

Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master Thesis

Generation of BPMN 2.0 Plans to Deploy Applications in OpenTOSCA

Kuang-Yu Li

Course of Study:	INFOTECH
Examiner:	Prof. Dr. Dr. h. c. Frank Leymann
Supervisor:	Lukas Harzenetter, M.Sc., Benjamin Weder, M.Sc.
Commenced:	November 11, 2021
Completed:	May 11, 2022

Abstract

Cloud Computing has become highly market penetrating for its economic and technical benefits such as elasticity, outsourcing, pay-per-use pricing models, scalability, and self-service usage of services. Automating the provisioning, configuration, and management of complex applications is one of the most difficult challenges in Cloud Computing because each technology and API comes with individual formats, domain-specific languages, invocation mechanisms, and prerequisites. *Topology and Orchestration Specification for Cloud Applications (TOSCA)* enables the standardized descriptions of Cloud applications and their management to overcome these challenges. A TOSCA application contains all software artifacts required to provision, operate, and manage the application, which can be run in standard-compliant TOSCA runtime environments which can be hosted across different providers. The OpenTOSCA ecosystem is one of the state of art runtime environments for TOSCA-based applications. It enables fully automated deployment and management of applications defined in TOSCA. TOSCA specifies two approaches for processing applications: *imperative processing* and *declarative processing*. The imperative approach requires an application to include imperative plans, which are executable process models, for different purposes such as provision, scale, migration, and termination. The imperative approach enables the full customization of automated management of complex applications. However, manual generation of these plans is not only error-prone but also time-consuming due to the immense complexity of tasks and heterogeneous components involved in the application. On the other hand, the declarative approach uses TOSCA runtime environments to infer the management logic from the topology of components in an application without the need for predefined plans. With the declarative approach, modelers may concentrate on the application's structure without having to create plans. Because application developers are unable to explicitly describe every functionality, the declarative approach is confined to simple application management. Since the two approaches do not exclude each other, it is preferable to provide an automated plan generation that can benefit from both approaches. The benefit is that the explicit declarative modeling of a complex application's structure is preserved while providing the developers with the ability to customize the generated imperative plans for individual requirements without creating the plan completely. *Business Process Model and Notation (BPMN)*, a workflow language with a standardized graphical notation and well-defined execution semantics, fits well in the context of model plans. As a result, a TOSCA runtime that supports automated plan generation and execution in BPMN is extremely desirable.

This work develops the concepts for automatically generating imperative provision plans in executable BPMN for cloud applications based on the TOSCA standard using a hybrid approach. The graphical component of the BPMN plan is generated in human-understandable execution order, which enables the application developer to customize the generated plan for individual requirements. This hybrid approach benefits the strengths of imperative and declarative processing. A prototype was developed based on the OpenTOSCA ecosystem to test the feasibility of the concepts. The concept and implementation for BPMN plan generation complete OpenTOSCA ecosystem with end-to-end support in BPMN, from modeling to provisioning.

Contents

1	Introduction	15
1.1	Problem Statement	16
1.2	Scope of Work	17
2	Fundamentals	19
2.1	OASIS TOSCA	19
2.2	OpenTOSCA	23
2.3	BPMN 2.0	26
3	Related Work	35
3.1	BPMN Extension for TOSCA	35
3.2	Combining Declarative and Imperative Processing for Plan Generation	36
3.3	Automated Cloud Service Provisioning Using BPMN	38
4	Concept and Specification	41
4.1	Use Case	41
4.2	Requirement Specifications	41
4.3	BPMN Plan Execution Context	42
4.4	BPMN Build Plan Generation	43
5	Design and Implementation	55
5.1	Design Overview	55
5.2	Implementation	58
6	Conclusion and Future Work	63
	Bibliography	65

List of Figures

2.1	TOSCA Concepts and Metamodel (based on [BBK+14])	20
2.2	Service Template Processing with TOSCA (based on [BEK+16])	22
2.3	OpenTOSCA Container Architecture [BBK+14]	24
2.4	BPMN Elements [Whi04]	26
2.5	Example of BPMN Process Diagram [Sil11]	27
2.6	BPMN Definitions Class Diagram [OMG11]	29
2.7	Expansion of BPMN Subprocess <i>Fill Order</i> [Sil11]	30
2.8	Example of BPMN Process with BPMNDI	31
2.9	Example of BPMN Operational Semantics	34
3.1	Steps of Plan Generation in [BBK+14]	37
4.1	Example of Script Task Request Response with OpenTOSCA Container API	43
4.2	Logical Order Example	45
4.3	Node Template Subprocess Example	47
4.4	Node Template Type Plugin	47
4.5	Example of Node Template Property Parameter Handling	49
4.6	Example of Docker Container Pattern	49
4.7	Example of Docker Engine Pattern	50
4.8	Example of Diagram Position in BPMN Plan	51
4.9	Example of Diagram Position in BPMN Subprocess	51
5.1	Sample Application Topology for Prototype	56
5.2	Plan Generator Architecture [Kép13]	57
5.3	UML Class Diagram for BPMN Data Model	59
5.4	UML Class Diagram for BPMN Plan Builder	60
5.5	UML Class Diagram for BPMN Plugin	62

List of Listings

2.1	Example of BPMN Process with BPMNDI	32
3.1	Data Structure of Tasks, Requirements, and Order [CCMT17]	39
4.1	Example XML Snippet for Process Elements Finalization	53
4.2	Example XML Snippet for Diagram Elements Finalization	53

List of Algorithms

4.1	BPMN Skeleton Generation Algorithm	46
4.2	BPMN Diagram Generation Algorithm	52

Acronyms

BFS Breadth-First Search. 44

BPEL Business Process Execution Language. 15

BPMN Business Process Model and Notation. 15

CSAR Cloud Service Archive. 19

DA Deployment Artifact. 21

DOM Document Object Model. 52

IA Implementation Artifact. 21

OASIS Organization for the Advancement of Structured Information Standards. 19

POG Provisioning Order Graph. 37

TOSCA Topology and Orchestration Specification for Cloud Applications. 15

UML Unified Modeling Language. 27

WAR Web Application Archive. 21

XML Extensible Markup Language. 17

1 Introduction

Cloud Computing has gained widespread popularity within the IT industry because of the economic benefits it has to offer and the technical ease and flexibility it provides for IT operations and management [Ley09]. Properties such as high scalability, high availability, and pay-as-you-go pricing models have led to a significant shift in the way today's applications are developed and deployed [Pet11]. 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) have further enabled the ease of development and deployment of complex applications within a short span of time [Pet11]. These service models provide companies a competitive edge by significantly reducing the time to value and time to market for their products.

On the other side, Cloud providers have to automate their internal Cloud management processes to achieve these properties for their Cloud offerings. Especially the rapid provisioning of applications is of vital importance to enable self-service and pay-on-demand pricing. Therefore, one of the most important issues from a Cloud provider's perspective is to fully automate these provisioning processes. However, automating the provisioning, configuration, and management of complex applications is one of the most difficult challenges in Cloud Computing because each technology and API comes with individual formats, domain-specific languages, invocation mechanisms, and prerequisites [BBK+13]. To tackle these issues, standardization efforts such as Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13b] are paving the way to enable standardized descriptions of Cloud applications and their management.

TOSCA is an OASIS standard to describe Cloud applications and their management in a portable and interoperable way. The goal of the standard was to address the growing complexity of Cloud applications. To achieve portability, a TOSCA application contains all software artifacts required to provision, operate, and manage the application, which can be run in standard-compliant TOSCA runtime environments. TOSCA runtime environments can be hosted across different providers. TOSCA specifies two approaches for processing applications: imperative processing and declarative processing. The imperative approach specifies precisely how a cloud application is structured and managed, while the declarative approach specifies what structural elements of a cloud application are needed and what management behavior is to be realized. The imperative approach requires an application to include imperative plans, which are executable process models. plans can be modeled by standardized workflow language such as Business Process Execution Language (BPEL) [OAS07] and Business Process Model and Notation (BPMN) [OMG11]. There are several kinds of plans. For example, a *Build Plan* is a plan that is able to install, deploy and provision the modeled application; a *Termination Plan* is a plan that is able to deinstall, undeploy, and deprovision the modeled one. Plans may implement arbitrary functionality by orchestrating all kinds of scripts, programs, APIs, etc. Therefore, the imperative approach enables the full customization of automated management of even more complex applications. However, manual generation of these plans is not only error-prone but also time-consuming due to the immense complexity of tasks and heterogeneous components involved in the application. On the other hand, the declarative approach requires TOSCA runtime

environments to automatically infer the management logic and related operations from the topology of components in an application without the need for imperative plans. The declarative approach enables modelers to focus on the application's structure without the need of creating plans. The declarative approach is limited to simple application management as application developers are not able to define precisely every logic. Because the two approaches do not exclude each other, it is preferable to provide an automated plan generation that can benefit from both approaches. The benefit is that the explicit declarative modeling of a complex application's structure is preserved while providing the developers with the ability to customize the generated imperative plans for individual requirements without creating the plan completely.

The *OpenTOSCA* ecosystem is an open-source TOSCA toolchain, consisting of standard-compliant components that enable modeling applications and automating their provisioning and management [BEK+16]. As part of the ecosystem, the *OpenTOSCA Container* is a runtime for TOSCA-based applications. It enables fully automated deployment and management of applications defined in TOSCA. The runtime supports the declarative provisioning and management approach as well as the imperative approach. The OpenTOSCA Container can interpret application topology to infer management logic without the need for the manual creation of plans. The Container supports generating imperative build, termination, and other plans based on declarative TOSCA models, which provide well-defined semantics of components in the application and reusable templates for properties and operations of the components. The generated plan can be deployed and executed to the integrated workflow engines: Apache ODE [ASF16] for BPEL-based plan deployment and execution and Camunda BPM platform ¹ for BPMN-based ones. Currently, the OpenTOSCA Container only supports generating BPEL-based plans.

Apart from well-defined execution semantics, workflow language BPMN fits in the context of model plans due to the advantage that it offers a standardized graphical notation. Tasks and the control flow between them are the central elements of BPMN to what the workflow does and in which order. BPMN defines tasks, e.g., to call services (service task), to execute scripts (script task), to trigger human actions (human task), which fits in the context of deploying and managing applications [KBBL12]. Therefore, It is highly desirable to provide a TOSCA runtime that supports automated plan generation and execution in BPMN 2.0 with a hybrid approach.

1.1 Problem Statement

As mentioned in the previous section, the OpenTOSCA Container runtime automatically generates BPEL plans by interpreting the application structure and its dependencies from the definitions in the TOSCA application. Although this approach aids in solving the problem of automatic provisioning of cloud applications, in its current form, it suffers from two drawbacks which are highlighted in the following.

First, BPEL left the visual representation dimension of such process models completely out of scope. Without the visual component, it makes the application developer [OAS13a], who is not an expert in BPEL, difficult to understand or interpret the generated imperative plan from the original declarative topology. It also poses difficulties for the developer to monitor and manage the steps

¹<https://camunda.com/products/camunda-platform/>

of execution of the plan. BPEL focuses on language aspects and operational semantics aspects of business processes to achieve “time to market”. This is a mistake when looking back in time from today [Ley10]. This visual dimension of BPMN fills this gap. By providing a visual modeling language for business processes, BPMN enables non-IT experts to communicate and mutually understand their models: a big progress in this area resulting in the wide-spread use of BPMN.

Secondly, it is almost impossible to modify or debug generated BPEL plan for the application developer in OpenTOSCA due to the lack of BPEL TOSCA extension and visualizing modeler tool. This makes the application developer fully rely on the correctness of the implementation of Plan Generator, deprived of the flexibility to extend based on the generated plan. Language extensions such as BPMN4TOSCA [KBBL12] and tools such as BOWIE² are targeted for the BPMN plan.

The purpose of this thesis is to explore the possibility of automatic generation of imperative plans in BPMN 2.0 for TOSCA application. This in turn would provide a possible approach to alleviating the problems of plans generated in BPEL. At the same time, this work also acts as proof of concept for future integration with existing BPMN tools in OpenTOSCA.

1.2 Scope of Work

This master thesis provides a concept and implementation of software components to generate BPMN 2.0 Build Plans in the OpenTOSCA Container. The component architecture design and implementation of the BPMN Plan Generator are based on the OpenTOSCA Container. With generated BPMN plan in Extensible Markup Language (XML) format, users are able to visualize the plan with BPMN tools such as Signavio³ and execute the plan with the integrated Camunda BPMN engine¹ in Container runtime. The concept and implementation for BPMN plan generation complete OpenTOSCA ecosystem with end-to-end support in BPMN from modeling to provisioning. The contribution of the thesis are listed as follows:

- Design and implement algorithms for generating BPMN process and diagram elements from Service Template
- Design and implement BPMN Plan Generator prototype to generate Build Plans in BPMN 2.0.
- Validate the generation and execution of BPMN Plan with unit test cases derived from existing TOSCA Topology Templates
- Integrate BPMN Plan Generator component into the OpenTOSCA Container with user-specific Plan language option.

The remainder of this thesis is structured as follows. Chapter 2 discusses the important terms and technologies that are necessary to understand the concepts introduced in the thesis. Chapter 3 discusses various research works related to generating provisioning plan in BPMN 2.0 for cloud application. Chapter 4 discusses the high-level concepts of this thesis. These concepts include use case, requirement, BPMN plan execution context, and plan generation. Chapter 5 describes the

²<https://github.com/OpenTOSCA/bowie>

³<https://www.signavio.com/>

low-level design and implementation of the concepts. The general architecture of the BPMN Plan Generator and the design of each individual component are discussed in detail. Chapter 6 presents a summary of the results of this work and provides an outlook for future work.

2 Fundamentals

In this chapter, the various terms and technologies that are necessary for a better understanding of the concepts introduced in this thesis are discussed. First, in Section 2.1, a brief overview of the OASIS TOSCA standard is introduced. Then, in Section 2.2, the concepts and tools of the OpenTOSCA ecosystem are introduced. The focus is on the OpenTOSCA Container, a runtime for TOSCA, and its Plan Generator component. Lastly, in Section 2.3, the essential features of BPMN 2.0 are presented.

2.1 OASIS TOSCA

The Topology and Orchestration Specification for Cloud Applications (TOSCA) [OAS13b] is an official standard introduced by the Organization for the Advancement of Structured Information Standards (OASIS) consortium. The goal of the standard is to address the growing complexity of Cloud applications and enables a standardized way to design and manage the applications in a portable and interoperable manner [OAS13a]. The subsequent sections discuss the overview, concepts, and metamodel of the TOSCA specification that are most relevant to the understanding of the concepts presented in the Chapter 4 and Chapter 5.

2.1.1 TOSCA Concepts and Metamodel

This section describes the underlying TOSCA concepts and its metamodel. The TOSCA standard enables modeling the application's structure in the form of *Topology Templates* and employs the concept of executable plans to describe all required management functionality. To specify the semantics of components and their relationships, *Node Types* and *Relationship Types* can be described in TOSCA. To provision and manage an application, different kinds of artifacts, *Implementation Artifacts* and *Deployment Artifacts*, are required to be included in TOSCA application. The main building blocks of a TOSCA model are shown in Figure 2.1 and are summarized in the following.

Cloud Service Archive

TOSCA defines a Cloud Service Archive (CSAR) that serves as a portable packaging format to package all the components of an application. Standard-compliant CSARs can be consumed by any TOSCA runtime environment to deploy and manage the described application. Thus, portability is achieved by using a standardized metamodel and a standardized packaging format.

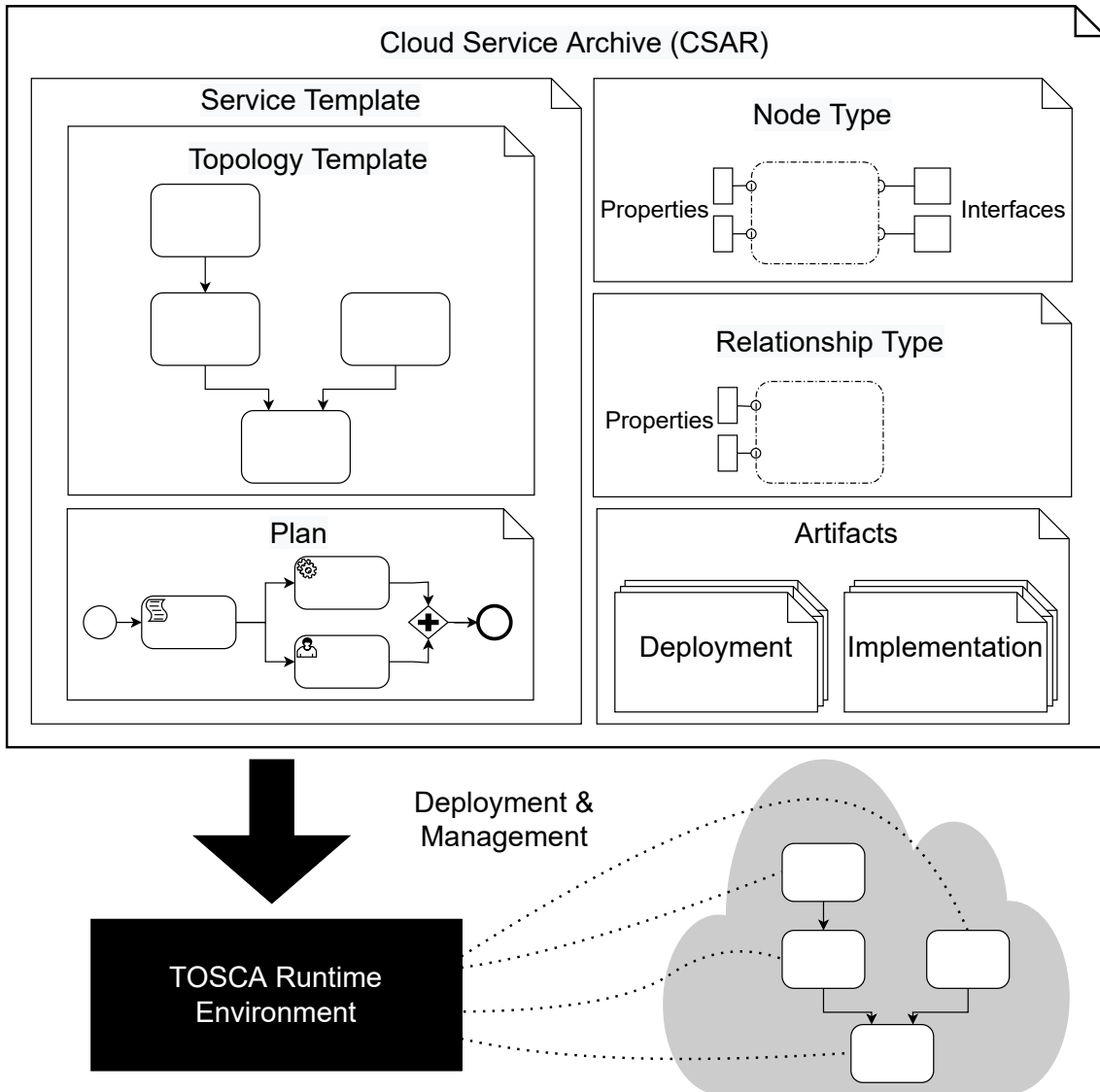


Figure 2.1: TOSCA Concepts and Metamodel (based on [BBK+14])

Service Templates

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. Service Template is deployed by running a special plan defined for the Service Template, often referred to as Build Plan. A deployed service is an instance of a Service Template.

The Build Plan provides actual values for the various properties of the various Node Templates and Relationship Templates of the Topology Template. These values can come from (i) input passed in by users as triggered by human interactions defined within the Build Plan, (ii) automated operations

defined within the Build Plan (such as a directory lookup), or (iii) the default values of properties specified in templates. The Build Plan typically makes use of operations of the Node Types of the Node Templates.

Topology Templates

A Topology Template consists of a set of Node Templates and Relationship Templates that together define the topology model of a service as a (not necessarily connected) directed graph. A node in this graph is represented by a Node Template. A Node Template specifies the occurrence of a Node Type as a component of a service. A Relationship Template specifies the occurrence of a relationship between nodes in a Topology Template. Each Relationship Template refers to a Relationship Type that defines the semantics and any properties of the relationship.

Node Types and Relationship Types

A Node Type defines the properties of such a component, via Node Type Properties, and the operations, via Interfaces, available to manipulate the component. Node Types are defined separately for reuse purposes and a Node Template references a Node Type and adds usage constraints, such as how many times the component can occur. Relationship Types are defined separately for reuse purposes. The Relationship Template indicates the elements it connects and the direction of the relationship by defining one source and one target element in nested *SourceElement* and *TargetElement* elements.

Implementation Artifacts and Deployment Artifacts

An Implementation Artifact (IA) represents the executable of an operation of a node type, and a Deployment Artifact (DA) represents the executable for materializing instances of a node. IA can either be simple Shell Scripts or may be packaged into Web Application Archive (WAR) files. DA can be image file to instantiate a VM or a Node JS script implementing a simple application.

Plans

Plans defined in a Service Template describe the management aspects of service instances, especially their creation and termination. These plans are defined as process models, i.e. a workflow of one or more steps. Relying on existing standards like BPMN or BPEL facilitates portability and interoperability. 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.

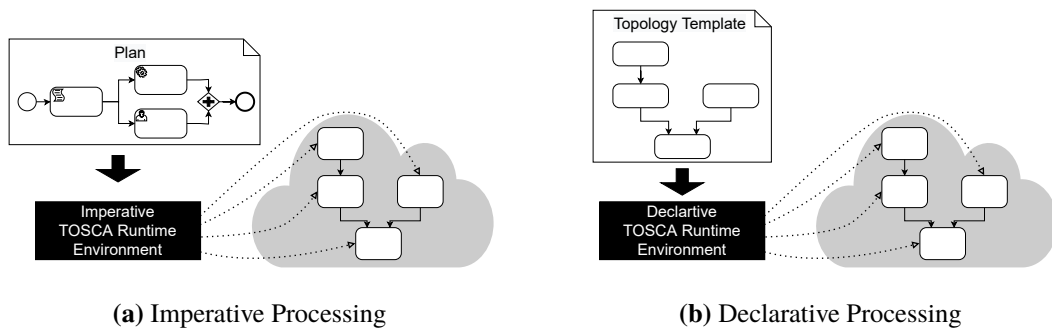


Figure 2.2: Service Template Processing with TOSCA (based on [BEK+16])

2.1.2 Service Template Processing

For processing Service Template of the modeled application, TOSCA distinguishes between two approaches: (i) imperative processing and (ii) declarative processing [OAS13a]. In short, the imperative processing flavor specifies precisely how a cloud application is structured and managed, while the declarative processing flavor specifies what structural elements of a cloud application are needed and what management behavior is to be realized. In this section, both approaches are introduced.

Imperative Processing

In this section, the concept of imperative processing is explained. Imperative processing requires that all needed management logic is contained in the CSAR. Therefore, CSARs contain fully automatically executable plans that imperatively describe high-level management tasks such as provisioning, scaling, or updating the application. A plan is a workflow orchestrating low-level management operations that are either provided by the application components themselves or by publicly accessible services, e. g., the Amazon Web Services API. These low-level operations typically encompass small functionality. To make plans portable, the TOSCA Plan Portability API is used by plans to communicate with the runtime, e. g., to access the Topology Template. Standardized workflow modeling such as BPEL and BPMN enables plans to be executed on any standard-compliant workflow engine.

An imperative TOSCA runtime environment executes plans contained in CSAR. As shown in Figure 2.2a, a workflow engine is employed to execute plans that are modeled as workflows. After consuming the CSAR, the environment extracts all plans and deploys them onto the engine. As a result, the plans can be invoked to execute the respective functionality. In the case of imperative provisioning, the Build Plan is executed for provisioning the modeled application.

Declarative Processing

In this section, the concept of declarative processing is explained. The declarative processing shifts management logic from plans to runtime. Contrary to the imperative approach, with the declarative provisioning approach, the topology model gets interpreted by a Declarative TOSCA runtime

environment, as shown in Figure 2.2b. The TOSCA runtime environment extracts the topology model including all type definitions and interprets application topologies to infer management logic without the need for plans. The set of provided management functionalities depends on the corresponding runtime and is not standardized by the TOSCA specification. This requires a precise definition of the semantics of nodes and relations based on well-defined Node Types and Relationship Types. For example, if component A has a hosted-on-dependency to component B, component B has to be provisioned first. TOSCA Primer defines the so-called Lifecycle Interface, which defines the semantics of the management operations install, configure, start, stop, and terminate [OAS13a]. Based on IAs that implement these operations, imperative plans as well as declarative TOSCA runtime environments are able to provision the individual components by executing these artifacts

The declarative approach enables modelers to focus on the application's structure without the need of creating Build Plans that orchestrate the management operations offered by Node Templates. Thus, this approach is much easier [BBK+14]. However, the approach is limited in terms of the application's complexity and works only for simple applications that employ common, semantically-defined component types. If a complex application consists of custom components, the imperative approach has to be used since arbitrary management logic can be modeled.

Both imperative and declarative processing approaches do not mutually exclude each other. There are concepts and technologies that are able to generate imperative plans out of declarative models, e.g., [BBKL13]. Build Plan Generator in OpenTOSCA ecosystem used hybrid approach to generate Build Plans that combines the benefits of both worlds

2.2 OpenTOSCA

In this section, an overview of the OpenTOSCA ecosystem including the architecture and components of OpenTOSCA Container is presented.

2.2.1 Overview of OpenTOSCA Ecosystem

The OpenTOSCA ecosystem mainly consists of the following tools: the TOSCA modeling tool Winery [KBBL13], the TOSCA runtime environment OpenTOSCA Container [BBH+13], and the TOSCA Self-Service Portal Vinothek [BBKL14]. These three tools work together and build an ecosystem to (i) model, (ii) deploy, (iii) manage, and (iv) instantiate TOSCA-based applications. The entire OpenTOSCA ecosystem ¹ including all tools is an open-source implementation and publicly available on GitHub ² and Eclipse ³.

The interplay of the three main tools is as follows. Winery is a standard-compliant TOSCA modeling tool that enables creating topology models using a graphical web-based editor. Moreover, Winery provides a backend system to create and maintain Node Types, Relationship Types, and other entities defined by the TOSCA metamodel. Winery supports exporting CSARs that contain all required files to deploy and manage the corresponding application using the OpenTOSCA Container.

¹<http://www.iaas.uni-stuttgart.de/OpenTOSCA/>

²<https://www.github.com/OpenTOSCA>

³<https://projects.eclipse.org/projects/soa.winery>

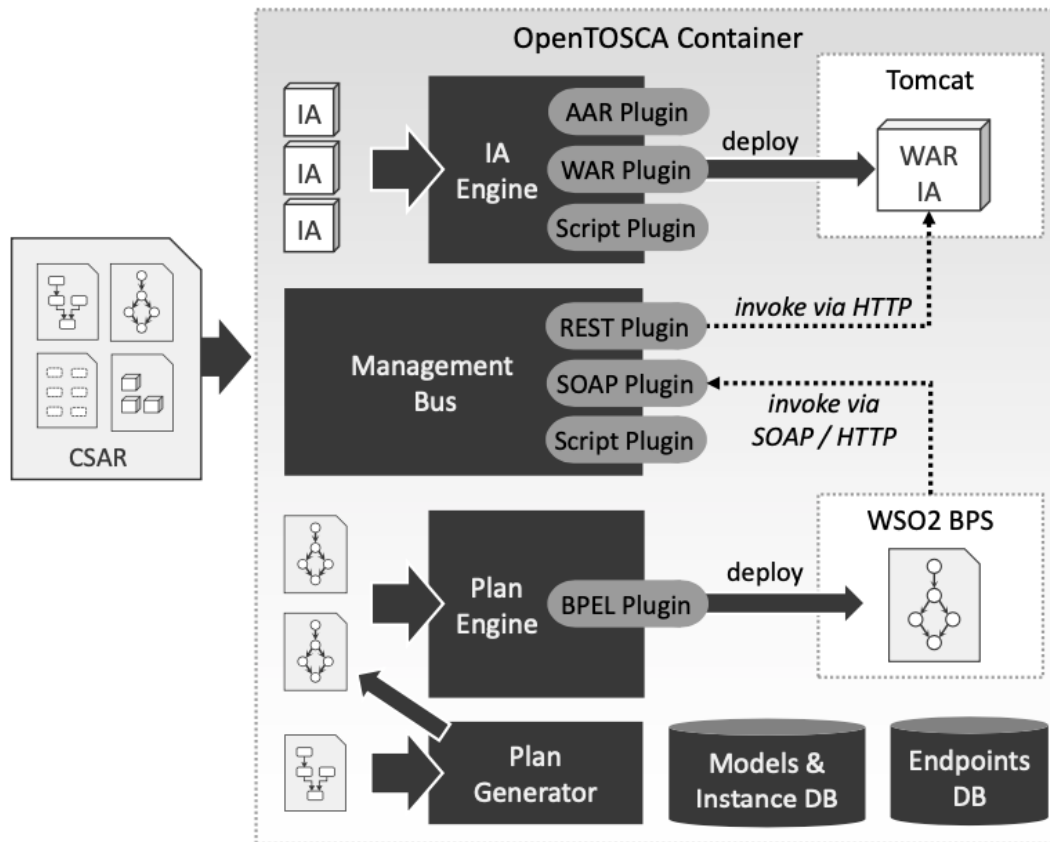


Figure 2.3: OpenTOSCA Container Architecture [BBK+14]

The OpenTOSCA Container is a standard-compliant TOSCA runtime environment that is able to consume CSARs for deploying the modeled application. The runtime environment supports the imperative processing and management approach by enabling the execution of BPEL-based and BPMN-based workflow models [OAS07]. Besides Build Plans, the runtime is able to execute arbitrary plans to manage a certain application instance. Moreover, the OpenTOSCA Container also supports the declarative processing approach in Section 2.1.2. Thus, the Container provides a Hybrid TOSCA runtime environment that supports both provisioning flavors.

The web-based self-service portal Vinothek provides an easy and intuitive user interface for end-users. This portal offers all applications that are installed in the OpenTOSCA Container and enables end-users to instantiate new instances of an application by just clicking on a provisioning button.

2.2.2 The OpenTOSCA Container

In this section, the OpenTOSCA Container, a TOSCA runtime environment of the OpenTOSCA ecosystem is described. The Container is able to process CSARs in a declarative and imperative manner, thus, providing a hybrid provisioning environment that is seamlessly integrated with Winery. The general processing of the Container can be described as follows. The OpenTOSCA Container enables the automated provisioning of Cloud applications that are modeled using TOSCA and

packaged as CSARs. To realize the provisioning, the container analyses the contained TOSCA model and invokes the contained Build Plan to instantiate a new application instance of the contained TOSCA model. If there is no Build Plan available, the Container generates a Build Plan on its own by using the Plan Generator component and invokes this plan for provisioning the application. During the lifetime of the application, the container enables managing application instances by invoking plans contained in the respective CSAR, for example, to scale application components. Plans can be either modeled manually using Winery or generated for provisioning, as mentioned above. Thus, the OpenTOSCA Container supports both imperative and declarative application provisioning approaches.

Figure 2.3 depicts an overview of the OpenTOSCA Container architecture. The Container mainly consists of the following components, which are explained in detail in the following subsections. The IA Engine is responsible for processing the IAs contained within the CSAR. Similarly, the Plan Engine is responsible for the processing of plans. The Management Bus [WBB+14; WBB+15] is a communication middleware inside the container that enables plans to invoke different kinds of management operations through a unified interface. The Plan Generator [BBK+14] allows generating imperative Build Plans based on declarative TOSCA models. Also, the OpenTOSCA Container needs data storages to manage information about, for example, known CSARs or available service endpoints. These information are stored in two independent databases: the TOSCA Models & Instance Database and the Endpoints Database.

Implementation Artifact Engine

Implementation Artifacts are contained in the CSAR that is passed to the OpenTOSCA Container and processed by the IA Engine. In OpenTOSCA, local IAs are SOAP-based WARs that are executed by the OpenTOSCA Container by deploying them on a local Apache Tomcat Servlet Container⁴. At the moment, the OpenTOSCA Container and its IA Engine contains plugins to support the processing of Java WARs and Axis Archives (AARs) [WBB+14].

Plan Engine

Similar to the IA Engine, the Plan Engine is responsible for the processing of the plans contained in a CSAR. Since plans implement imperative provisioning and management logic for the application inside the CSAR, plans need to be invocable and executable. Currently, the OpenTOSCA Container employs two local workflow engine: Apache ODE [ASF16] and Camunda. Apache ODE is used to deploy BPEL-based plans and to make them executable. Camunda is used to deploy BPMN-based plans. The Plan Engine provides a plugin-system for adding support for other languages.

Management Bus

Management Bus provides a uniform interface for invoking different kinds of IAs. OpenTOSCA offers a SOAP/HTTP-based API that can be easily invoked using BPEL workflow models. Thus, plans themselves only invoke the operation via this API while all technical details about the actual

⁴<https://tomcat.apache.org/>

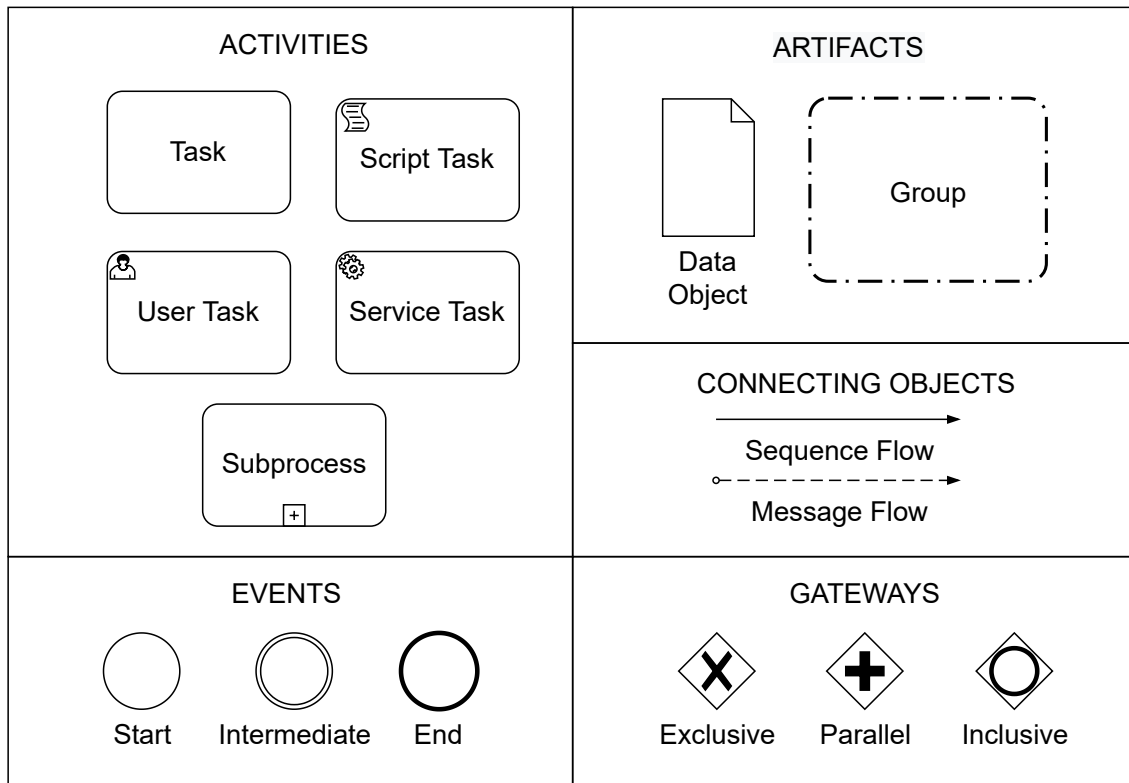


Figure 2.4: BPMN Elements [Whi04]

execution of the associated IA is handled and hidden by the bus. The Management Bus also supports the asynchronous communication used by the Java-based Web Services generated using the modeling tool Winery Wettinger et al. [WBB+14; WBB+15].

Plan Generator

The Plan Generator is an integrated processor of declarative TOSCA topology models and generates an imperative BPEL-based plan. As depicted in Figure 2.3, generated Build Plans are passed to the Plan Engine, which makes them executable by deploying them onto the local workflow engine. BPEL plan generation details are described in Chapter 3.

2.3 BPMN 2.0

This section presents a high-level overview of the BPMN standard. The following part introduces BPMN 2.0, the latest major version of BPMN. The basic concepts are introduced in Section 2.3.1. Section 2.3.2 describes the underlying metamodel of BPMN. Section 2.3.3 presents the basic elements in BPMN. Section 2.3.4 describes the graphical model of BPMN diagram. The execution semantic of BPMN 2.0 is described in Section 2.3.5. Lastly, the executable of BPMN is described in Section 2.3.6.

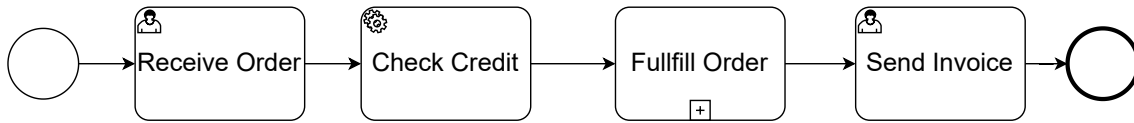


Figure 2.5: Example of BPMN Process Diagram [Sil11]

2.3.1 BPMN Basic

A business process (BP) is a set of one or more linked procedures or activities executed following a predefined order which collectively realize a business objective or policy goal, normally within the context of an organizational structure defining functional roles or relationships. The Object Management Group (OMG) has developed a standard Business Process Model and Notation (BPMN). The primary goal of BPMN is to provide a notation that is readily understandable by all business users, from the business analysts that create the initial drafts of the processes, to the technical developers responsible for implementing the technology that will perform those processes, and finally, to the business people who will manage and monitor those processes. BPMN creates a standardized bridge for the gap between the business process design and process implementation [Whi04]. BPMN provides a graphical notation in order to represent a business process as a diagram. BPMN defines the notation and semantics of three types of diagram: *Process*, *Collaboration*, and *Choreography*. A Process describes a sequence or flow of *Activities* in an organization with the objective of carrying out work, which fits in the context of plan in TOSCA. Therefore, this work focuses on the Process diagrams of BPMN.

BPMN defines elements that be used in more than one type of diagram [Sil11]. A summary of the most common BPMN elements is shown in Figure 2.4. Starting from BPMN 1.2 the number of elements increases. BPMN 1.x versions did not have clearly defined semantics nor a native serialization format. The BPMN 2.0 specification extends the scope and capabilities of the BPMN 1.2 in several areas: (i) it formalizes the execution semantics for all BPMN elements, (ii) it defines an extensibility mechanism for both Process model extensions and graphical extensions, refines Event composition and correlation, (iii) it extends the definition of human interactions, defines Choreography and Conversation models, and (iv) it also resolves known BPMN 1.2 inconsistencies and ambiguities [CT12].

The BPMN diagram is both a visualization and a data entry device for the underlying metamodel model. An example of BPMN Process diagram is shown in Figure 2.5. The example Process is based on a scenario of a company's process to handle an order. The company receives the order, checks the buyer's credit, fulfills the order, and sends an invoice. The thin circle at the start of the process is called a start event. It indicates where the process starts. The thick circle at the end is called an end event, signifying the process is complete. The rounded rectangles are activities. The result detail elements of the Process diagram will be elaborated in Section 2.3.3.

2.3.2 BPMN Metamodel

BPMN 2.0 metamodel contains a semantic model and a graphical model and is represented by Unified Modeling Language (UML) class diagrams, augmented by tables and text in the narrative. For example, Figure 2.6 depicts the Definitions class. The classes are organized in sets called

packages. The packages are layered for extensibility, each layer building on and extending lower layers. The metamodel is published in two XML formats: XML Metadata Interchange (XMI) and XML Schema Definition (XSD). This work concerns only the Process package in XSD. The graphical model, called BPMNDI, contains information concerning the graphical layout of shapes, such as position, size, and connection points. The semantic model reference the semantic elements, such as a start event, a User task, an end event, etc. A valid BPMN model may omit BPMNDI entirely, but may not omit the semantic model. BPMNDI without semantic model information is meaningless. Both semantic and graphical models are enclosed within a single definitions element.

Most elements in the BPMN 2.0 XSD have an ID attribute of type `xsd:ID`, a type defined by the XSD language for the use in attributes only. The values of ID must be unique within an XML instance, regardless of the attribute's name. This uniqueness is critical because relationships between model elements are maintained by pointers to other elements via their ID value. For example, a Sequence Flow's *sourceRef* attribute matches the ID of the flow node connected to the tail of the Sequence Flow. An XML instance document will not pass schema validation if any pointer elements or attributes point to an ID value that is missing in the document, or if duplicate ID values exist anywhere in the document.

2.3.3 BPMN Elements

In this section, the basic elements of BPMN are introduced. An example of BPMN Process diagram is shown in Figure 2.5.

BPMN has four categories of graphical elements to build diagrams: *Flow Objects*, *Connecting Objects*, *Swimlanes*, and *Artifacts*. The Flow Objects and Connecting Object are described in detail as they are the fundamental components of BPMN plan of this work. The basic Flow Objects are: Activity, Event, Gateway.

Activity

An Activity is represented by a rounded-corner rectangle (see Figure 2.4 to the top left) and is a generic term for work that company performs. An Activity can be atomic or non-atomic (compound). The types of Activities are: Task and Subprocess. The Subprocess is distinguished by a small plus sign in the bottom center of the shape. In the example of Figure 2.5, an Activity like *Check Credit* represents an action, a specific unit of work performed, as distinct from a function (e.g., Credit Check) or a state (e.g., Credit OK).

Task

A Task is atomic, meaning it has no internal subparts described by the process model. A Task is represented in the diagram by the activity shape, a rounded rectangle, with the Task type indicated by a small icon in the upper left corner. The three commonly used Tasks are:

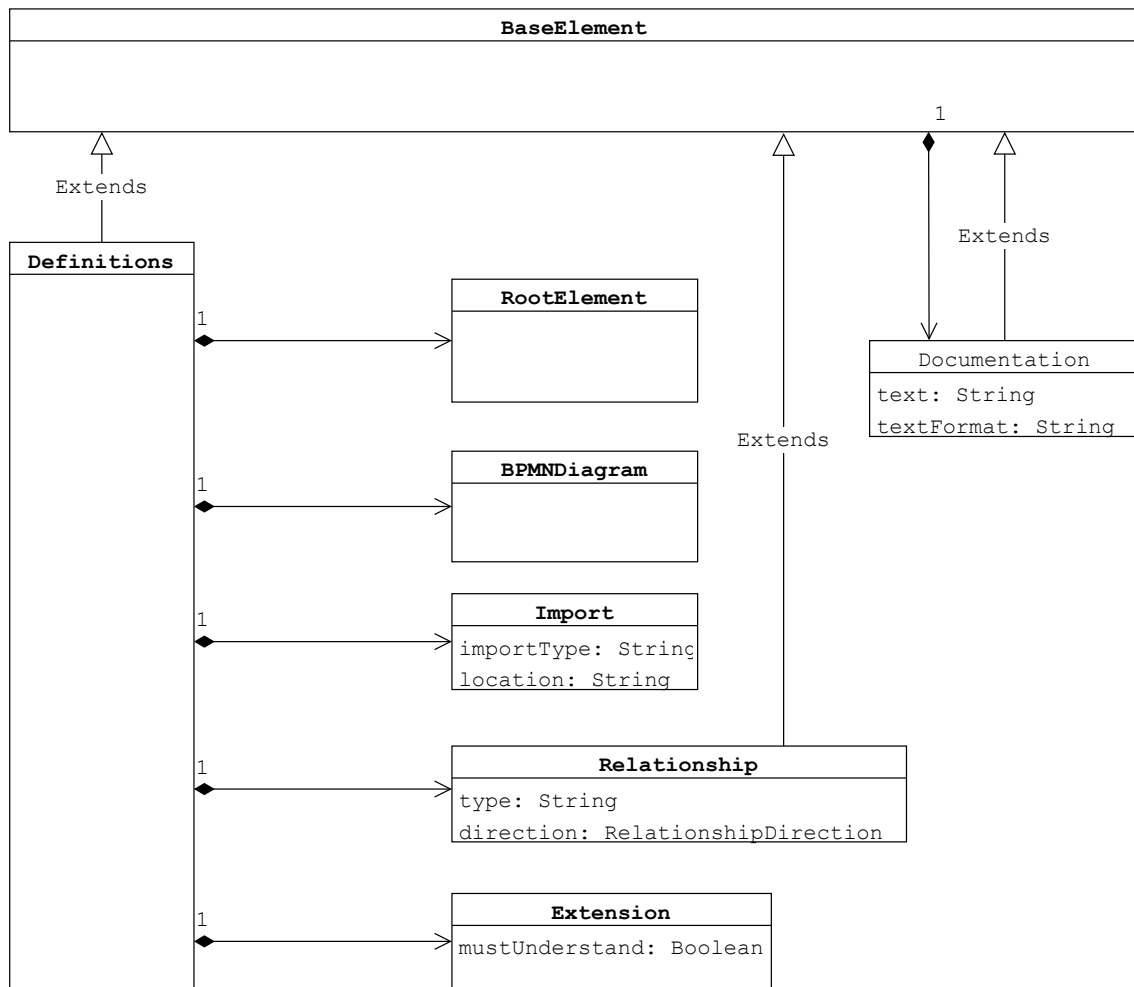


Figure 2.6: BPMN Definitions Class Diagram [OMG11]

- User Task, with the head-and-shoulders icon, means a task performed by a person. In the example of Figure 2.5, *Receive Order* and *Send Invoice* are two human task which requires human, sale person, to perform the task.
- Service Task, with the gears icon, means an automated activity. Automated means when the sequence flow arrives, the task starts automatically, with zero human intervention. If a person has to just click a button and the rest is automatic, that is a User Task, not a Service Task. In the example of Figure 2.5, *Check Credit* is an automated task which is done automatically by the company credit system once order is received.
- Script Task with the scroll icon means an automated function performed by the process engine itself. The implementation is a short program, typically Javascript⁵ or Groovy⁶, embedded in the process definition XML. Because the process engine is usually busy executing the process

⁵<https://developer.mozilla.org/en-US/docs/Web/JavaScript>

⁶<https://groovy-lang.org/index.html>

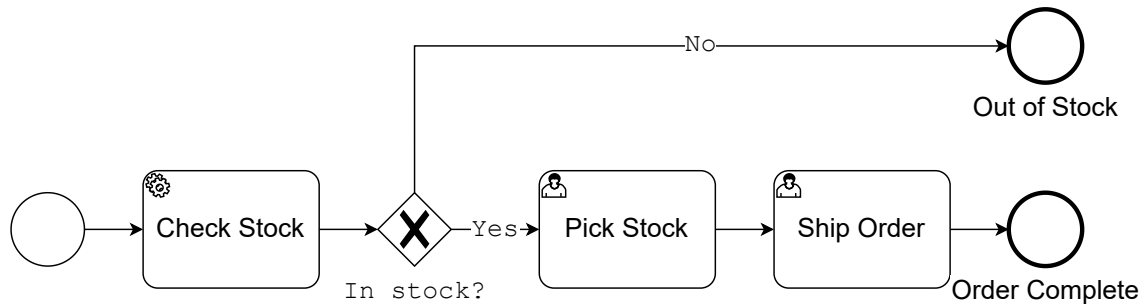


Figure 2.7: Expansion of BPMN Subprocess *Fill Order* [Sil11]

logic, it does not have time to perform complex tasks, so Script tasks are typically used for simple computations such as data mapping. Script task should only be used in an executable process.

Subprocess

A Subprocess is a compound Activity, meaning an Activity with subparts that can be described as a child-level process. A Subprocess can be represented in multiple ways in the diagram. A collapsed Subprocess is drawn in the parent-level diagram using a normal-size activity shape with a [+] symbol at the bottom center. An expanded Subprocess is drawn as an enlarged activity shape in the parent-level flow that encloses the child-level expansion in the same diagram. There is no semantic difference between a collapsed Subprocess and an expanded Subprocess. In BPMN 2.0, the only difference is in the graphical model. A Subprocess start event must have a None trigger. Subprocesses are a valuable feature of BPMN because they provide ability to (i) visualize end-to-end process (ii) enable top-down modeling, (iii) clarify governance boundaries, and (iv) Scope event handling. In the example of Figure 2.5, *Fill Order* is a collapsed Subprocess which contains other Tasks. The expanded Subprocess is shown in Figure 2.7. The Subprocess includes three tasks, *Check Stock*, *Pick Stock*, and *Ship Order*, a gateway, and three events.

Event

An Event is represented by a circle and is something that “happens” during the course of a business process. These Events affect the flow of the process and usually have a cause (trigger) or an impact (result). Events are circles with open centers to allow internal markers to differentiate different triggers or results. There are three types of Events, based on when they affect the flow: Start, Intermediate, and End. (see Figure 2.4 to the bottom left)

A Start Event is always represented as a circle with a single thin border. Its purpose is to indicate where and how a process or Subprocess starts. Normally a process or Subprocess has only one start event.

An End Event is always represented as a circle with a single thick border. It indicates the end of a path in a process or Subprocess. An End Event in either a process or Subprocess may be drawn with a black or “filled” icon inside, indicating the resulting signal thrown when the event is reached.

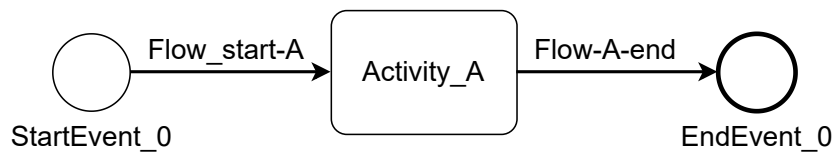


Figure 2.8: Example of BPMN Process with BPMNDI

Gateway

A Gateway is represented by the familiar diamond shape (see Figure 2.4 to the top bottom right) and is used to control the divergence and convergence of Sequence Flow. Thus, it will determine traditional decisions, as well as the forking, merging, and joining of paths. Internal Markers will indicate the type of behavior control. In the example of Figure 2.7, the Exclusive Gateway with label *In stock?* and two Sequence Flows with label *Yes* and *No* are shown. The Exclusive Gateway means take one path or the other based on some data condition. In this case, the condition is based on the stock level.

Connecting Objects connect Flow Objects together in a diagram to create the basic skeletal structure of a business process. There are three Connecting Objects: Sequence Flow, Message Flow, Association, shown in Figure 2.4 to the down right.

Sequence Flow

A Sequence Flow is represented by a solid line with a solid arrowhead and is used to show the order (the sequence) that activities will be performed in a Process. Note that the term “control flow” is generally not used in BPMN.

Message Flow

A Message Flow is represented by a dashed line with an open arrowhead and is used to show the flow of messages between two separate Process Participants (business entities or business roles) that send and receive them. In BPMN, two separate Pools in the Diagram will represent the two Participants.

2.3.4 BPMNDI

BPMN 2.0 also provides an XML schema for the graphical model, called BPMN Diagram Interchange, or BPMNDI. It describes the location and size of shapes and connectors, as well as the linked page structure of the model diagrams. In BPMN, the graphical model can never stand alone. It must be accompanied by the semantic model information. For example, the only way BPMNDI distinguishes a task shape from a Timer boundary event, or a sequence flow from a data association, is via the shape’s `bpmnElement` attribute, a pointer to the corresponding semantic element. The location of a shape is defined as the x,y coordinates of the top left corner of a rectangular bounding

2 Fundamentals

Listing 2.1 Example of BPMN Process with BPMNDI

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <bpmn:definitions>
3   <bpmn:process id="Process_Example" isExecutable="true">
4     <bpmn:startEvent id="StartEvent_0" name="StartEvent_0">
5       <bpmn:outgoing>Flow_start-A</bpmn:outgoing>
6     </bpmn:startEvent>
7     <bpmn:task id="Activity_A" name="Activity_A">
8       <bpmn:incoming>Flow_start-A</bpmn:incoming>
9       <bpmn:outgoing>Flow_A-end</bpmn:outgoing>
10    </bpmn:task>
11    <bpmn:endEvent id="EndEvent_0" name="EndEvent_0">
12      <bpmn:incoming>Flow_A-end</bpmn:incoming>
13    </bpmn:endEvent>
14    <bpmn:sequenceFlow id="Flow_start-A" name="Flow_start-A" sourceRef="StartEvent_0"
targetRef="Activity_A" />
15    <bpmn:sequenceFlow id="Flow_A-end" name="Flow_A-end" sourceRef="Activity_A" targetRef="
EndEvent_0" />
16  </bpmn:process>
17  <bpmndi:BPMNDiagram id="BPMNDiagram_1">
18    <bpmndi:BPMNPlane id="BPMNPlane_1" bpmnElement="Process_Example">
19      <bpmndi:BPMNEdge id="Flow_0igkl7g_di" bpmnElement="Flow_start-A">
20        <di:waypoint x="215" y="117" />
21        <di:waypoint x="290" y="117" />
22      <bpmndi:BPMNLabel>
23        <dc:Bounds x="222" y="99" width="62" height="14" />
24      </bpmndi:BPMNLabel>
25    </bpmndi:BPMNEdge>
26    <bpmndi:BPMNEdge id="Flow_0ykycd7_di" bpmnElement="Flow_A-end">
27      <di:waypoint x="390" y="117" />
28      <di:waypoint x="472" y="117" />
29    <bpmndi:BPMNLabel>
30      <dc:Bounds x="402" y="99" width="59" height="14" />
31    </bpmndi:BPMNLabel>
32  </bpmndi:BPMNEdge>
33  <bpmndi:BPMNShape id="_BPMNShape_StartEvent_2" bpmnElement="StartEvent_0">
34    <dc:Bounds x="179" y="99" width="36" height="36" />
35    <bpmndi:BPMNLabel>
36      <dc:Bounds x="165" y="142" width="64" height="14" />
37    </bpmndi:BPMNLabel>
38  </bpmndi:BPMNShape>
39  <bpmndi:BPMNShape id="Activity_0e17gob_di" bpmnElement="Activity_A">
40    <dc:Bounds x="290" y="77" width="100" height="80" />
41  </bpmndi:BPMNShape>
42  <bpmndi:BPMNShape id="Event_091t5n0_di" bpmnElement="EndEvent_0">
43    <dc:Bounds x="472" y="99" width="36" height="36" />
44    <bpmndi:BPMNLabel>
45      <dc:Bounds x="460" y="142" width="60" height="14" />
46    </bpmndi:BPMNLabel>
47  </bpmndi:BPMNShape>
48  </bpmndi:BPMNPlane>
49  </bpmndi:BPMNDiagram>
50 </bpmn:definitions>
```


box enclosing the shape. The size of a shape is likewise the width and height of that bounding box. The top-level element in BPMNDI is BPMNDiagram, representing a page. Each BPMNDiagram has a required child element BPMNPlane. BPMNPlane contains an ordered list of BPMNShape and BPMNEdge child elements representing the shapes and connectors on the page. The BPMNShape element represents the visualization of a single BPMN semantic element other than a connector. A BPMNEdge element is the graphical representation of a single BPMN connector.

Figure 2.8 illustrates a simple BPMN Process. The serialization, including BPMNDI, is shown in Listing 2.1. The XML namespaces are skipped for simplicity. There are three BPMNShape elements and two BPMNEdges contained in BPMNDI. Each diagram element represents an process element. For example, BPMNShape with id attribute *_BPMNShape_StartEvent_2* in line 33 represents the Start Event *StartEvent_0* in BPMN Process element in line 4. BPMNEdge with id attribute *Flow_0ykycd7_di* in line 26 represents Sequence Flow *Flow_A-end* in BPMN Process element in line 15.

2.3.5 BPMN Operational Semantic

BPMN operational semantic is based on passing tokens. An Activity or Gateway is performed when "required" tokens arrives at the Activity or Gateway. At completion, an Activity or Gateway produces tokens on all of its outbound Sequence Flows. A Start Event generates a token for each of its leaving Sequence Flows. All tokens that are generated within the process must eventually be consumed by an End Event before the process is completed. If the process is a Subprocess, it can be stopped prior to normal completion through interrupting Intermediate Events. In this situation the tokens will be consumed by an Intermediate Event attached to the boundary of the Subprocess.

- Step-0: Message arrives: Start Event generates a token. Activity A gets activated
- Step-1: Activity A completes Token is generated Parallel Gateway gets activated
- Step-2: Parallel Gateway produces tokens for each leaving Sequence Flow. Activities B and C may be performed
- Step-3: Activity B completes and produces a token. Exclusive Gateway gets activated since it only needs a single token for activation. Activity C is not activated E.g. because user is still busy.
- Step-4: Exclusive Gateway produces a token Activity C is still not activated
- Step-5: Activity D completes and produces a token Activity C completes and produces a token End Event does not consume token (that would terminate the process) because "upstream" token may reach it later on
- Step-6: Exclusive Gateway produces a token. End Event does still not consume token because another token is still "on its way"
- Step-7: Activity D consumes the token, completes and produces a token

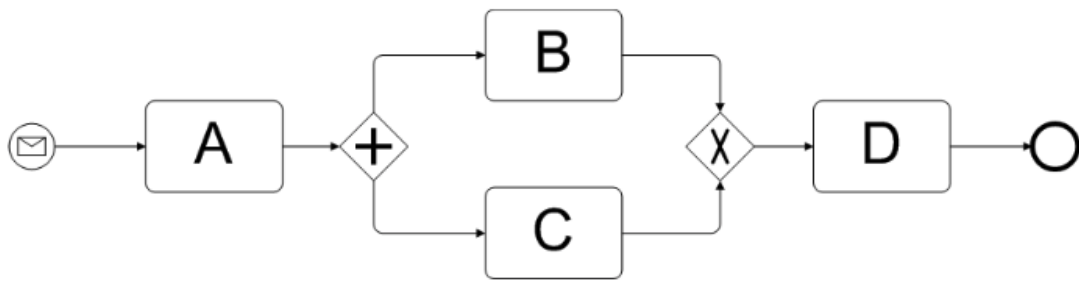


Figure 2.9: Example of BPMN Operational Semantics

2.3.6 Executable BPMN

Most of the effort in developing the BPMN 2.0 specification involves elements related to executable processes [Sil11].

In an executable process, a software engine automates the flow of model execution from process instantiation to completion. This requires additional details to be specified for each BPMN element, which includes Process variables, Task input and output data, and their mappings to variables, Task user interface forms and screenflows, Task performer assignment logic, Conditional expressions, Event definitions, and Messages.

These details are invisible in the diagram, but BPMN 2.0 provides XML elements to specify them. Silver [Sil11] refers “executable BPMN” as to a tool’s ability to specify and export execution-related details, such as those listed above, consistent with the BPMN 2.0 metamodel and schema. BPMN 2.0 spec enumerates the elements and attributes supported for basic executable BPMN, called the Common Executable subclass, which requires support of XML Schema as the type definition language, WSDL as the definition language for service interfaces, and XPath as the language for referencing data elements. Process data is at the core of executable BPMN. Whether data flow is visualized in the diagram or not, data mapping is critical to all aspects of executable BPMN. A practical way to implement complex data mapping in BPMN is to use a Script Task. A Script Task is code, embedded in the BPMN, that is executed on the process engine. A script is a set of statements, a program, not just a single expression. The script languages supported will vary from one process engine to the next. They could include Javascript or Groovy. BPMN 2.0 Common Executable subclass does not require support for any particular script language. Script Task has a *scriptFormat* attribute that specifies the script language as a MIME type string, such as *text/x-groovy* for Groovy or a child script element contains the script text, which may be enclosed in a CDATA section to prevent XML parsing of the script.

3 Related Work

In this chapter, the related research works relevant to the concepts presented in this thesis are discussed. Section 3.1 discusses research work for BPMN extension for TOSCA. Section 3.2 presents concept for Build Plan generation by combining imperative and declarative approach. Section 3.3 introduces a framework for automated cloud server provisioning for TOSCA application using BPMN.

3.1 BPMN Extension for TOSCA

Kopp et al. [KBBL12] propose BPMN4TOSCA, a BPMN 2.0 extension for TOSCA, to simplify the modeling of plans. Their motivation is to overcome the problem that, at the time of development, BPMN plans directly point to the service interfaces and are not linked to the topology anymore. BPMN4TOSCA enables convenient integration and direct access to the TOSCA topology and provided management operations. Their contribution includes: (i) analysis of the requirements for modeling TOSCA plans using BPMN, (ii) BPMN4TOSCA, a BPMN extension allowing tight integration of topology data and management operations into plans, (iii) a transformation of BPMN4TOSCA into standard-compliant BPMN, and (iv) a prototype of BPMN4TOSCA supported in a TOSCA modeling tool.

They derived the general requirements on the plan modeling language from a concrete TOSCA use case. Based on the requirements, they designed BPMN4TOSCA, which extends BPMN with four TOSCA-specific elements: (a) TOSCA Topology Management Task, (b) TOSCA Node Management Task, (c) TOSCA Script Task, and (d) TOSCA Data Object.

TOSCA Topology Management Task accesses the TOSCA service topology from plans. TOSCA Node Management Task simplifies selecting and invoking management operations of nodes. Both TOSCA Topology Management Task and TOSCA Node Management Task extend the BPMN service task. TOSCA Script Task provides the opportunity of referencing scripts and corresponding nodes on which they shall be performed. The TOSCA Script Task inherits from BPMN's script task. When the script is part of the task, the Script Task semantics and its attributes *scriptFormat* and *script* is re-used. When the task references a script stored in the service template, these two attributes are not used. Instead, three attributes are added: *scriptReference*, which references a script defined in the TOSCA file and *targetNodeTemplateId* which references the node template on which the script has to be executed, *targetNodeInstanceId*, defining the concrete instance of the node template if multiple instances are allowed. TOSCA Data Object automatically provides access to runtime property information of nodes and relationships, without the need to explicitly model BPMN service tasks requesting the respective information from the TOSCA container and sending modifications to the container. TOSCA Data Objects extends the BPMN data object by adding TOSCA-related attributes.

Since the BPMN4TOSCA extension leads to a non-standards-compliant BPMN, needs special treatment to be executed. They chose the option of transforming the functionalities into standards-compliant executable elements before deployment over extending the modeling tool and the workflow engine to support the new functionality. The reasons are that they want to preserve the portability across different standards-compliant BPMN execution environments and they want to keep all benefits of the BPMN4TOSCA for the modeler. The TOSCA Topology Management Task references operations provided by the TOSCA container. A new implementation reference is added. It points to the concrete port type, where the WSDL service of the TOSCA container is offered. The TOSCA Node Management Task references a node template in a service topology and one operation. This information is replaced by a reference to the concrete WSDL port type and WSDL operation of the referenced operation. TOSCA Script Task is replaced by three BPMN service tasks which are executed in sequence: deploy script, run script, undeploy script. The TOSCA container binds these three tasks to the services offered by the IA of the corresponding node. TOSCA Data Objects are converted to BPMN data objects. For reading TOSCA data, additional BPMN service tasks are injected. They implemented a prototype and integrated BPMN4TOSCA into the TOSCA modeling tool Valesca.

They summarized that scripts play an important role in the management of composite applications, especially during deployment. The concept and implementation presented in BPMN4TOSCA contributes the use of Script Task in imperative provisioning plan which will be further discussed in Chapter 4.

3.2 Combining Declarative and Imperative Processing for Plan Generation

Breitenbucher et al. [BBK+14] proposes a standards-based approach to generate provisioning plans based on TOSCA topology models. Their approach combines both imperative and declarative processing to support automating provisioning. Their approach resolves the drawbacks and profit from benefits of both flavor of TOSCA processing. They proved the technical feasibility of the approach by an end-to-end open source toolchain, OpenTOSCA.

The imperative approach enables application developers to define every detail of the provisioning explicitly by writing custom plans. However, there are two major drawbacks. First, creating plans manually is the labor-intensive nature of workflow authoring that is typically a hard, time-consuming, costly, and error-prone task. For examples, the challenges may result from heterogeneous management services, which needs to be orchestrated, script-centric technologies, which must be wrapped, and data formats, which are required to be handled [BBK+13; EEKS11]. Second, plans are tightly coupled to a certain application topology and sensitive to structural changes. That is, different combinations of components lead to different plans. New plans have to be created for new application. The declarative approach solves this problem as plans are not needed, which eases and speeds up the development of TOSCA applications. However, TOSCA runtime environments are required to understand the components or at least the management operations they provide to infer provisioning logic. This introduces two drawbacks. First, provisioning capability is limited to common types of components and pre-defined operations that are known and orchestrated by the runtime. Second, application developers are not able to define complex custom provisioning

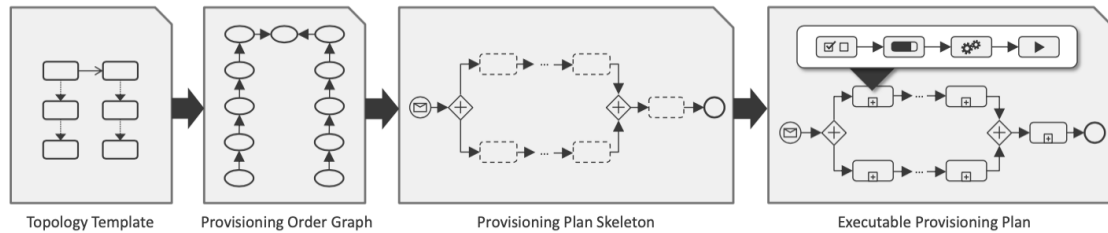


Figure 3.1: Steps of Plan Generation in [BBK+14]

logic that may be needed for the provisioning of complex applications. The approach enables to benefit from strengths of both flavors that leads to economical advantages for TOSCA application development.

Breitenbucher et al. [BBK+14] addresses the drawbacks by presenting an approach that interprets TOSCA Topology Templates for generating executable Provisioning Plans implemented in general-purpose workflow languages. The high-level overview on the plan generation approach is shown in Figure 3.1. The plan generation is divided into three steps. First, the CSAR to be provisioned serves as input for the generation of a Provisioning Order Graph (POG). Second, POG is translated into a Provisioning Plan Skeleton. Third, Provisioning Plan Skeleton is completed to an Executable Provisioning Plan.

In the first step, the Topology Template contained in the input CSAR is analyzed and transformed into a workflow language-independent POG. POG defines the order in which the topology's nodes and relations have to be provisioned. Each vertex in the POG represents the task to provision a certain Node or Relationship Template. The directed edges between two vertices define the temporal provisioning order: the vertex at the source of the edge must be processed, i. e., provisioned, before the vertex at the edge's target. This definition is based on [Mie10] but extended to support the explicit provisioning of relationships, too.

In the second step, the POG is used to generate a Provisioning Plan Skeleton. This skeleton is implemented in a certain workflow language, but defines only the structure of the final Provisioning Plan. Thus, the skeleton is not executable and contains only Empty Provisioning Activities for each Node and Relationship Template to be provisioned. These activities are placeholders for the actual provisioning logic. Therefore, vertices in the POG get transformed into Empty Provisioning Activities, POG edges into control constructs between the respective activities. Thus, the partial order of provisioning steps defined by the POG is retained.

In the last step, the Provisioning Plan Skeleton gets completed to an Executable Provisioning Plan, which is a fully automatically executable workflow. Therefore, each Empty Provisioning Activity is replaced by one or multiple Provisioning Subprocess Templates that implement the actual logic to provision a certain Node or Relationship Template based on its type. The resulting plan may be adapted by the application developer afterwards to customize the provisioning.

The concepts presented in [BBK+14] are used as the foundation of Plan Generator component of OpenTOSCA. The high-level concepts and components in Plan Generator are adapted for generating BPMN Build Plans in this thesis.

3.3 Automated Cloud Service Provisioning Using BPMN

Calcaterra et al. [CCMT17] proposed a framework for combining TOSCA and BPMN to enable automated cloud service provisioning. The novelty of their approach is a clear separation between the orchestration of the provisioning tasks and the provisioning services. They introduced a converter which takes a TOSCA template as input and produces a workflow that is ready to be executed by a workflow engine. The BPMN notation was used to represent both the workflow and the data that enrich the workflow. To support the viability of the proposed idea, a use case was developed and discussed in the paper.

This work addresses the design and implementation of a software framework that aims at easing and automating the processes that support the operational management of cloud services. They propose a framework where the provisioning services may be supplied by third party service providers, while the provisioning tasks orchestrated by the workflow engine will draw on those services in a SOA (Service Oriented Architecture) fashion.

The orchestration of the provisioning tasks are intended as the scheduling of the logical steps to be taken, and the provisioning services are the services implementing the tasks' instructions. As for the orchestration aspect, they devised a mechanism that automatically builds a plain BPMN orchestration plan starting from a cloud application's TOSCA model. BPMN is selected for its robust standard and support for data modeling. They developed an Orchestrator, ad-hoc YAML-to-BPMN converter, which transforms TOSCA Simple Profile in the YAML language to executable BPMN.

The Orchestrator consists of three components: TOSCA-Parser, BPMN-Generator, and BPMN-Validator generates BPMN plan in three stages. The TOSCA-Parser deals with the Service Template by providing means to load, parse and validate the YAML file, and creates the dependency graph, a data structure containing the relationships between all of the nodes in the TOSCA template. Vertices in the graph represent Nodes, while edges represent relationships occurring between them. The BPMN-Generator grounds the creation of the Provisioning Plan on the parsed Service Template and the dependency graph. The BPMN-Validator validates the automatically generated Plan against the BPMN specification.

In the first stage, TOSCA-Parser parses `ToscaTemplate` and `ToscaGraph` from TOSCA YAML template. `ToscaTemplate` is a parser class consists of `Topology Template`, `Node Template` and `Relationship Template`. `ToscaGraph` keeps track of all nodes and dependency relationships between them in the TOSCA template.

In the second stage, BPMN-Generator takes the `ToscaGraph` and `ToscaTemplate` generates the BPMN Provisioning Plan. BPMN generation is composed of the two steps BPMN-Generator first generates a Workflow modeling a detailed sequence of business activities to perform Second, BPMN-Generator generates a Dataflow modeling the data to be read, written or updated during the Workflow execution. The generated workflow basically comprises a BPMN process made of Service Tasks, Sequence Flows and Gateways.

Service Tasks are derived from node templates. Service Tasks dependencies are then obtained by taking into account the node requirements and the lifecycle operations they represent by traversing `ToscaGraph`. Service Tasks with the lowest execution order are called Service Tasks Endpoint; tasks with the highest execution order are called Service Tasks Startpoint. The execution order of all Service Tasks are computed by the dependency. The example of data structures are shown in

Listing 3.1 Data Structure of Tasks, Requirements, and Order [CCMT17]

```
service_tasks = ['server', 'sw_create', 'sw_configure', 'sw_start']
service_tasks_requirements = {'sw_create': ['server'], 'sw_configure': ['sw_create'],
                              'sw_start': ['sw_configure']}
service_tasks_order = {'server': 4, 'sw_create': 3, 'sw_configure': 2, 'sw_start': 1}
service_tasks_startpoint = ['server']
service_tasks_endpoint = ['sw_start']
```

Listing 3.1. Endpoint and Startpoint play a role in the creation of Start Event, End Event and Service Tasks. Service Tasks and related Sequence Flows are created by proceeding in ascending Service Tasks priority fashion (i.e., in their reverse execution order). From lowest to highest priority, each Service Task is created and then their incoming and outgoing paths are determined by distinguishing three further cases: a) the Service Task belongs to Service Tasks Endpoint set, b) the Service Task belongs to Service Tasks Startpoint set, c) the Service Task belongs to neither of them. Data Inputs, Data Outputs and Data Objects are derived from node templates and their data requirements in the YAML Service Template. Data Objects are utilized to model data requirements in the YAML Service Template with Data Associations determining how information stored in Data Objects is handled and passed between Process flow elements. This leads to a Data Input and a Data Association between the Start Event and the server Service Task. In the last stage, the BPMN-Validator validates the BPMN Plan generated in the previous step against the BPMN XML Schema.

The concepts of BPMN-Generator and Service Task presented in [CCMT17] provides an insight for generating provisioning plan in BPMN 2.0 from TOSCA Service Template.

4 Concept and Specification

This chapter describes the high-level concepts behind the thesis for the generation of the Build Plan, which is the main contribution of this paper. The general architecture of the approach is outlined and the building blocks are discussed in detail in the following subsections.

This work proposes an approach to generate Build Plans in BPMN 2.0 for Service Templates in a given CSAR. In OpenTOSCA, the Build Plan refers to provisioning plan. A provisioning plan is a workflow that can be executed fully automatically to provision the Service Template of a CSAR with all its Node Templates and Relationship Templates. Thus, running a provisioning plan instantiates a new application instance. provisioning plans are, therefore, used to implement higher-level provisioning logic for entire applications by orchestrating lower-level operations that provision single Node Templates or Relationship Templates. For simplicity, the provisioning plan and the software component for generating Plan in BPMN are referred to as Build Plan and BPMN Plan Generator in this work.

4.1 Use Case

Before the plan generation is presented in detail, the overall use case is introduced. An application developer develops a CSAR containing the Topology Template of the application to be provisioned and all required types, artifacts, and files but without Build Plan. The developer employs the presented Plan Generator to generate a Build Plan fully automatically. After the generation, the developer may adapt the generated plan for individual needs or use the plan as it is. The generated Build Plan is added to the Service Template as Plan Element and exported as CSAR. The final CSAR is then sent to a TOSCA-enabled Cloud Provider hosting a TOSCA runtime environment. The provider is responsible for installing the CSAR and maintaining the environment infrastructure. To instantiate application instances, providers typically offer Web-based APIs or self-service portals to start provisioning plans. Plans and IAs are deployed by the runtime fully automatically.

4.2 Requirement Specifications

The purpose of the BPMN Plan Generator is to generate BPMN Build Plan for a given Service Template. The generated Build Plan should be an executable BPMN plan in XML format and be able to be deployed to BPMN-compliant engines. The generated BPMN Build Plan should be an executable BPMN mentioned in Section 2.3.6. The executability of Build Plan enables the automated provisioning of cloud services. The BPMN Plan Generator should be integrated into OpenTOSCA Container. Therefore, Camunda Engine is the target Plan Engine for deployment.

The intended audience for using BPMN Plan Generator is application developers. The user only needs to provide a declarative provision topology template with the specified input and output. Then the imperative provision plan is automatically generated by BPMN Plan Generator.

4.3 BPMN Plan Execution Context

BPMN Script Task is chosen to be the core of activity for the generated BPMN Plan. The reason to use Script Task is that Task Script is defined to be an automated activity and fits into the case of automatic provisioning. BPMN defines a task to be an atomic activity that is included within a process. A Task is used when the work in the Process is not broken down to a finer level of Process Model detail. Generally, an end-user and/or an application is used to perform the Task when it is executed. Script Task is a kind of task that is executed by a business process engine. The modeler or implementer defines a script in a language that the engine understands how to execute. When the Task is ready to start, the engine will execute the script. When the script is completed, the Task will also be completed. The script of a Script Task could be inlined as part of an element in the Script Task or referenced as an external source. Kopp et al. [KBBL12] proposes the use of script and BOWIE, integrated as part of OpenTOSCA toolchain, implements the proposal with Groovy script. Groovy is a powerful, optionally typed and dynamic language, with static-typing and static compilation capabilities, for the Java platform aimed at improving developer productivity thanks to a concise, familiar, and easy-to-learn syntax. It integrates smoothly with any Java program and immediately delivers to your application powerful features, including scripting capabilities, domain-specific language authoring, runtime, compile-time meta-programming, and functional programming. The OpenTOSCA ingrates Camunda Engine as the BPMN Engine. Camunda Platform works with most of the JSR-223 compatible script engine implementations and has been tested for integration for Groovy. As a part of OpenTOSCA ecosystem, this work develops Groovy scripts based on the concepts from BOWIE and BPMN4TOSCA. Therefore, the plan execution context highly depends on the Groovy Script execution on Camunda Platform. For a successful plan deployment and execution, careful variable handling is required.

There are two main types of operations to be invoked from Script for orchestrating applications in OpenTOSCA. The first type of operation is responsible for making an HTTP request to OpenTOSCA Instance API for managing the state and properties of the instance of Service Template, Node Template, Relationship Template, and Plan. As shown in Figure 4.1, by executing a Create Service Template Script Task, a script is executed to make an HTTP POST request to the REST API endpoint of OpenTOSCA Container¹. The POST request to Endpoint *DataInstanceUrl* will create an instance of Service Template. The response of the HTTP POST request is a URL of the created instance. This URL should be stored as a variable of Camunda Execution Context for later Script Task. The second type of operation is to make an HTTP request to Management Bus End Point to invoke operations on IA Engine for application. The following lists the operations required for basic orchestration in OpenTOSCA:

- Create Service Template Instance
- Create Node Template Instance

¹<https://github.com/OpenTOSCA/container/tree/master/docs/container-api>

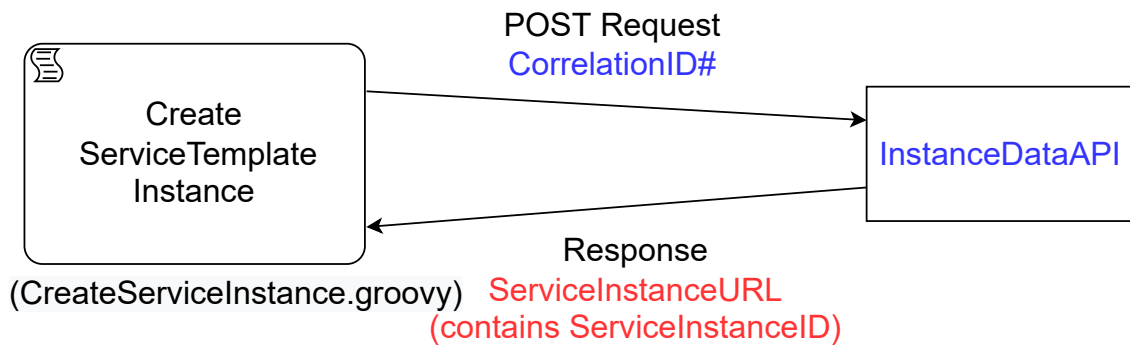


Figure 4.1: Example of Script Task Request Response with OpenTOSCA Container API

- Create Relationship Template Instance
- Set Instance Property
- Set Instance State
- Call Node Template Operation

The operations are designed to be as orthogonal as possible to reduce the complexity of parameter handling for a Script Task. Each Script Task has its functionality and thus has different execution semantics, which required specific input/output parameters. In a sense, each Script Task can be viewed as a method and each method has a different signature and return value. When Camunda Engine executes a Script Task, it is as if the engine is calling different methods consecutively in the same execution environment.

4.4 BPMN Build Plan Generation

This section introduces the concept of Build Plan generation. As mentioned in the previous section, the Script Task is central to the executable BPMN plan. Therefore, three aspects need to be addressed for an executable BPMN plan generation: (i) execution order of Script Task, (ii) type of script for Script Task, (iii) and input/output parameter and variable handling for Groovy script. The execution order of Script Task needs to follow the Topology Template and at the same time preserve the logical order of the plan. The type of script for Script Task needs to be associated correctly with the task based on Node/Relationship Template. The input/output parameter for Groovy Script needs to match the type of script. A consistent variable handling ensures the successful execution of the Script Tasks. Based on these three aspects the Build Plan generation involves three stages:

- **BPMN Plan Skeleton Generation:** the Plan Generator generates a skeleton for all the required activities in BPMN process based on the topology.
- **BPMN Subprocess Completion:** the Plan Generator completes the detail of each activity within the Subprocess with respect to the properties of each corresponding Node/Relationship Templates.

- **BPMN Process and Diagram Finalization:** the Plan Generator finalizes the BPMN Build Plan by transforming each activity into BPMN process elements in XML and generating the BPMN diagram elements based on the process elements.

4.4.1 BPMN Plan Skeleton Generation

Skeleton generation is the first stage of Plan Generation. The Plan Generator generates a skeleton for all the required activities in BPMN process based on the POG described in Section 3.2. The skeleton consists of placeholder BPMN activities in a specific order based on the Topology Template and the type of plan to be generated. The BPMN activities in the generated skeleton are completed in later stage regarding the pattern, input/output parameter handling, interfaces/operation of Node Template, and Relationship Template.

There are two reasons for generating the whole BPMN skeleton at first, rather than creating each activity while iterating through each of the Node Template or Relationship Template. The first reason is that it is desired to preserve the logical order of provisioning in the generated plan, instead of following the same order of POG. In this way, the generated plan can easily be understood by the application developer and be modified according to the developer's requirements. Therefore, the end result should be both correct and logical. A case in point, Figure 4.2 below lists a simple scenario where a Docker Container Node Template is hosted on a Docker Engine Node Template. The logical provisioning order should be the first Subprocess for creating Docker Engine Node Template, the second Subprocess for Docker Container Node Template, and the last Activity for creating HostOn Relationship Template because the HostOn can only be established after the Docker Container is started. Therefore, when the Plan Generator starts with any Node Template, it must iterate until the second Node Template to create the Activity/Subprocess. The second reason is that BPMN Plan Generator uses a pattern-based approach to determine the required operation for different patterns of Node Template. This approach will be discussed in detail in Section 4.4.3.

The Plan Generator employs an algorithm adapted from Breadth-First Search (BFS) to iterate through the Topology Template. The algorithm is shown in Algorithm 4.1. The algorithm, in its current form, supports a single-source, single-sink, and single-path topology as topologies with multiple sources, sinks, and paths require delicate adaptations for gateways. The algorithm first builds a graph from all the activities in POG and then starts traversing the activities which are identified as sources. For a POG, there could be multiple sources and sinks. The algorithm traverses POG from one Node Template to the other Node Template. In between the Node Template lies the Relationship Template. Then, the algorithm creates the corresponding BPMN process element in the order of host Node Template, target Node Template, and Relationship Template. For each Node Template, a Subprocess is created. This Subprocess is the entity for the later completion stage. For each Relationship Template, a Script Task is created. The reason that a Subprocess is generated for Node Template instead of a single activity is that the targeted operation based on Node Template pattern with respect to Relationship Template is categorized as Node Template Activity. This categorization makes the execution semantics simpler by separating all the operations from Relationship Template. Script Task for Relationship Template only needs to handle instance creation and state setting, which can easily be combined into a single script. On the other hand, there are several Tasks for a Node Template. For example, a provisioning Subprocess includes tasks for creating a Node Template instance, calling the operation based on the pattern, and setting the

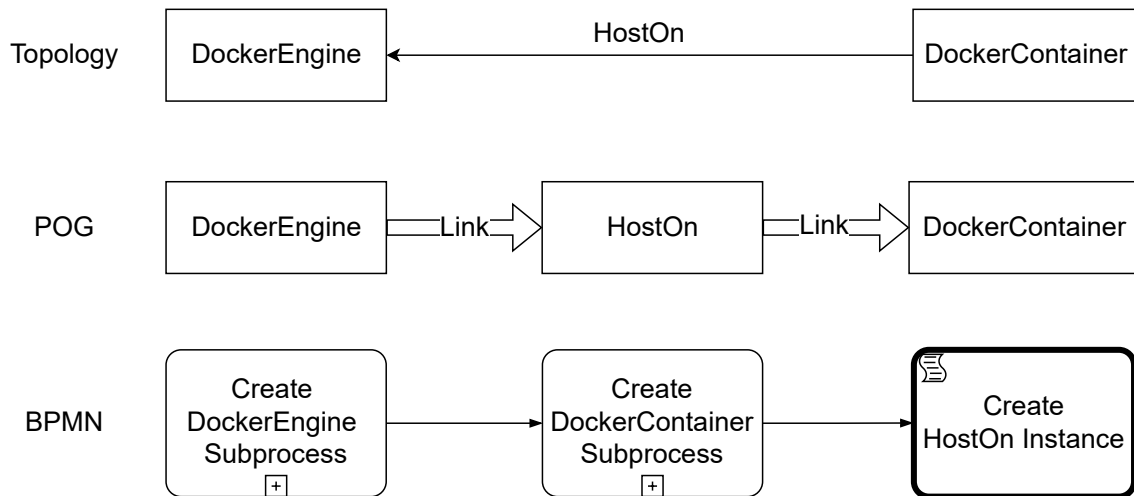


Figure 4.2: Logical Order Example

properties. The example is shown in Figure 4.3. After the Skeleton is generated, the Plan Generator will no longer require to traverse the POG. The detail of each Task inside Subprocess is left for the Plugin to complete, which is the second stage of plan generation.

4.4.2 BPMN Subprocess Completion

This section introduces the concept of Subprocess completion. Subprocess completion is the second stage of Plan Generation. The purpose of Subprocess completion is to complete the Script Tasks inside Subprocess with coherent variable naming and assignment to ensure correct execution of every Script Task. The completion is achieved with different Plugins and Node Type Handlers according to the type of Script Task. The type of Script Task depends on the type of plan. For Build Plan, there are three types of Script Tasks in a Subprocess as shown in Figure 4.3:

- Create Node Instance Script Task: instantiates the Node Template in the Container so that user can monitor the state of instance.
- Call Node Operation Script Task: invokes the Interface Operation to the Management Bus based on the pattern of Node Template
- Set Node Properties Script Task: sets the properties of the Node Template instance so that user can later manage.

Each Script Task needs to be completed with coherent input/output parameters based on the type of Script Task. To provide flexibility for extending the new Script Task, the idea of the Script Task Plugin is introduced. This follows the principle of *Closed for Modification and Open for Extension*. Each plugin is only responsible for completing its type of Script Task. When a new Script Task is introduced in the future, the developer only needs to implement a new Plugin for the Script Task, rather than changing a monolithic completion component. As shown in Figure 4.4, three Script Task Plugins are required to complete Subprocess for Build Plan. Each Plugin may have different functionalities. Create Node Instance Plugin first handles the global variable associated with Node Template and creates an instance.

Algorithm 4.1 BPMN Skeleton Generation Algorithm

```
procedure CREATE_BPMN_PLAN_SKELETON( $Plan_{abstract}$ )
  let  $Q$  be a queue of tuple (abstract activity, BPMN element)
  let  $Task_{ST}$  be create service template script task
  let  $Task_{SS}$  be a set service template state scrip task
  let  $Event_{start}$  be start event
  let  $Even_{end}$  be end event
  LINK( $E_{start}, Task_{ST}$ )
  let  $A_{curr}$  be pointer of abstract activity in current iteration
  let  $B_{prev}$  be pointer of BPMN element from previous iteration
  for all  $Activity_{abstract}$  in  $Plan_{abstract}$  do
    if  $Activity_{abstract}$  is source then
       $Q$  enqueues new tuple ( $Activity_{abstract}, Task_{ST}$ )
    end if
  end for
  while  $Q$  is not empty do
     $size \leftarrow Q.size()$ 
    while  $0 \leq size$  do
       $Tuple_{curr} \leftarrow Q.poll()$ 
       $size \leftarrow size - 1$ 
       $A_{curr} \leftarrow Tuple_{curr}.first()$ 
       $B_{curr} \leftarrow Tuple_{curr}.second()$ 
       $List_{activity} \leftarrow COLLECT\_UNTIL\_NEXT\_NODETEMPLATE(A_{curr})$ 
      for all  $A_i$  in  $List_{activity}$  do
         $B_{prev} \leftarrow CREATE\_BPMN\_ELEMENT\_FROM\_ACTIVITY(A_i)$ 
        label  $A_i$  as visited
        LINK( $B_{prev}, B_{curr}$ )
         $B_{prev} \leftarrow B_{curr}$ 
      end for
       $Tuple_{next} \leftarrow (A_{curr}, B_{curr})$ 
       $Q.enqueue(Tuple_{next})$ 
    end while
  end while
   $B_{last} \leftarrow B_{prev}$ 
  LINK( $B_{last}, Task_{SS}$ )
  LINK( $Task_{SS}, Event_{end}$ )
end procedure
```

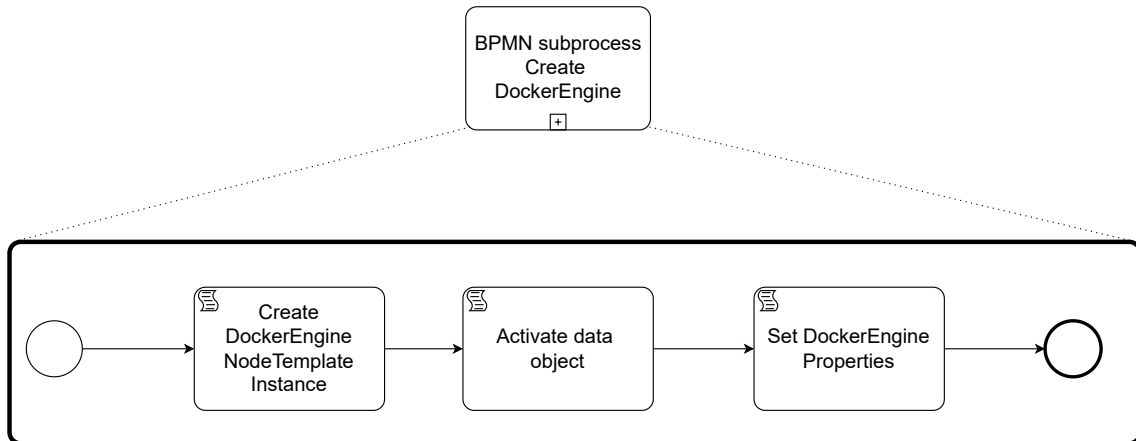


Figure 4.3: Node Template Subprocess Example

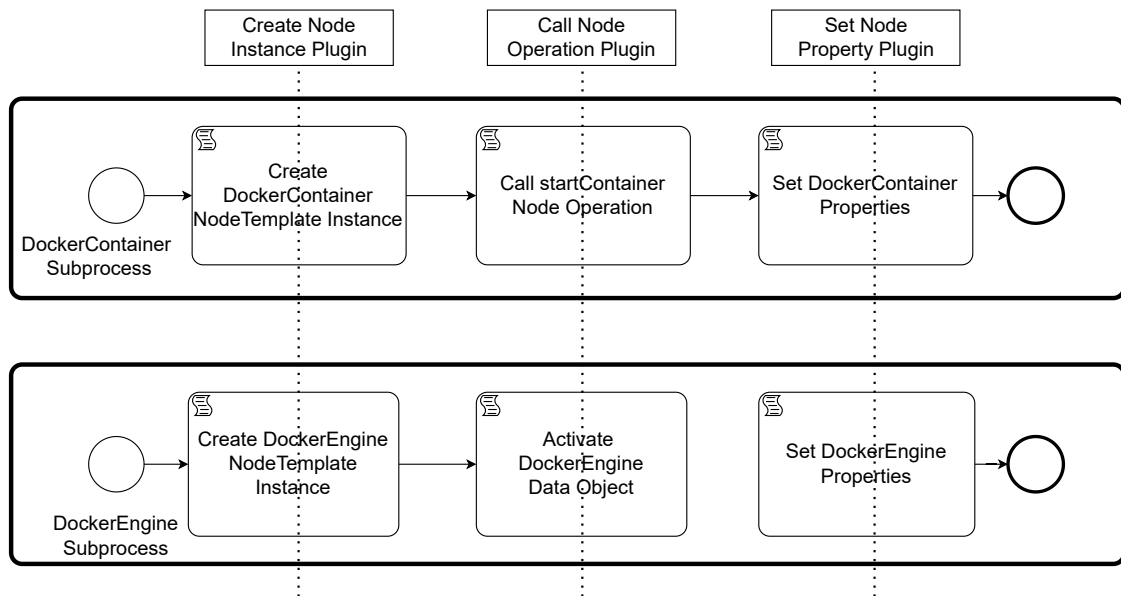


Figure 4.4: Node Template Type Plugin

Call Node Operation Plugin identifies the Node Template pattern for operation and prepares the parameters for invoking Management Bus. Set Node Property Plugin handles the input parameters for setting Node Template instance properties. One of the most common functions of Plugin is to handle parameters from Node Template Property.

Node Template Property Parameter Handling

As mentioned, handling parameters for Script Task according to Node Template Property is the most common function of Plugin. Deciding the right granularity for handling Properties and Parameters is critical for plan execution and is the central contribution of this work. Before discussing the topic in detail, below are the definitions to make the later explanation clear.

- Property: refers to the properties of Node Type defined in TOSCA
- Parameter: refers to the input and output parameters for Script Task
- Variable: refers to the variable which is declared, defined, or assigned in Plan Engine Runtime
- Plan Generation Time: refers to the temporal interval during which a plan is being generated
- Plan Execution Time: refers to the temporal interval during which a plan is being executed.

For a given Node Template in a Service Template, the property name is defined by the Node Type of Node Template while the property value of the could either be assigned with a fixed value or with user input during instantiating. Property with fixed value is known by the Plan Generator before Plan Generation Time. Property with user input is only known to Plan Engine before Plan Execution Time.

Input and Output parameters derived from the properties of Node Template need to accommodate the consistency for property for both fixed value and user input. Therefore, a variable is used as place holder for user input property by Plan Generator. The variable name is generated by Plan Generator during Plan Generation Time while the variable values are assigned via Plan Input Parameter at Plan Execution Time.

To facilitate the user-input property parsing, a preserved prefix String is used as a part of the property value in the Node Template. The Plan Generator parses out the prefix and declares the property value as a plan input parameter.

Figure 4.5 shows an example for Node Template Property Parameter Handling. In the Topology Template of Figure 4.5, Docker Container is hosted on the Docker Engine. Docker Engine has a property *DockerEngineURL* associated with value from user input *get_input: DockerEngineUrl*. Docker Container has a property of *ContainerPort* associated with value from user input *get_input: ApplicationPort* while other properties are assigned with fixed value. The user input is parsed from Node Template property and declared as global variable with name *DockerEngineUrl* and *ApplicationPort*. These variables are accessible through the Plan Execution that arrives in the Script Task and can be used within the script. During the completion of Script Task, the input parameters for the Script Task is associated with variable. In the case of Call Node Operation Script Task, one of the inputs for invoking *StartContainer* operation in IA is *ContainersPorts*, which is the port mapping for the container. The user-input property *ContainerPort* is combined with fixed-value *Port* to form *ContainersPorts*. The parameter value of *ContainersPorts* is assigned with $\${ApplicationPort}$, 80 in Build Plan. If the user is assign *ApplicationPort* with value of 9999, then at Plan Execution Time, the value of *ContainersPorts* is rendered as 9999, 80. The above-mentioned example shows how the introduction of variable guarantees the parameter-property-consistency.

4.4.3 Pattern-Based Activity Handle

Apart from the common function of Plugin, each Plugin has its functionality for completing a specific Script Type. Call Node Operation Plugin is responsible for completing Call Node Operation Script Task. The main function of the task is to invoke the Interface Operation to the Management Bus based on the pattern of Node Template. Call Node Operation Plugin uses a pattern-based approach to decide which operation of associated Interface in a Node Template to invoke. The reason for using a pattern-based approach is that the relationship between Node Template follows the

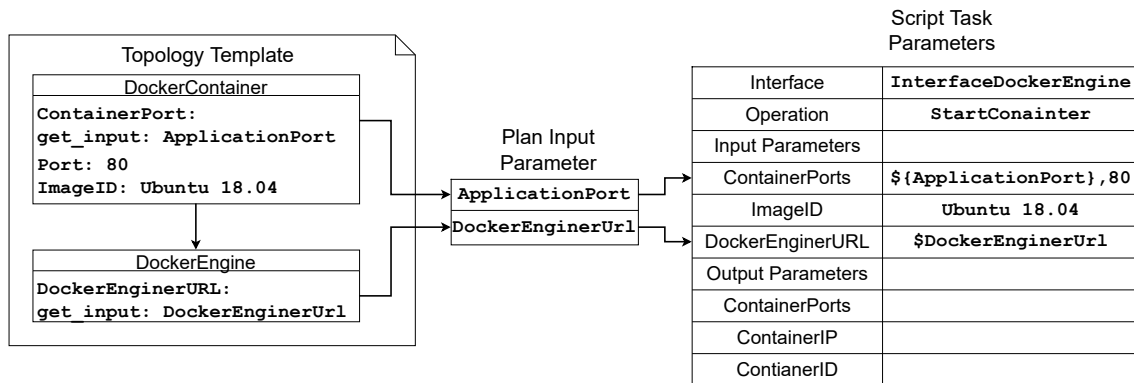


Figure 4.5: Example of Node Template Property Parameter Handling

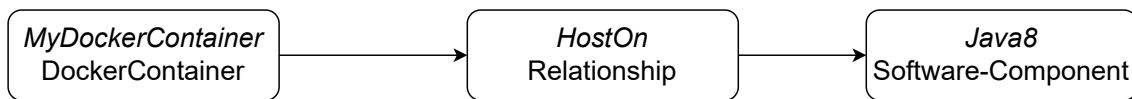


Figure 4.6: Example of Docker Container Pattern

pattern decided by the Node Type. For example, a Node Template of Docker Container Node Type must be hosted on a Node Template of Docker Engine Node Type. A Node Template of Software Component Node Type must be installed on either a Node Template of Docker Container Node Type or Virtual Machine Node Template. These recurring relationships fit the use case for a pattern-based approach. The Cloud Computing Patterns of Fehling et al. [FLR+14] describes a problem that frequently occurs in a certain context and provides a proven solution for that problem describing abstract solutions to recurring problems in the domain of Cloud Computing to capture timeless knowledge that is independent of concrete providers, products, programming languages, etc. The pattern-Based approach allows the design of Activity Handling to be flexible. Whenever a new pattern occurs with a new Node Type, a new pattern handler can be implemented without affecting the existing pattern. In this work, three types of patterns are proposed for generating provision plans: Docker Engine Pattern, Container Engine Pattern, and Life Cycle Pattern.

Docker Engine Pattern

Docker Engine Pattern is identified when the Host Node Template is of *DockerEngine* Node Type and the Target Node Template is of *DockerContainer* Node Type. The reason that this is identified as a pattern is that the input for the operation is required to be derived from properties of both Node Template. Figure 4.7 shows an example of the pattern. The operation for this pattern of the Interface *InterfaceDockerEngine* in Docker Engine.

From the example, to invoke *startContainer* operation, input parameters such as *ContainerImage*, *ContainerPorts* are properties of Docker Container while *DockerEngineURL* and *DockerEngineCertificate* are from Docker Engine. Another important reason is that the response of the operation on Docker Engine may be used for setting properties of Docker Container. From the example, *ContainerID* is one of the output for *startContainer* and is used for setting same property *ContainerID* of Docker Container.

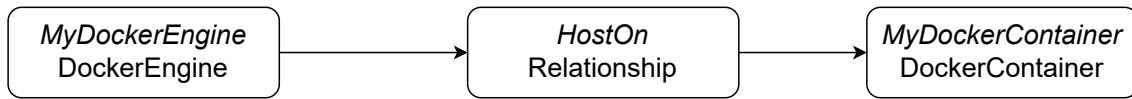


Figure 4.7: Example of Docker Engine Pattern

Container Engine Pattern

Docker Container Pattern is identified when the Host Node Template is of *DockerContainer* Node Type. Figure 4.6 shows an example of the pattern. The operation for this pattern of the Interface *ContainerManagementInterface* in *DockerContainer* Node Type. The operation could either be *runScript* or *transferFile* depending on the relationship Template. The reason that this is identified as a pattern is that the input for the operation is required to be derived from properties of both Node Template. From the example, to invoke *runScript* for installing Java8 JVM on host *Docker Container*, input parameters *ContainerID* is from properties of Docker Container while the source of *Script* are from the IA of Java8.

Life Cycle Pattern

Life Cycle Pattern is identified when the Operation of Target Node Template is performed on the Node Template itself. The operation for this pattern of the Interface <http://www.example.com/interfaces/lifecycle>. It is worth mentioning that having the *Life Cycle* interface doesn't guarantee the pattern to be Life Cycle. The pattern still depends on the relationship between Node Template. For example, when a Docker Engine Node Template with running state is the source in the Topology Template, it means that is not hosted by no other Node Template. Therefore, no operation is required. So the Call Node Operation Task will fall back to setting the property of Node Template as a global variable.

4.4.4 BPMN Diagram Generation

One of the biggest characteristics of BPMN 2.0 is that it incorporates visual components in its metamodel. Generating a visual component for the BPMN Build Plan is a key contribution to this work. BPMN specification doesn't enforce correct 1-to-1 mapping between BPMN process elements and BPMN diagram elements. However, the correct rendering of diagram elements helps the application developer understand the execution of generated plan. The diagram elements are generated after the completion of all process elements, which include Subprocess and Script Task within. In this way, the diagram generation is guaranteed to be consistent with all process elements. The algorithm for diagram generation is adapted from BFS by traversing the process elements from Start Event. The pseudocode for the algorithm is shown in Algorithm 4.2.

The algorithm traverses the Process elements by one element at a time, starting from the Start Event of the Process. By keeping track of the position from the last traverse elements, the new element position is calculated based on the type of elements. The length and width depend on the type of elements, which is predefined in implementation. Since the plan diagram is a layout on in 2D plane, the ways to describe the position for the diagram depends on the type of diagram. For example, Sequence Flow is a line, and its position is defined by two waypoints. On the other hand, Scrip Task

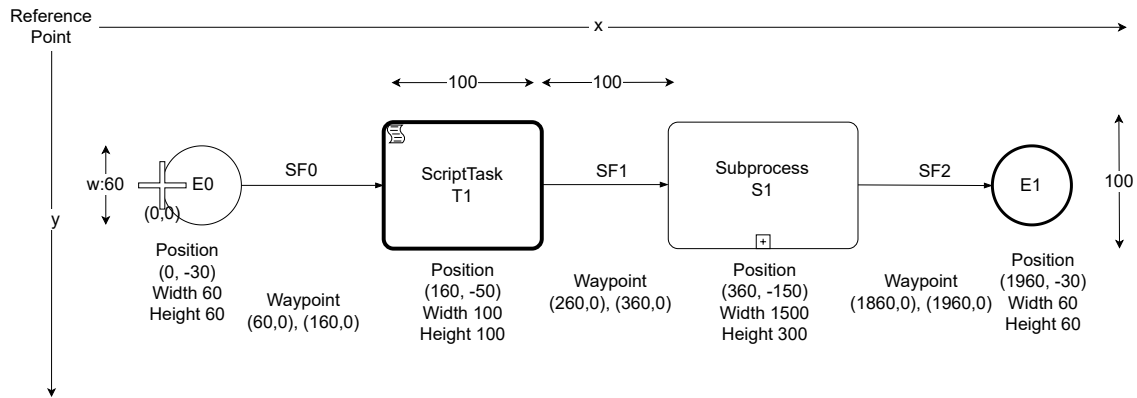


Figure 4.8: Example of Diagram Position in BPMN Plan

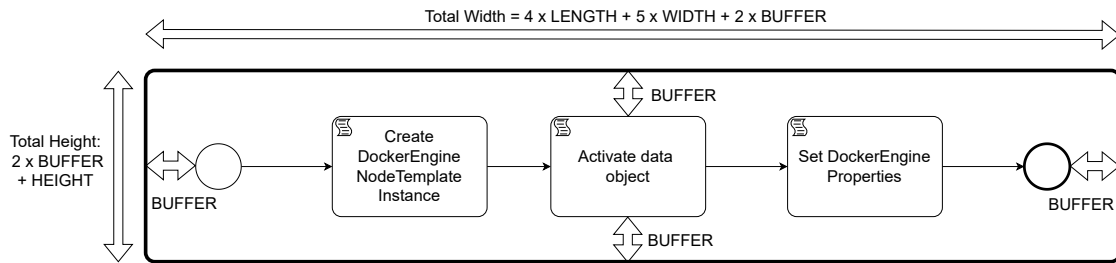


Figure 4.9: Example of Diagram Position in BPMN Subprocess

has a rectangular shape, and its position is defined by a reference point in the x and y direction, and the length and width of the shape. After traversing each element, the position will be stored as a diagram object associated with process element. All diagram objects will be generated into elements of a single XML DOM document in the Finalization stage.

Figure 4.8 shows a example of detailed position when generating. In the Figure 4.8, the start position is set to (0, 0) on the left side of Start Event. The length and width of Start Event *E0* are set to both 60 units. The position for reference Start Event is (0, -30), which is the top left corner of the diagram. After generating the position for Start Event *E0*, the generation process points to the next process element, Sequence Flow, *SF0*, which is the outgoing link for Start Event with the waypoint (60, 0), which is the middle point of the right boundary of Start Event *E0*. For Sequence Flow *SF0*, the length is set to 100 unit. This makes its left waypoint (60, 0) and right waypoint (160, 0). And the steps for generating a diagram go on. One thing worth mentioning is that, when generating a diagram for Subprocess, this work uses a recursive approach.

The recursive approach treats generating Activities in Subprocess as a new Process and generates diagrams based on the previous waypoint before Subprocess. Figure 4.9 shows a example for generating a Subprocess. To prevent the diagram from overlapping, buffer space is preserved inside Subprocess diagram. The length of buffer space is taken into account as the edge of Subprocess. For readability, the Subprocess is set to expand by default.

Algorithm 4.2 BPMN Diagram Generation Algorithm

```
procedure CREATE_BPMN_DIAGRAM_FROM_START_EVENT( $E_{start}, P_{x,y}$ )
  let Q be a queue of tuple (BPMN element, Position)
   $Q.enqueue(E_{start}, P_{x,y})$ 
  while Q is not empty do
     $size \leftarrow Q.size()$ 
    while  $0 \leq size$  do
       $Tuple_{curr} \leftarrow Q.poll()$ 
       $size \leftarrow size - 1$ 
       $B_{curr} \leftarrow Tuple_{curr}.first()$ 
       $P_{curr} \leftarrow Tuple_{curr}.second()$ 
       $D_{curr} \leftarrow CREATE\_DIAGRAM\_ELEMENT(E_{curr}, P_{curr})$ 
      COLLECT( $D_{curr}$ )
      if  $E_{curr}$  is SUBPROCESS then
        CREATE_BPMN_DIAGRAM_FROM_START_EVENT( $E_{curr}.getStartEvent(), P_{curr}$ )
      end if
      for all  $B_{next}$  in  $B_{curr}.outgoingLinks()$  do
        if  $B_{next}$  is not visited then
          label  $B_{next}$  as visited
           $P_{next} \leftarrow GET\_NEXT\_POSITION(D_{curr})$ 
           $Q.enqueue(B_{next}, P_{next})$ 
        end if
      end for
    end while
  end while
end procedure
```

4.4.5 BPMN Plan Finalization

BPMN Plan Finalization is the last stage for BPMN plan generation after both process elements and diagram elements are generated. The Plan finalization transforms the process elements and diagram elements from placeholder objects into XML elements in a Document Object Model (DOM) document with the DOM parser. To simplify the transformation of objects into XML, this work uses a Template String approach. This approach avoids the complexity of designing an XML-serializable class for each Type of BPMN process. It simplifies the object-serialization into String manipulation and DOM element operation. This approach first replaces the placeholder keyword in a predefined String Templates by matching the XML snippet with the field in the placeholder object. Then, the modified string is transformed into a DOM node and the node is appended into the target plan DOM document in XML. Each Type of BPMN process and diagram has its corresponding XML snippet. Each snippet contains several placeholder strings to be replaced. Example of Create Node Template Script Task is shown in Listing 4.1 and Diagram elements in Listing 4.2 Placeholder such as *TaskIdToReplace*, *TaskNameToSet* will be replaced by the ID and name of associated Create Node Template Script Task object.

Listing 4.1 Example XML Snippet for Process Elements Finalization

```
<bpmn:scriptTask id="TaskIdToReplace" name="TaskNameToSet" scriptFormat="groovy"
camunda:resultVariable="ResultVariableToSet">
  <bpmn:extensionElements>
    <camunda:inputOutput>
      <camunda:inputParameter name="State">StateToSet</camunda:inputParameter>
      <camunda:inputParameter name="NodeTemplate">NodeTemplateToSet</
camunda:inputParameter>
    </camunda:inputOutput>
  </bpmn:extensionElements>
</bpmn:scriptTask>
```

Listing 4.2 Example XML Snippet for Diagram Elements Finalization

```
<bpmndi:BPMNShape id="Shape_IdToReplace" bpmnElement="Event_IdToReplace">
  <dc:Bounds x="X_PosToReplace" y="Y_PosToReplace" width="WidthToReplace" height="
HeightToReplace" />
</bpmndi:BPMNShape>
```

At the end of BPMN plan generation, the generated plan and Groovy Scripts are exported to the uploaded CSAR. The BPMN Build Plan is attached to the Service Template associated with *initiate* operation of *OpenTOSCA-Lifecycle-Interface* interface.

5 Design and Implementation

This section describes the design and implementation of the BPMN Plan Generator prototype based on OpenTOSCA developed in this master thesis. The low-level architecture is introduced in Section 5.1 and the implementation detail of the components are presented in Section 5.2.

5.1 Design Overview

In this section, the overall design of BPMN Plan Generator is presented. Section 5.1.1 introduces the developed prototype base on OpenTOSCA ecosystem. Section 5.1.2 presents the architecture of BPMN Plan Generator as well as all components.

5.1.1 Prototype

The prototype is developed using the OpenTOSCA ecosystem discussed in Section 2.2. *The BPMN Plan Builder, BPMN Fragment Library, BPMN Plan Plugin, BPMN Plan model, BPMN Facade* are developed and integrated with the existing ecosystem. The CSAR Model and Integration Layer are reused from the existing component from BPEL Plan Generator. A simple application topology was used for the prototype, as shown in Figure 5.1. The topology is designed with the Winery Topology Modeler, packed, and exported into a CSAR using the functionality provided by the tool. The CSAR is then imported into the OpenTOSCA Container with the help of the OpenTOSCA UI web interface. The Plan Generator component then imports the CSAR, parses the Service Template, and generates BPMN Plan Skeleton from POG based on Topology Template using BPMN Plan Model. The BPMN Plan Plugin completes the detail for each Subprocess related to Node Template. The BPMN Plan Builder then finalizes the complete plan with BPMN Fragment Library and adds it to the CSAR. The CSAR is repackaged and exported back into the OpenTOSCA Container. The Container is responsible for deploying the plans into the compatible plan engine, Camunda Engine, which is integrated within the runtime environment.

5.1.2 Architecture

The low-level architecture for the implementation of the prototype is depicted in Figure 5.2. The architecture reuses the architecture of BPEL Plan-Generator [Kép13] in OpenTOSCA Container. Four components of the prototype: BPMN Plan Builder, BPMN Fragment Library, BPMN Plan Plugin, BPMN Plan Model, and BPMN Facade are the major contribution of this work. The relevant components of the architecture are discussed briefly in the following section.

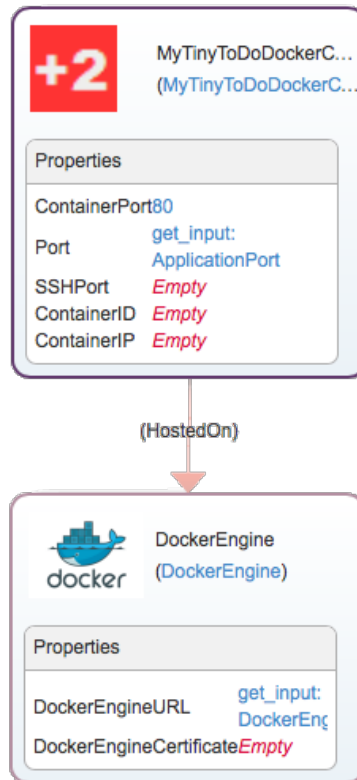


Figure 5.1: Sample Application Topology for Prototype

BPMN Plan Builder

The BPMN Plan Builder component is responsible for generating the plan skeleton, invoking the BPMN plugin for Subprocess completion, generating the BPMN diagram, and finalizing BPMN Plan. The finalization is achieved by requesting from the Template Fragment Library. This component implements the provisioning plan generation algorithm described in Section 4.4.

Template Fragment Library

Template Fragment Library is responsible for transforming the process elements and diagram elements from placeholder object into XML elements of the generated XML document. It is used by BPMN Plan Builder during the BPMN Plan Finalization step described in Section 4.4.5.

Plugins

The BPMN Plugins are responsible for completing the Subprocess in plan skeleton. Each Plugin is associated with the Script Task of Node Template as described in Section 4.4.2. Each Plugin is declared as Spring Bean to achieve loose-coupling for future extension.

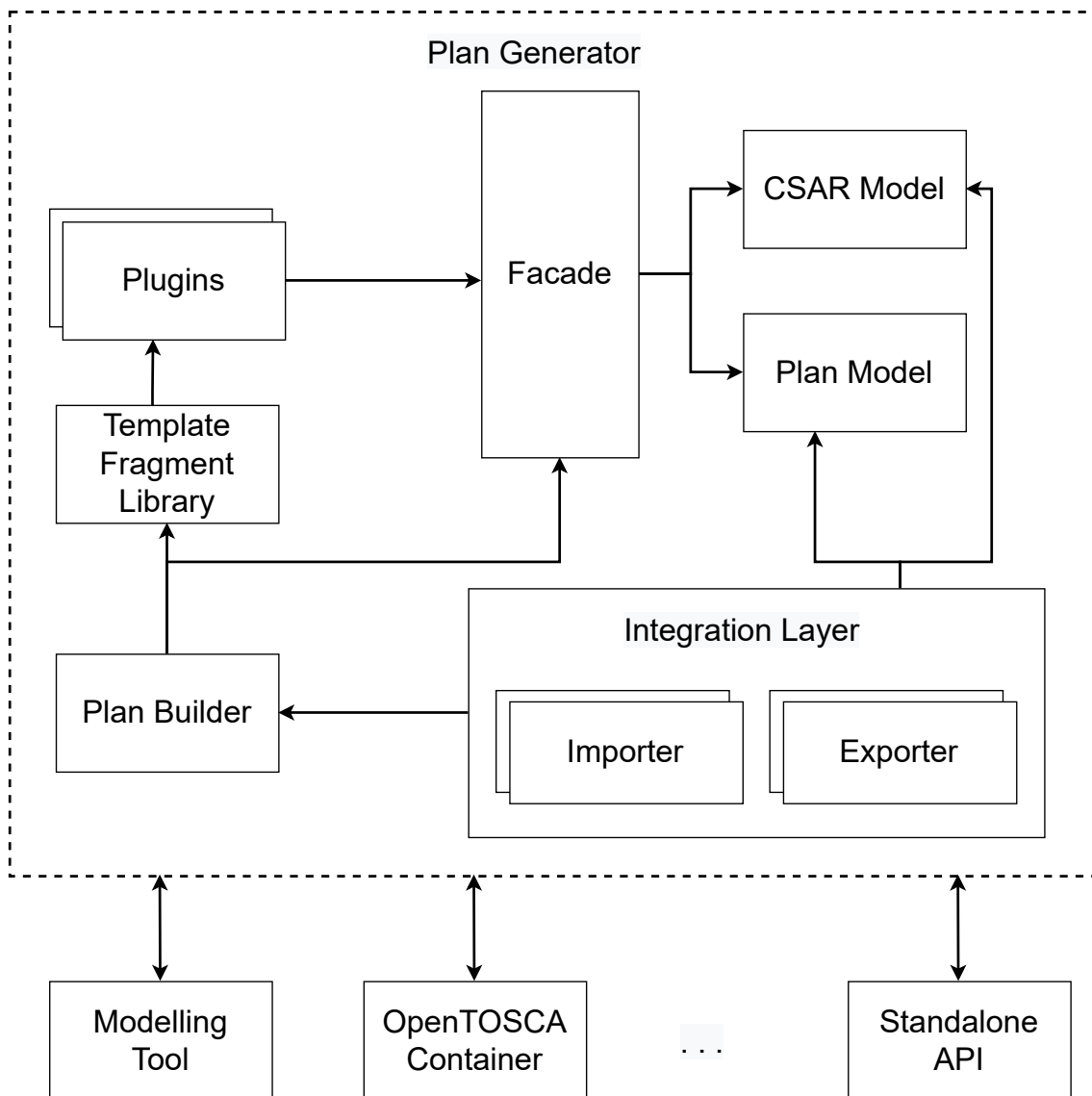


Figure 5.2: Plan Generator Architecture [Kép13]

Plan Model

The Plan Model [Kép13] corresponds to the model in which the BPMN Build Plan is implemented. The model contains all constructs of the implementation language since the facade encapsulates low-level operations on the model.

CSAR Model

The CSAR Model [Kép13] component corresponds to a whole CSAR. The model can be used by Plan Builder and Plugin to access all elements of a TOSCA definition. Imported elements (definitions, node types, etc.) are resolved within the entry definition. Plan elements can be added to the CSAR model.

Facade

The Facade [Kép13] 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.

Integration Layer

The Integration Layer [Kép13] 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.

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. Section 5.2.1 introduces the concrete BPMN Data Model used by BPMN Plan Generator programmatically. Section 5.2.2 describes the detail implementation of BPMN Plan Builder Section 5.2.4 describes the detail implementation of three BPMN Plugins. In the last section, Section 5.2.5 describes the unit testing implementation for the fundamental components in Plan Generator.

5.2.1 BPMN Data Model

Figure 5.3 shows the UML class diagram of BPMN data model. Three classes, *BPMNPlan*, *BPMNScope*, and *BPMNDiagram*, are the basic components constituting a BPMN Plan for the Plan Generator. The *BPMNPlan* is a class, used as the placeholder for all elements of a single BPMN plan and is the object that BPMN Plan Builder manipulated. The class contains fields related to TOSCA specification from a imported CSAR , objects regarding the generated BPMN Plan, and elements and documents for DOM document for BPMN plan export. The *BPMNPlan* extends abstract class *AbstractPlan*, which is the parent of *BPEL Plan*. This helps the *BPMNPlan* class to integrate with Integration Layer of OpenTOSCA and avoids the need to rewrite common methods that are already implemented in *AbstractPlan*.

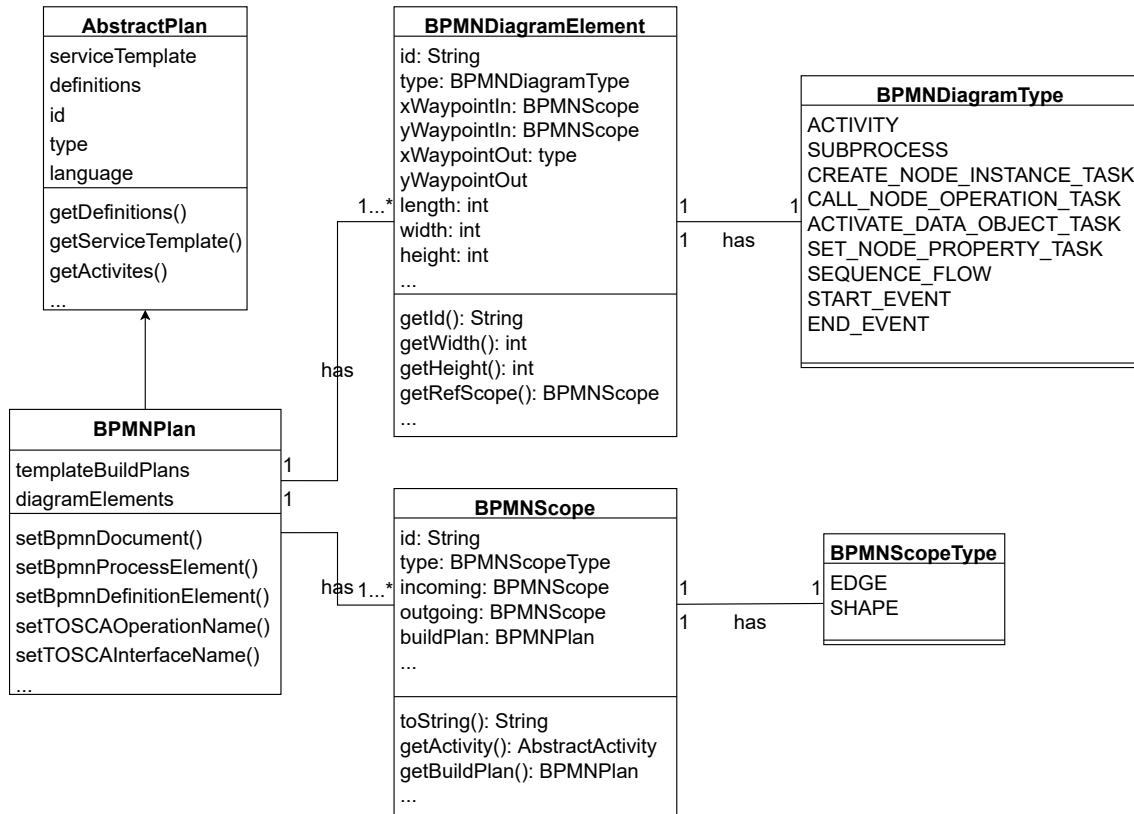


Figure 5.3: UML Class Diagram for BPMN Data Model

BPMNScope is a class, acting as place holder for all BPMN Process Elements. *BPMNScope* contains a field of *BPMNScopeType*, which specifies the type of process element the object is holding. *BPMNScopeType* is an enumeration class for the BPMN process element types, including Activity, Sequenced Flow, Event, Subprocess, Task ... etc. Each enumeration in *BPMNScopeType* has a 1-to-1 mapping of String Template from a predefined XML snippet of a process element, which would be used at the finalization stage.

BPMNDiagramElement is a class, acting as the placeholder for all BPMN diagram elements. The *BPMNDiagramElement* contains a field of *BPMNDiagramType*, which specifies the type of diagram element the object is holding. *BPMNDiagramType* is an enumeration class for the BPMN diagram element types, including Shape and Edge. Each enumeration in *BPMNDiagramType* has a 1-to-1 mapping of String Template from a predefined XML snippet of diagram element, which would be used at the finalization stage.

5.2.2 BPMN Plan Builder

Figure 5.4 shows the UML class diagram for BPMN Plan Builder. *BPMNBuildProcessBuilder* is the class for generating the high-level logic in BPMN Build Plan mentioned in Section 4.4. The *BPMNBuildProcessBuilder* extends abstract class *AbstractBuildPlanBuilder*, which is the parent class of *BPELPlanBuildProcessBuilder*. It invokes methods of *AbstractBuildPlanBuilder* to generate

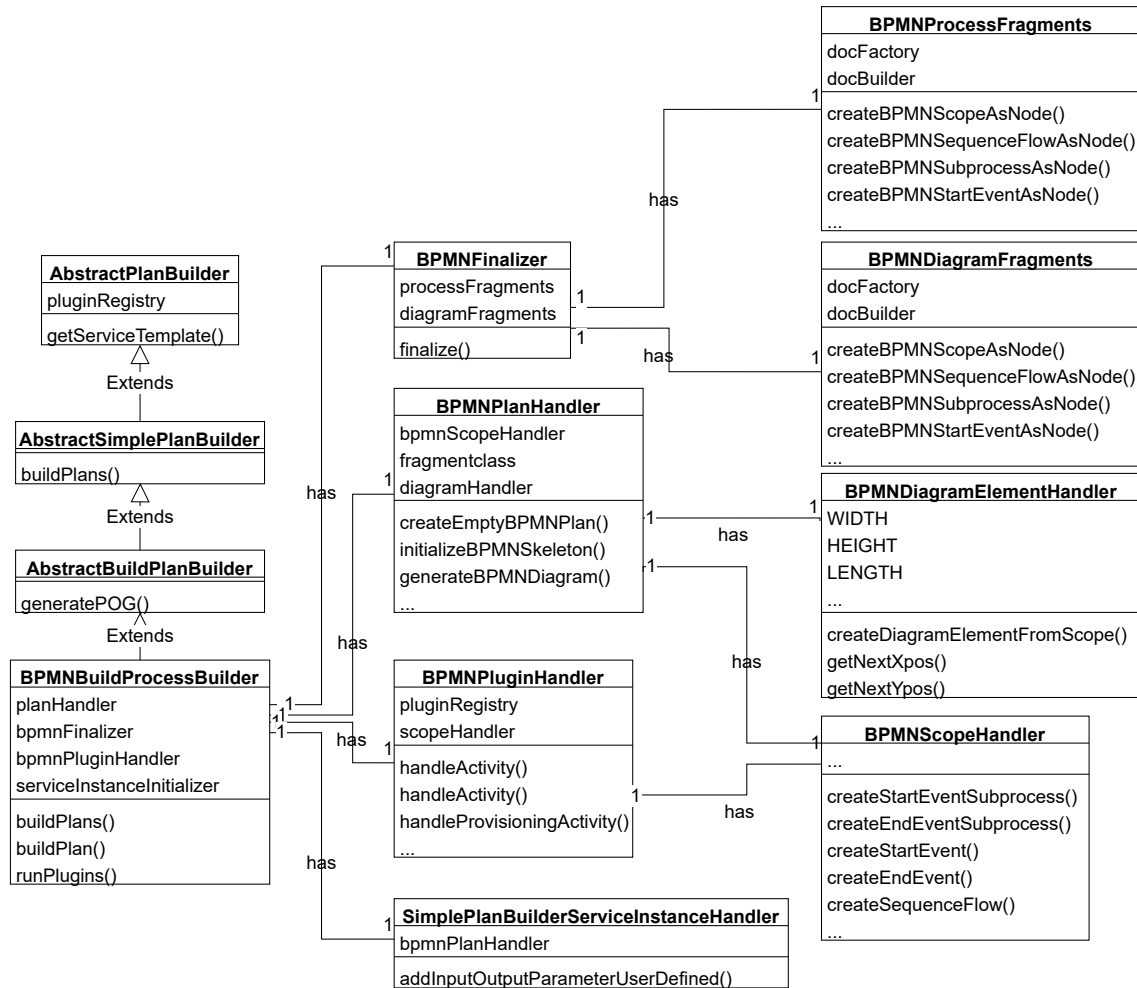


Figure 5.4: UML Class Diagram for BPMN Plan Builder

POG from the imported Service Template and definitions. *BPMNBuildProcessBuilder* instantiates *BPMNPlan*, generates skeleton for BPMN Plan, completes Script Task in Subprocess, generates BPMN Diagram elements, and finalizes the BPMN Plan. *BPMNBuildProcessBuilder* uses other handler classes to delegate tasks for achieving a more modular design.

BPMNPluginHandler class is implemented for invoking plugins for completing the Script Task in Subprocess. *BPMNPlanHandler* class is implemented for initializing basic elements in XML documents, attaching Groovy scripts for exporting, generates skeletons for BPMN plan and diagrams. *BPMNFinalizer* finalizes the plan generation process by generating XML elements from all the *BPMNScope* and *BPMNDiagramElement* contained in *BPMNPlan* and imports the elements to XML document. The handler classes do not directly interact with Data Model objects. They use several facade classes, which are discussed in the next section.

5.2.3 BPMN Facade

Four classes are implemented as the Facade to interact with BPMN Data Model classes: *BPMN-ScopeHandlers*, *BPMNDiagramElementHandler*, *BPMNDiagramFragments*, and *emphBPMN-ProcessFragments*. The Facade classes hide the complexities of Data Model classes and provide a simpler interface for the BPMN Plan Builder. *BPMNScopeHandlers* is responsible for instantiating and interacting with *BPMNScope* objects while *BPMNDiagramElementHandler* for *BPMNDiagram* objects. *BPMNProcessFragments* is used for transforming the *BPMNScope* object into a DOM node based on the type of object.

BPMNDiagramFragments is used for transforming *BPMNDiagramElement* objects into DOM nodes based on the type of object. Both *BPMNProcessFragments* and *BPMNDiagramFragments* uses predefined String Template mentioned in Section 4.4.5.

5.2.4 BPMN Plugin

As described in Section 4.4.3, three Task Script Plugins: *BPMNSetNodePropertyPlugin*, *BPMN-CallNodeOperationPlugin*, and *BPMNCreateNodeInstancePlugin* are implemented for handling the property of Node Template and input/output parameters of the corresponding Groovy Script.

The Plugins are registered in the Plugin Registry [Kép13]. This Plugin Registry is used for both BPEL and BPMN Plan Builder. Depending on the type of Script Task, the *BPMNPluginHandler* will invoke the matching Plugin. To increase the flexibility for implementing a new Plugin for the new Node Template, each type of Plugin has its delicate Interface. Three Plugin interfaces are declared: *IPlanBuilderTypeCreateInstancePlugin*, *IPlanBuilderTypeCallNodeOperationPlugin*, and *IPlanBuilderTypeSetPropertyPlugin*. Figure 5.5 shows the UML diagram of the Plugins and Plugins Registry.

BPMNCallNodeOperationPlugin is responsible for determining the Node Template Pattern using a pattern-based approach mentioned in Section 4.4.3. Again, to increase the molecularity of *BPMN-CallNodeOperationPlugin*, it uses different Pattern Handlers. Each Pattern Handler is responsible for handing a specific pattern. The pattern handlers extend *AbstractHandler* to inherit the common method for property and input processing. *DockerEngineHandler* and *EnginePatternHandler* are implemented. If a new pattern is introduced, the developer only needs to implement a new pattern handler. Thus, a pattern handler can be viewed as the plugin for *BPMNCallNodeOperationPlugin*.

5.2.5 Unit Testing

Testing is a critical part of good software development [Ola03]. Unit testing can be used to improve the quality and correctness of software. In addition to the functional classes, unit testing classes are implemented for Data Model and Facade in the course of development. The unit testing classes are meant to guarantee the consistency of interaction between Facade and Data Model. The implemented tests classes include *BPMNDiagramFragmentTests*, *BPMNProcessFragmentsTests*, *BPMNDiagramElementHandlerTests*, *BPMNPlanHandlerTests*, and *BPMNScopeHandlerTests*. Unit tests help to fix bugs early in the development cycle and save costs. It also helps the developers

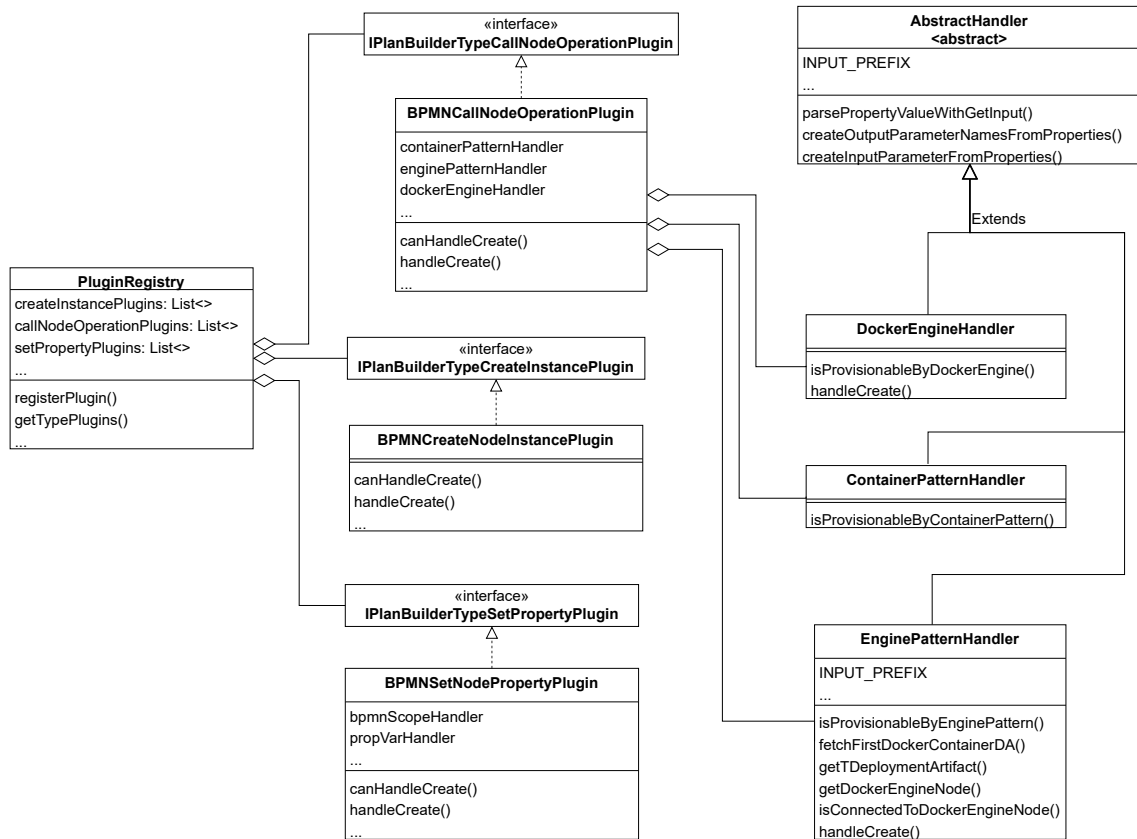


Figure 5.5: UML Class Diagram for BPMN Plugin

to understand the testing code base and enables them to make changes quickly. JUnit ¹ is used as the unit test framework for OpenTOSCA. JUnit is a free-to-use testing tool used for Java programming language. JUnit is most commonly used to test the functionality of individual methods [WSW05]. It provides assertions to identify the test method.

¹<https://junit.org/junit5/>

6 Conclusion and Future Work

In this master thesis, a concept for generating imperative provision plans in executable BPMN for cloud applications based on the TOSCA standard is developed. The thesis focus on generating an imperative provision plan automatically. Deploying and executing provision plan creates an instance of the complete application. Concepts from the state of the art research are built upon to support and develop the proposed concepts in this thesis. Further, the feasibility of the concepts introduced is shown with the help of a prototype designed using the OpenTOSCA Container.

The fundamentals concepts and terminologies of OASIS TOSCA, OpenTOSCA, BPMN 2.0 are introduced in Chapter 2. An extensive survey of generating provisioning plans regarding TOSCA and BPMN 2.0 is presented in Chapter 3. Several approaches to generating provision plans and their use cases are discussed. These approaches lay the foundation for the contributions presented in this thesis.

The major contribution of this thesis is the development of the concepts for generating imperative provision plans in executable BPMN for cloud applications based on the TOSCA standard. The executable BPMN 2.0 plan facilitates the capability of automatic provision, which is the critical property of Cloud Computing. The graphical component of the BPMN 2.0 plan in human-understandable execution order allows the application developer to customize the generated plan for individual requirements. Script Task is chosen to be the core activity in the provisioning plan for the atomic and excitable property. Plan generation is divided into three steps: Skeleton Generation, Subprocess Completion, and Plan Finalization. An algorithm is introduced to generate BPMN Activity Skeleton in a logical order based on POG, which is derived from the Topology Template. Subprocess Completion completes detail of Script Tasks using the extensible Plugin for individual Node Type. Various types of Groovy Scripts are used to interact with the well-defined semantics, data models, and rich APIs of the OpenTOSCA Core. A pattern-based approach is proposed to determine the Operation to invoke for Call Node Operation Script Task. For Plan finalization, another algorithm is introduced to generate BPMN Diagram elements based on the generated Process elements in a human-readable format.

A prototype is developed based on the OpenTOSCA ecosystem to test the feasibility of the concepts. The Plan Generator reuses the concepts introduced by [Kép13]. BPMN Plan Builder, Template Library, Script Task Plugin, and BPMN Facade are implemented. BPMN Plan Generator is integrated with the OpenTOSCA Container with a user-specific language option. The BPMN executable plan generated from the sample application has been tested successfully in the embedded Camunda Platform.

The future works include extending the BPMN Plan Generator to support generating different plans, such as termination, scale-out, and migration plans. This would also include developing new plugins for different Node Types. Another future work includes extending the BPMN Plan Generator to support generating compensation path in the Build Plan to enable fault-tolerant provisioning [CCMT18].

Bibliography

- [ASF16] The Apache Software Foundation. *Apache ODE™ – The Orchestration Director Engine*. 2016. URL: <http://ode.apache.org> (cit. on pp. 16, 25).
- [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: *Service-Oriented Computing*. Springer Berlin Heidelberg, 2013, pp. 692–695 (cit. on p. 23).
- [BBK+13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, J. Wettinger. “Integrated Cloud Application Provisioning: Interconnecting Service-Centric and Script-Centric Management Technologies”. In: *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*. Springer Berlin Heidelberg, 2013, pp. 130–148 (cit. on pp. 15, 36).
- [BBK+14] U. Breitenbücher, T. Binz, K. Kepes, O. Kopp, F. Leymann, J. Wettinger. “Combining Declarative and Imperative Cloud Application Provisioning Based on TOSCA”. In: *2014 IEEE International Conference on Cloud Engineering*. IEEE, 2014 (cit. on pp. 20, 23–25, 36, 37).
- [BBKL13] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. “Pattern-based Runtime Management of Composite Cloud Applications”. In: *CLOSER 2013 - Proceedings of the 3rd International Conference on Cloud Computing and Services Science (2013)* (cit. on p. 23).
- [BBKL14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann. “Vinothek-A Self-Service Portal for TOSCA.” In: *Proceedings of the 6th Central-European Workshop on Services and their Com- position (ZEUS 2014)*. Citeseer. 2014, pp. 69–72 (cit. on p. 23).
- [BEK+16] U. Breitenbücher, C. Endres, K. Képes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. “The OpenTOSCA Ecosystem - Concepts & Tools”. In: *European Space project on Smart Systems, Big Data, Future Internet - Towards Serving the Grand Societal Challenges*. SCITEPRESS - Science and Technology Publications, 2016 (cit. on pp. 16, 22).
- [CCMT17] D. Calcaterra, V. Cartelli, G. D. Modica, O. Tomarchio. “Combining TOSCA and BPMN to Enable Automated Cloud Service Provisioning”. In: *Proceedings of the 7th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2017 (cit. on pp. 38, 39).
- [CCMT18] D. Calcaterra, V. Cartelli, G. D. Modica, O. Tomarchio. “Exploiting BPMN Features to Design a Fault-aware TOSCA Orchestrator”. In: *Proceedings of the 8th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2018 (cit. on p. 63).
- [CT12] M. Chinosi, A. Trombetta. “BPMN: An introduction to the standard”. In: *Computer Standards & Interfaces* 34.1 (2012), pp. 124–134 (cit. on p. 27).

- [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 (IM 2011) and Workshops*. IEEE, 2011 (cit. on p. 36).
- [FLR+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns*. Springer Vienna, 2014 (cit. on p. 49).
- [KBBL12] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “BPMN4TOSCA: A Domain-Specific Language to Model Management Plans for Composite Applications”. In: *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2012, pp. 38–52 (cit. on pp. 16, 17, 35, 42).
- [KBBL13] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann. “Winery – A Modeling Tool for TOSCA-Based Cloud Applications”. In: *Service-Oriented Computing*. Springer Berlin Heidelberg, 2013, pp. 700–704 (cit. on p. 23).
- [Kép13] K. Képes. “Konzept und Implementierung einer Java-Komponente zur Generierung von WS-BPEL 2.0 BuildPlans für OpenTOSCA”. In: (2013) (cit. on pp. 55, 57, 58, 61, 63).
- [Ley09] F. Leymann. “Cloud Computing: The Next Revolution in IT”. In: (2009), pp. 3–12 (cit. on p. 15).
- [Ley10] F. Leymann. “BPEL vs. BPMN 2.0: Should You Care?” In: *Lecture Notes in Business Information Processing*. Springer Berlin Heidelberg, 2010, pp. 8–13 (cit. on p. 17).
- [Mie10] R. Mietzner. *A method and implementation to define and provision variable composite applications, and its usage in cloud computing*. en. 2010 (cit. on p. 37).
- [OAS07] OASIS. *Web Services Business Process Execution Language (BPEL)*. Version 2.0. 2007. URL: <http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html> (cit. on pp. 15, 24).
- [OAS13a] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer*. 2013. URL: <http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html> (cit. on pp. 16, 19, 22, 23).
- [OAS13b] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*, 2013. URL: <http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.html> (cit. on pp. 15, 19).
- [Ola03] M. Olan. “Unit Testing: Test Early, Test Often”. In: *J. Comput. Sci. Coll.* 19.2 (2003), pp. 319–328 (cit. on p. 61).
- [OMG11] OMG. “Business Process Model and Notation (BPMN)”. Version 2.0. In: *OMG Specification, Object Management Group* (2011) (cit. on pp. 15, 29).
- [Pet11] T. G. Peter Mell. *The NIST Definition of CloudComputing*. 2011 (cit. on p. 15).
- [Sil11] B. Silver. *BPMN Method and Style: With BPMN Implementer’s Guide*. Cody-Cassidy Press, 2011 (cit. on pp. 27, 30, 34).
- [WBB+14] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, M. Zimmermann. “Unified Invocation of Scripts and Services for Provisioning, Deployment, and Management of Cloud Applications Based on TOSCA”. In: *Proceedings of the 4th International Conference on Cloud Computing and Services Science*. CLOSER 2014. Barcelona, Spain: SCITEPRESS - Science and Technology Publications, Lda, 2014, pp. 559–568 (cit. on pp. 25, 26).

- [WBB+15] J. Wettinger, T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. “Streamlining Cloud Management Automation by Unifying the Invocation of Scripts and Services Based on TOSCA”. In: *Cloud Technology*. IGI Global, 2015, pp. 2240–2261 (cit. on pp. 25, 26).
- [Whi04] S. A. White. “Introduction to BPMN”. In: *BPTrends* (2004) (cit. on pp. 26, 27).
- [WSW05] M. Wick, D. Stevenson, P. Wagner. “Using testing and JUnit across the curriculum”. In: *Proceedings of the 36th SIGCSE technical symposium on Computer science education - SIGCSE '05*. ACM Press, 2005 (cit. on p. 62).

All links were last followed on April 15, 2022.

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.

A handwritten signature in black ink, enclosed within a thin black rectangular border. The signature is cursive and appears to read 'J. J. Li'.

Taipei, May 4, 2022

place, date, signature