSCA container that reused the well-defined semantics, data models, and rich APIs of the OpenTOSCA Core. For the *Partial Plan Generator* component, the OpenTOSCA Plan Builder which is also a part of the OpenTOSCA container was reused as is. Finally, the *Partial Plan Orchestrator* was developed as an external component using the OpenTOSCA client that interacts with the OpenTOSCA Core with the help of a set of well-defined REST APIs.

Future Work

Future work includes the complete automation of the *Topology Splitting* and *Partial Plan Orchestration* processes and their seamless integration with a TOSCA run-time engine like OpenTOSCA. This would also include developing the aforementioned components as pluggable and reusable entities that can be easily extended to any TOSCA compliant runtime environment. Secondly, the possibility of increasing the granularity of the topology splits needs to be researched further. Based on the current approach, the size of the splits is dependent on the infrastructure dependencies of the nodes in the topology. Increasing the granularity to a single node would result in smaller and more generic partial plans. Thirdly, mechanisms to reuse the same set of partial plans for provisioning different applications need to be defined. Another future work includes extending the provisioning process to an IoT scenario where the smaller partial plans are run on thin clients such as mobile and other smart devices.

Bibliography

[AWMT12] A. Weiß. *Merging of TOSCA Cloud Topology Templates*. Oct. 2012 (cit. on p. 52).

[BBH+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA – A Runtime for TOSCA-based Cloud Applications." In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (cit. on pp. 25, 30, 31).

[BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. *Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 130–148. ISBN: 978-3-642-41030-7. DOI: 10.1007/978-3-642-41030-7_9. URL: https://doi.org/10.1007/978-3-642-41030-7_9 (cit. on pp. 13, 14, 42).

[BBK+14] U. Breitenbücher, T. Binz, K. Képes, O. Kopp, F. Leymann, J. Wettinger. "Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA." In: *Proceedings of the IEEE International Conference on Cloud Engineering (IEEE IC2E 2014)*. IEEE Computer Society, Mar. 2014, pp. 87–96. DOI: DOI10.1109/IC2E.2014.56 (cit. on pp. 15, 16, 42, 43, 69, 71, 72, 75, 83).

[BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. "Pattern-based Runtime Management of Composite Cloud Applications." In: *Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013*. SciTePress, 2013 (cit. on pp. 40, 42).

[BPEP07] O. for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language (WS-BPEL) Version 2.0 Primer*. 2007 (cit. on pp. 25, 31).

[BPES07] O. for the Advancement of Structured Information Standards (OASIS). *Web Services Business Process Execution Language Version 2.0 OASIS Standard*. 2007 (cit. on p. 32).

[BPS+08] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau. "Extensible Markup Language (XML) 1.0 (Fifth Edition)." In: (Nov. 2008) (cit. on p. 14).

[EEKS11] T. Eilam, M. Elder, A. V. Konstantinou, E. Snible. "Pattern-based Composite Application Deployment." In: *12th IFIP/IEEE International Symposium on Integrated Network Management 2011* (Aug. 2011). DOI: 10.1109/INM.2011.5990694 (cit. on pp. 37, 38, 42).

[KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery – Modeling Tool for TOSCA-based Cloud Applications.” In: *11th International Conference on Service-Oriented Computing*. LNCS. Springer, 2013 (cit. on p. 28).

[KBL17] K. Képes, U. Breitenbücher, F. Leymann. “Integrating IoT Devices based on Automatically Generated Scale-Out Plans.” In: *Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications (SOCA), 2017*. IEEE, 2017 (cit. on pp. 44, 45, 57, 68, 69, 73–75, 77, 78, 80, 83).

[KEP13] K. Képes. *Konzept und Implementierung einer Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA*. July 2013 (cit. on pp. 74–78, 80).

[LSS+13] H. Lu, M. Shtern, B. Simmons, M. Smit, M. Litoiu. “Pattern-Based Deployment Service for Next Generation Clouds.” In: *2013 IEEE Ninth World Congress on Services* 32.1 (Nov. 2013), pp. 464–471. DOI: 10.1109/SERVICES.2013.54 (cit. on p. 39).

[MG11] P. Mell, T. Grance. “The NIST Definition of Cloud Computing.” In: (Nov. 2011) (cit. on pp. 13, 20–23).

[OMG11] O. M. G. (OMG). *Business Process Model and Notation (BPMN) Version 2.0*. 2011 (cit. on pp. 14, 25).

[OMG14] *Model Driven Architecture (MDA)*. 2014. URL: http://www.omg.org/mda/specs.htm (cit. on p. 20).

[OSG08] O. Alliance. *OSGi Service Platform Core Specification*. 2008. URL: https://www.osgi.org/developer/specifications/ (cit. on p. 29).

[OST16] J. Opara-Martins, R. Sahandi, F. Tian. “Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective.” In: *Journal of Cloud Computing* 5.1 (Apr. 2016), p. 4. ISSN: 2192-113X. DOI: 10.1186/s13677-016-0054-z. URL: https://doi.org/10.1186/s13677-016-0054-z (cit. on p. 14).

[OTOSCa] T. Binz, U. B. O. Kopp, F. Leymann. *OpenTOSCA Open Source Ecosystem for OASIS TOSCA*. URL: http://www.opentosca.org/documents/Presentation_OpenTOSCA.pdf (cit. on p. 29).

[OTOSCb] *What is OpenTOSCA*. URL: http://www.opentosca.org/index.html (cit. on p. 29).

[OTTC] O. T. T. Committee. *What is TOSCA? - Frequently Asked Questions*. URL: https://www.oasis-open.org/committees/tosca/faq.php (cit. on p. 14).

[PB03] R. Pottinger, P. A. Bernstein. “Merging models based on given correspondences.” In: *VLDB ’03 Proceedings of the 29th international conference on Very large data bases - Volume 29*. Vol. 29. VLDB Endowment, Sept. 2003, pp. 862–873. ISBN: 0-12-722442-4 (cit. on p. 52).

[POP94] D. S. Weld. *An introduction to least commitment planning*. 1994 (cit. on p. 42).

[SAMT14] S. Agarwal. *A Service-oriented and Cloud-based Statistical Analysis Framework*. Nov. 2014 (cit. on p. 21).

[SBKL17] K. Saatkamp, U. Breitenbücher, O. Kopp, F. Leymann. "Topology Splitting and Matching for Multi-Cloud Deployments." In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science (CLOSER 2017)*. SciTePress, Apr. 2017, pp. 247–258. ISBN: 978-989-758-243-1 (cit. on pp. 35, 46, 53, 59, 61, 62, 66, 83).

[TCATTC] O. T. T. Committee. *OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC*. URL: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca (cit. on pp. 14, 23).

[TOSCAa13] O. T. T. Committee. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. Nov. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf (cit. on pp. 23, 25).

[TOSCAb13] O. T. T. Committee. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*. Jan. 2013. URL: http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf (cit. on pp. 24, 26, 27).

[Yu06] C. Yu. "Model-driven and pattern-based development using Rational Software Architect: Part 1. Overview of the model-driven development paradigm with patterns." In: (Nov. 206) (cit. on p. 20).

[ZBL17] M. Zimmermann, U. Breitenbücher, F. Leymann. "A TOSCA-based Programming Model for Interacting Components of Automatically Deployed Cloud and IoT Applications." In: *Proceedings of the 19th International Conference on Enterprise Information Systems - Volume 2: ICEIS*. SciTePress, Apr. 2017, pp. 121–131 (cit. on p. 23).

All links were last followed on October 24, 2017.

Declaration

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

---

place, date, signature