Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis No. 3747

# Modeling Approaches to Enable Data Shipping and Function Shipping by Means of TOSCA

Karoline Saatkamp

**Course of Study:** Wirtschaftsinformatik

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

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

**Commenced:** February 1, 2016

**Completed:** July 29, 2016

**CR-Classification:** C.2.4, E.0, I.6.5

# Abstract

The Cloud Computing paradigm requires standardized mechanisms to enable the portability of applications and data in and between clouds to avoid vendor lock-in and to enable the automated deployment to reduce the management effort. With the standard Topology and Orchestration Specification for Cloud Applications (TOSCA) language, application topologies representing the components of an application and their relations can be defined. By means of a self-contained and portable archive the application can be moved between different environments and can be automatically deployed. However, data management aspects in terms of data provisioning and wiring with the application are not considered by TOSCA. Due to the increasing amount of captured data in context of the Internet of Things, for example, and new data analysis opportunities, the portability of data and the connection between the data and the functions processing these data are getting more important. Thereby, either the data can be shipped to the function site (data shipping) or the function can be shipped and executed at the data site (function shipping).

In this thesis various modeling concepts are developed to enable data shipping and function shipping. Different approaches for modeling the assignment between data and function and for the integration of data extraction and transformation mechanisms are formulated. Each concept is applicable under different environmental conditions and is described in an abstract manner. How these abstract modeling concepts can be implemented by means of TOSCA is shown by the TOSCA realization options, which are discussed in this thesis. Furthermore, the expressiveness of TOSCA and extension options for TOSCA to implement the modeling concepts are analyzed.

# Contents


1 Introduction 11

2 Fundamentals 15
2.1 Topology and Orchestration Specification for Cloud Applications . . . . . 15
2.2 Terminology: Data Shipping versus Function Shipping . . . . . . . . . . 21

3 Related Work 23
3.1 Function Shipping Concepts . . . . . . . . . . . . . . . . . . . . . . . . . 23
3.2 Data Shipping Concepts . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4 Data Shipping and Function Shipping Use Cases 33
4.1 Modeling Scenario for Data Shipping and Function Shipping . . . . . . . 33
4.2 Use Cases for Data Shipping . . . . . . . . . . . . . . . . . . . . . . . . . 34
4.3 Use Cases for Function Shipping . . . . . . . . . . . . . . . . . . . . . . . 35

5 Modeling Concepts for Data Shipping and Function Shipping 37
5.1 Concept 1: Uniquely Addressable Data Resource . . . . . . . . . . . . . . 40
5.2 Concept 2: Uniquely Addressable Data Resource with Assigned Processing Logic Identifier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.3 Concept 3: Uniquely Addressable Processing Logic . . . . . . . . . . . . 54
5.4 Concept 4: Uniquely Addressable Processing Logic with Assigned Data Identifier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58
5.5 Concept 5: Data Connector between Processing Logic and Data Resource 68
5.6 Concept 6: Data Connector with Transformation Capability . . . . . . . . 73
5.7 Concept 7: Data Connector between Processing Logic and Data Collection 84
5.8 Concept 8: Data Connector with Operations Applied to Multiple Data Resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92
5.9 Modeling Concepts Summary . . . . . . . . . . . . . . . . . . . . . . . . 100

6 Analysis of Extension Options for TOSCA 103

7 Conclusion and Future Work 105

Bibliography 107

# List of Figures


2.1 General elements of a TOSCA Service Template with the Application Interface extension . . . 16
2.2 Structur of a CSAR . . . 20

3.1 Examplary architecture with AWS Lambda . . . 26
3.2 SIMPL Metadata classes . . . 30

4.1 Modeling scenario . . . 33
4.2 Use cases for data shipping . . . 35
4.3 Use cases for function shipping . . . 36

5.1 Overview of the modeling concepts and the related TOSCA realization options . . . 38
5.2 Concept 1: Uniquely addressable data resource . . . 40
5.3 Option 1.1: Data location assignment via deployment artifact . . . 42
5.4 Exemplary structure of an CSAR containing data . . . 43
5.5 Option 1.2: Data location assignment via property . . . 45
5.6 Concept 2: Uniquely addressable data resource with assigned processing logic identifier . . . 47
5.7 Option 2.1: Processing logic assignment via property . . . 48
5.8 Option 2.2: Processing logic input parameter assignment via property . . . 51
5.9 Concept 3: Uniquely addressable processing logic . . . 54
5.10 Option 3.1: Processing logic Node Type definition . . . 56
5.11 Exemplary structure of a CSAR containing the processing logic . . . 57
5.12 Concept 4: Uniquely addressable processing logic with assigned data identifier . . . 58
5.13 Option 4.1: Data assignment via property . . . 60
5.14 Option 4.2: Data assignment via input parameter . . . 63
5.15 Option 4.3: Data assignment via property and input parameter matching 65
5.16 Concept 5: Data connector between processing logic and data resource . 68
5.17 Option 5.1: Data connector with input parameter matching . . . 70
5.18 Concept 6: Data connector with transformation capability . . . 73
5.19 Option 6.1: Transformation assignment via property . . . 75

5.20 Option 6.2: Transformation assignment via transformation interface . . 79
5.21 Option 6.3: Explicitly modeled data query . . . . . . . . . . . . . . . . . . 82
5.22 Concept 7: Data connector between processing logic and data collection 85
5.23 Option 7.1: Separated data resources . . . . . . . . . . . . . . . . . . . . . . . 87
5.24 Exemplary structure of an CSAR containing two different files . . . . . . 88
5.25 Option 7.2: Single data collection . . . . . . . . . . . . . . . . . . . . . . . . 90
5.26 Concept 8: Data connector with operations applied to multiple data resources . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 93
5.27 Option 8.1: Join operation assignment via join interface . . . . . . . . . 95
5.28 Option 8.2: Implicit join operation . . . . . . . . . . . . . . . . . . . . . . . 98
5.29 Overview of all concepts and the related realizable use cases . . . . . . . 102

# List of Listings


5.1 Option 1.1: Node Type definition *DataResource* and Node Template *DataResourceFile* . . . 42
5.2 Option 1.1: Artifact Type definition *FileArtifact* and Artifact Template *FileContent* . . . 43
5.3 Node Type definition *DataResource* with properties . . . 45
5.4 *Location* property definition for Node Type *DataResource* . . . 46
5.5 Node Template *DataResourceFile* with property *Location* . . . 46
5.6 Option 2.1: Properties definition for Node Type *DataResource* . . . 49
5.7 Option 2.1: Node Template *DataResourceFile* . . . 49
5.8 Option 2.2: Properties definition for Node Type *DataResource* . . . 52
5.9 Option 2.2: Node Template *DataResourceFile* . . . 53
5.10 Option 3.1: Node Type definition *ProcessingLogic* . . . 56
5.11 Option 3.1 and 4.2: Node Template *Calculator* . . . 56
5.12 Artifact Type definition *JARArtifact* and Artifact Template *CalculatorInstallable* . . . 57
5.13 Option 4.1: Node Type definition *ProcessingLogic* . . . 60
5.14 Option 4.1: Properties definition for Node Type *ProcessingLogic* . . . 61
5.15 Option 4.1: Node Template *Calculator* . . . 61
5.16 Option 4.2: Node Type definition *ProcessingLogic* . . . 63
5.17 Node Type definition *ProcessingLogic* with application interface . . . 66
5.18 Option 4.3: Properties definition for Node Type *ProcessingLogic* . . . 66
5.19 Option 4.3: Node Template *Calculator* . . . 67
5.20 Relationship Type definition *DataConnector* with properties . . . 71
5.21 Properties definition for Relationship Type *DataConnector* . . . 71
5.22 Option 5.1: Relationship Template *DataConnectorFile* . . . 71
5.23 Extended properties definition for Node Type *DataResource* . . . 76
5.24 Option 6.1: Properties definition for Relationship Type *DataConnector* . . . 77
5.25 Option 6.2: Relationship Type definition *DataConnector* . . . 80
5.26 Node Type definition *DataCollection* . . . 91
5.27 Properties definition for Node Type *DataCollection* . . . 91
5.28 Node Template *DataCollectionFileSystem* . . . 91

# 1 Introduction

Cloud Computing changes the way enterprises use IT. Large investments into own IT infrastructures, applications, and costly IT maintenance become obsolete as software, platform, and infrastructure services can be used on demand [Arm+10]. The automated management of the offered enterprise applications in terms of deployment and configuration as well as the portability between different environments is of great importance to ensure efficient cloud services. Often parts of the application have to be redesigned and reimplemented when the cloud provider changes [Bin+14; Pet+14].

The standard Topology and Orchestration Specification for Cloud Applications (TOSCA) addresses three problems of Cloud Computing: (1) automated application deployment and management, (2) portability between different environments, and (3) interoperability and reusability of application components. TOSCA enables the full-automated management of the application through the combination of two main concepts: application topologies and management plans. An application topology describes an application's structure and the management capabilities of the components. Management plans define process flows that have to be executed to deploy, configure, manage, and operate the application. All relevant files are packaged into a self-contained and portable archive, called Cloud Service Archive (CSAR) to move the application between different environments [Bin+14; OAS13b].

Thus, TOSCA focuses on application structures and their management but does not consider how data can be moved in and between clouds and how they can be associated with the business logic. This affects all aspects of the data management in terms of data provisioning and wiring with the application.

New trends like Internet of Things or Cyber-Physical System-based manufacturing in the manufacturing industries increase the volume of data captured by multiple sensors [Atz+10; Lee+14]. Especially in the industrial environment as part of the fourth industrial revolution (called Industrie 4.0 in Germany) and Industrial Internet initiatives, intelligent interconntected machines and products capture data during the production process, e.g., about machine performance or health which can be used to increase reliability, availability and performance [For+13; Con16]. Often the situation arises that " [...] Who owns the data is often not capable of analyzing it" [Bud+15]. This means, the processing logic which is required to analyze for example the machine

data is often provided by the machine manufacturer or a data scientist, whereas the data are captured and owned by the respective production company [Lee+14; Bud+15].

For the processing of data that resist at another location than the processing logic, two approaches can be differentiated in terms of the provisioning of data and processing logic: data shipping and function shipping. In case of data shipping, the data are shipped to the location where the processing logic is executed. In case of function shipping, the processing logic is shipped to and executed at the location where the data resist. For each approach, two use cases can be distinguished: shipping of the actual data or processing logic and shipping of a reference to a remote location of the data or the processing logic. In the first case, the data or processing logic, respectively, are packaged in an archive and sent to the target runtime environment. In the second case, only a reference to the data or processing logic is packaged in an archive and transmitted. The data or processing logic can then be retrieved from the remote location at a later time.

The data are often stored in various heterogeneous data sources by the different companies. Due to the fact the processing logic is not only developed for one specific company and data source, methods are required to enable the integration and processing of heterogeneous data sources. Extraction, transformation, and load (ETL) mechanisms are already known for data provisioning in different areas, such as business intelligence solutions [Kem+04] and data flows in workflow modelling [Sad+04]. Especially extraction and transformation mechanisms are also required in other areas to enable the data integration for example in web service compositions [Lé+08] and the Internet of Things [Bar+12]. These are key aspects in terms of data provisioning to enable data processing of heterogeneous data.

Due to the capability of TOSCA to enable the automated deployment of applications and the portability between different environments by means of a specific archive format, TOSCA is a good basis for modeling data shipping and function shipping. So far, data management aspects are not considered in TOSCA. For this, approaches to model data, extraction and transformation mechanisms, and the assignment between data and processing logic have to be developed.

The first goal of this master's thesis is the development and evaluation of modeling concepts to enable (1) data shipping as well as function shipping between different environments and (2) the association between the processing logic and the data. With the concepts various modeling approaches for the assignment between data and processing logic as well as for extraction and transformation mechanisms have to be formulated. Each concept is applicable under different environmental conditions. The second goal is to specify options which implement these modeling concepts by means of TOSCA. In particular, the usage of existing TOSCA language elements and required extensions of TOSCA are analyzed.

This thesis is structured in the following way:

**Chapter 2 – Fundamentals** explains the fundamentals this thesis bases on. This includes the benefits and syntax of TOSCA as well as the approach of data shipping and function shipping and their usage in different research areas.

**Chapter 3 – Related Work** discusses the work related to this thesis in terms of data shipping and function shipping concepts and evaluates the adaptability for the modeling concepts.

**Chapter 4 – Data Shipping and Function Shipping Use Cases** presents the modeling scenario, which serves as an example for the concepts and options. Additionally, the data shipping as well as function shipping use cases, which should be realized by the modeling concepts, are described in detail.

**Chapter 5 – Modeling Concepts for Data Shipping and Function Shipping** covers all developed modeling concepts and the corresponding TOSCA realization options. The concepts and TOSCA realization options are described and the applicability of each concept and option for data shipping and function shipping use cases are evaluated. The required TOSCA extensions for each option are discussed.

**Chapter 6 – Analysis of Extension Options for TOSCA** analyzes the different possibilities available to extend TOSCA. Advantages and Disadvantages are discussed and the extensions used for the TOSCA realization options are evaluated.

**Chapter 7 – Conclusion and Future Work** summarizes the contributions of this thesis and suggests related topics for future research.

# 2 Fundamentals

## 2.1 Topology and Orchestration Specification for Cloud Applications

The OASIS standard TOSCA introduces an XML-based language to describe the structure of composite cloud applications and their management during their life cycle [OAS13b]. The three main goals of TOSCA are (1) automated management, (2) portability of applications, and (3) definition of interoperable and reusable application components [Bin+14]. TOSCA allows the application creator that has deep knowledge about the application to specify management plans to fulfill management tasks. They are used to deploy, configure, and manage the application. Therefore, users do not have to obtain additional knowledge to manage the application. These management plans are just required in case of a imperative provisioning approach, i.e., all management tasks are explicitly defined [Bre+16]. Another approach is the declarative provisioning. Simple management tasks such as the provisioning of an application can be fullfilled without user-defined management plans [Bre+16]. To enable the portability, TOSCA formalizes application topologies and management plans in a self-contained way. Each environment supporting TOSCA can deploy and manage the application. The third goal is achieved by defining the components in a reusable manner. Different parties can define components which can be composed in several applications.

The two main concepts of TOSCA are (1) application topologies and (2) management plans [Bin+14]. An application topology describes the structure of the application's components, the relations between them, and the management capabilities of each component. Management capabilities are operations offered by a component to manage this component. Such management operations are used to, e.g., install, configure, or uninstall the component. The management plans combine these management operations to define process flows for different management tasks. To define these plans, existing workflow languages like BPMN or BPEL are used.

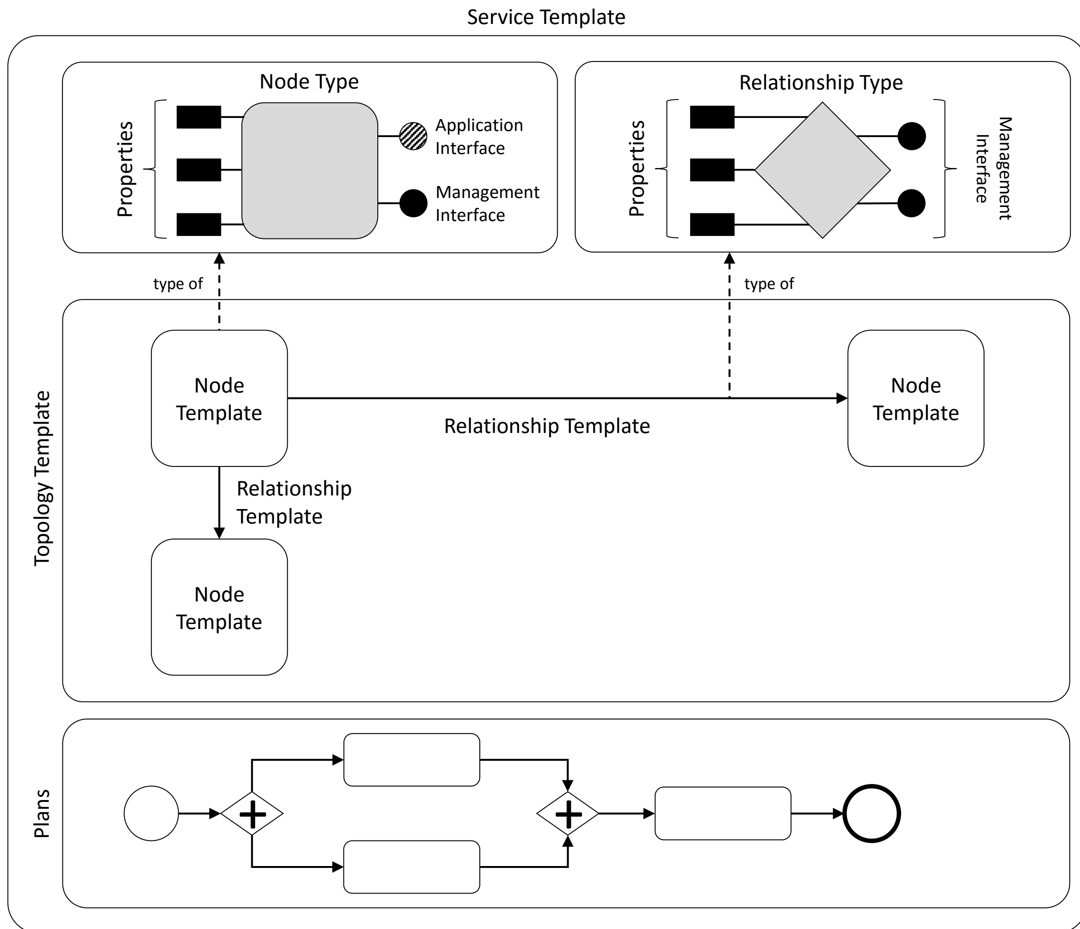



**Figure 2.1:** General elements of a TOSCA Service Template with the Application Interface extension adopted from [OAS13b]

### 2.1.1 TOSCA Syntax

In this section the most important elements of TOSCA are described, in particular the elements that define application's components and the relationship among them. It is based on the latest published version of the TOSCA specification [OAS13b].

The TOSCA specification provides a metamodel to define IT services. A *Service Template*, depicted in Fig. 2.1, contains the application topology as well as the management plans and therefore describes the complete service. The structure of the application is defined by a *Topology Template* and the management plans by means of *Plans*.

The Topology Template comprises a set of *Node Templates* and *Relationship Templates*. The Node Templates represent the components the application consists of and can be instantiated. That means a Node Template indicates only the occurrence of a component

and not a concrete instance of the component. The *Node Types* define reusable entities and their *Properties* and *Interfaces*. Node Templates of these types can be specified.

A service can for example consist of an application server and an application. To describe this service in a Topology Template, a Node Template of type *Application Server* and a Node Template of type *Application* is included. The Application Node Type defines properties such as the owner of the application and interfaces with operations to install, configure, and shutdown the application.

Besides the Node Templates, *Relationship Templates* are essential elements of the Topology Template. A Relationship Template specifies the relationship between two components of an application. The source element and target element determine the Node Templates for which this relationship exists. *Relationship Types* define reusable entities, which are types for the Relationship Templates. A Relationship Template between an Application Node Template and an Application Server Node Template is for example of type *Hosted On*. Other examples for relationships between Node Templates, i.e., cloud service components, are *Installed On*, *Deployed On*, or *Connected To*.

Plans combine the management operations of the components to fulfill management tasks. The specification of the Plans relies on existing workflow languages like BPMN or BPEL. Due to the fact Plans are not covered in this thesis, they are not further discussed. The *Application Interface*, which represents an extension of TOSCA is covered later in Section 2.1.3.

The elements of TOSCA used in the modeling concepts in Chapter 5 are explained in the following, whereby only those elements are described that are required to understand the modeling concepts. Elements which are not required in this thesis are omitted.

*Node Type and Node Template*

As mentioned above, Node Types define reusable entities for Node Templates. The Node Templates form the concrete application. The TOSCA specification does not contain concrete Node Types. They have to be standardized by domain experts and can then be used by an application architect to specify an application's structure [OAS13a]. Only the metamodel language elements are contained in the specification.

A Node Type is identified by the attribute *name* which has to be unique. The optional attribute *abstract* is used to specify if instances of a Node Template using this type can be created. Via the *PropertiesDefinition* element the structure of properties, i.e., names, datatypes, and allowed values of the properties, is defined. The properties are defined in form of a XML schema and referenced by the Node Type [W3C12]. A Node Template of this type assigns values to the defined properties.

The Node Templates use the Node Types to specify the components an application consists of. The attribute *id* identifies a particular Node Template and has to be unique.

Optionally, an attribute *name* can be specified. The attribute *type* refers to the Node Type the Node Template based on. Furthermore, initial values for the *Properties* defined by the Node Type are specified for the Node Template. An instance document of the XML schema defining the Node Type properties is provided. For the instantiation of the Node Template all properties must be valid according to the properties definition. In case the Node Type does not define properties, no properties are assigned to the Node Template. Constraints for the values of the Properties can be specified by the *PropertyConstraints*. A *PropertyConstraint* element points to the *property* the constraint applies on and indicates the *constraintType*, i.e., the semantic and the format of the constraint, by means of an URI.

Two further optional elements of a Node Template are *Policies* and *DeploymentArtifacts*. The first one specifies policies associated with the Node Template. Policies are non-functional requirements and quality-of-service (QoS) aspects valid for the Node Template. Such policies are for example high availability or a specific power consumption. The second one, Deployment Artifacts, specifies the artifacts required for the instantiation and implementation of the Node Template. These are software files such as installables or executables. For example an image containing all files of an Tomcat[1] application server can be a deployment artifact of a Node Template Application Server. A *DeploymentArtifact* element has a *name* and specifies the *Artifact Type* by the attribute *artifactType*. With the attribute *artifactRef* the concrete *Artifact Template*, which serves as a deployment artifact for the Node Template is identified. An Artifact Type "EAR file" (Enterprise Application Archive)[2] might be defined to describe java archive files. An Artifact Template of this type represents a concrete EAR file, which is the deployment artifact for a Node Template.

*Relationship Type and Relationship Template*

Similar to Node Types, Relationship Types define reusable entities for Relationship Templates. They specify the relation between application's components. Like Node Types, concrete Relationship Types have to be standardized by domain experts to make them available for the application architects to wire the set of Node Templates [OAS13a].

A Relationship Type is uniquely identified by the attribute *name* and can optionally be declared as *abstract*. The *PropertiesDefinition* element specifies the properties in form of an XML schema. The properties can be defined externally and referred by the Relationship Type. With the two optional elements *ValidSource* and *ValidTarget* the type of elements allowed as source or target of the relationship can be specified. If these elements are not determined, any Node Type can serve as source or target.

[1]http://tomcat.apache.org/index.html
[2]http://docs.oracle.com/javaee/6/tutorial/doc/bnaby.html

A Relationship Template bases on a Relationship Type and specifies the relation between two Node Templates. For each Relationship Template the source and target of the relationship have to be determined. The attribute *id* is the identifier of the Relationship Template. Additionally, a *name* can be specified. The Relationship Type providing the type of the Relationship Template is specified by the attribute *type*. In case properties are defined by this type, initial values for these properties can be assigned to the Node Template by the *Properties* element. For this, an instance of the XML schema defining the properties is created. In the same way as described above, *PropertyConstraints* can be defined for each property. The elements *SourceElement* and *TargetElement* are provided to determine the origin and target of the relationship by referencing the ID of a Node Template comprised in the same Service Template.

Node Types, Relationship Types, and properties are often defined in separate TOSCA Definitions or XML schema documents because they are specified by different experts. These external documents can be imported to Definitions documents containing the Service Template. All used types in a Topology Template have to be part of the same Definitions document or have to be imported. This supports a modular design of Service Templates.

To enable the deployment and management of an application in a certain environment, all relevant definitions and artifacts have to be available in the environment. This includes the Service Template and type definitions as well as the required software files.

### 2.1.2 TOSCA Cloud Service Archive

A standardized archive format to package the Service Template and the associated files is the Cloud Service Archive (CSAR) [OAS13b]. Besides the TOSCA Definitions documents, Plans, deployment artifacts as well as implementation artifacts representing the executables of the management operations are included. Thus, the application can be packaged in a fully self-contained manner [Bin+14].

A CSAR is a zip file organized in subdirectories with several different files. It can be structured as required for a specific application, but two subdirectories are fixed: the *TOSCA-Metadata* and the *Definitions* subdirectory. The TOSCA-Metadata directory contains a *TOSCA meta file* with metadata about the CSAR itself such as the version and the creator of the CSAR and metadata describing the other files a CSAR is comprised of. The other mandatory one is the Definitions directory. It includes all TOSCA Definitions documents specifying for example the Service Template, related Node Types, and Relationship Types required for the application. Further subdirectories required to package a specific application can be defined by the creator [OAS13b].

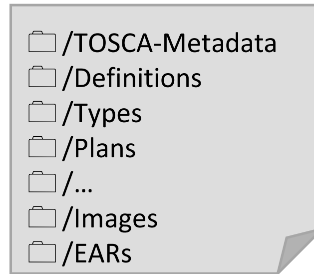



**Figure 2.2:** Structur of a CSAR adopted from [OAS13b]

In Fig. 2.2 an example of the structure of a CSAR is presented. In this example an extra folder for the types and a folder containing the plans in the form of, for example, BPMN files are created. Additionally, two subdirectories *Images* and *EARs* are defined containing deployment artifacts, for example an image for the installation of an application server and an EAR for an application hosted on the application server.

This standardized format is required to make the application portable between different TOSCA runtime environments. A TOSCA runtime environment can process the CSAR and can deploy and instantiate the application described by the contained Service Template.

### 2.1.3 TOSCA Application Interface Extension

The existing TOSCA standard allows to specify management operations offered by a Node Type to manage itself in terms of the installation, configuration, and other management tasks via standardized interface descriptions [Bin+14]. However, application operations cannot be modeled with the elements provided by the TOSCA standard [OAS13b]. In contrast to management operations, application operations provide the actual functionality of the application. They are installed by the management operations and are available when the deployment and configuration of the application is finished [Zim16]. Therefore, the operations can be made available for invocations by other applications.

Zimmermann [Zim16] introduced an *ApplicationInterfaces* element as an extension of TOSCA to differ between management operations and application operations. Several interfaces can be defined in the ApplicationInterfaces element. For one *Interface* the same rules apply as for the management interfaces defined for Node Types, which are set in the TOSCA specification [OAS13b].

An Interface is identified by a *name* which has to be unique for the Node Type. Within the Interface multiple *Operation* elements can be defined. One Operation element defines an operation that an instance of this Node Type provides to others. An Operation

has an unique *name* within the Interface. For the Operation multiple *InputParameters* and *OutputParameters* can be defined depending on the operation it represents. An *InputParameter* as well as an *OutputParameter* are identified by an unique *name* and specify what type of input parameter are expected. Therefore, the application functionality can be modeled in TOSCA.

## 2.2 Terminology: Data Shipping versus Function Shipping

Two approaches for data processing can be distinguished: data shipping and function shipping. In general, data shipping means that the data are retrieved from the data location and processed where the computation resides, whereas in case of function shipping the computation is executed at the data site. These principles are applied in different research fields in various manifestations, which range from microcomputers and microprocessor systems to database systems and distributed computing. Especially function shipping is a much-discussed approach because in many areas function shipping replaces data shipping to deal with a large amount of data.

In the area of microcomputers and microprocessor systems function shipping is a rich area of research known as *processing-in-memory (PIM)* and *near-data processing (NDP)* [Bal+14]. With the increasing amount of data the system model shifted from a data shipping approach to a function shipping one. The goal of a system model that is based on function shipping is to place the computation resources as close as possible to the location of the data. The research in the field of PIM started in the 1990s with integrating RAM and processing units on a single PIM chip [Pat+97; Ell+99]. The main purpose was to minimize the data movement which has an impact on the memory latency and energy efficiency. The NDP concept applies also to other levels of the memory hierachy like hard disk drives and solid-state devices [Bal+14; Rie+98; Tiw+13]. In this research field the microstructure of a single system or computer device are considered.

On a different level the approaches are applied to database systems in terms of query executions. For client-server database architectures *data shipping* and *query shipping (function shipping)* are distinguished. In case of data shipping, required data are retrieved from the server and processed on the client, while in case of query shipping the query is shipped to and executed on the server and only the result is sent to the client. On the one hand query shipping reduces the communication costs because only the query result instead of the raw data has to be sent. On the other hand data shipping better utilizes the resources available at the client machine and enables client caching in which data can be prefetched for the processing at a later time. Because neither data shipping nor query shipping is applicable in every situation, a *hybrid shipping* approach is considered.

Each operation of a query is executed either at the server or on the client side [Fra+96; Vor+04].

Not only queries or parts of queries can be shipped to and executed on the server through query shipping and hybrid shipping, but also parts of the application code. The database middleware system MOCHA (Middleware Based On a Code Shipping Architecture) provides a possibility to ship application code to the data site and to automatically deploy it [RM+00]. MOCHA integrates distributed data sources and extends the query shipping approach by the capability to deploy Java code on the remote sites to manipulate the data. This is called *code shipping*. The Java classes containing application-specific functionality are shipped, deployed and executed for a given data resource.

Another approach is not to ship the function, but to define and to store them at the data site. One possibility are *stored procedures* or *stored functions* which are defined and stored on the server [Har+06]. Thus, database-centric logic used by different applications can be isolated. The stored procedure or stored function can be invoked by an application with the respective parameters and the result is sent back to it. This is also called function shipping or function request shipping, respectively [Cor+86]. It differs from the above described function shipping approach in terms of the content shipped to the data site: in this case only a function call is shipped. This does not correspond with the definition of function shipping used in this thesis. Sending a request containing the function identifier with the corresponding parameters which should be used is also called operation shipping [Sel+98].

The function request shipping or operation shipping approach is also used in distributed computing environments in which for example remote procedure calls are used. The client sends a request, which identifies the required function and contains the parameters required for the server to fulfill the request. After processing, the result is sent back to the client. In addition to the mentioned research areas, many others use the data shipping or function shipping principles. A simple example for the data shipping approach is also the retrieval of files from a file server.

Although data shipping and function shipping are used in different areas, the underlying approach is always the same. In the context of this thesis the data shipping and function shipping approaches are considered in relation to data analysis. The initial situation is that processing logics, which are required to analyze data resources are stored at a different location than the data. Either the data are transferred to the processing logic (data shipping) or the processing logic is packaged, transferred to, deployed, and executed as close as possible to the data (function shipping). Compared to the areas of function shipping described before, the whole processing logic is deployed and executed near the data and not only parts of the computation logic. The execution can then be triggered for example by a workflow or manually.

# 3 Related Work

As already mentioned, the function shipping and data shipping approaches are used in various research areas. In this chapter existing concepts in the area of function shipping as well as data shipping are presented in detail. It is evaluated which aspects of these concepts are useful to derive modeling concepts for data shipping and function shipping.

## 3.1 Function Shipping Concepts

TOSCA is introduced, among other things, to enable portability of applications between different cloud environments by means of a CSAR as described in Section 2.1. Applications can be modeled, packaged, transferred, and deployed in another cloud provider's environment supporting TOSCA. This includes the whole application topology but does not consider the connection with the data. Other concepts exist to run processing logic close to the data. In the following the concept of code shipping and serverless architecture are described in detail and their usefulness in terms of the modeling concepts are evaluated. For both concepts just the code is shipped without the whole topology stack.

### 3.1.1 Middleware Based On a Code Shipping Architecture

As already mentioned in Section 2.2, in context with database systems different mechanisms in terms of function shipping are used such as stored procedures, stored functions, query shipping, or code shipping. In particular aspects of the code shipping mechanism provided by the database middleware system MOCHA for distributed data sources are useful for the development of the modeling concepts for function shipping [RM+00].

The main principle of MOCHA is to ship the code used for operations on a data source to the corresponding data site and to deploy the code in an automated manner. The goal is to minimize the data movement over the network. For this in MOCHA Java code containing application-specific functionality to manipulate data at a remote site can be shipped and deployed at the remote data site automatically. This refers to functionality

which are not provided by the data source management systems. Compared to the described query shipping and hybrid shipping approach which are restricted to the execution of operations already implemented at the data site, new functionality to process a query can be implemented on demand and becomes available automatically.

MOCHA consists of four main components: the *Client Application*, the *Query Processing Coordinator (QPC)*, the *Data Access Provider (DAP)*, and the *Data Server*. The Client Applications are in the most cases clients used as GUI to visualize the query results. Different kinds of clients are supported by MOCHA. The queries from the clients are handled by the QPC. The QPC is responsible for the deployment of the functionality required to execute the query at the remote data sources. A query execution plan is created. It indicates the distribution of the query to data sources as well as the functionality which has to be dynamically deployed at each data site. The required code is stored in a code repository and can be retrieved and provided by the QPC for the deployment. The DAP is located at the data site and provides an uniform access mechanism for the QPC as well as an query execution engine to run the application-specific code at the data site. The code is delivered by the QPC and executed by the DAP. The last component is the Data Server where the data are stored.

The code shipping works as follows. The QPC receives a request from a client, creates an execution plan, and prepares the code for the distribution to each DAP which includes the retrieval of the code from the code repository. Before the actual query are executed the Java classes are shipped to the DAPs. After the DAP is ready the data processing starts and the results are sent back to the client. Thus, functionality required for the data processing can be dynamically deployed at the data site.

The principle to ship code to the data site and to deploy it automatically is useful for the development of modeling concepts for function shipping. Just the required code is shipped and at the data site a runtime environment or processing component exists able to deploy the code and to process the data. MOCHA is limited to the shipping of Java code and code which is stored in a central code repository. Additionally, a network connection between the central QPC and the DAPs at the remote data site are required to distribute the code. The modeling concepts to be developed should facilitate the function shipping also in cases where the data site is not directly accessible over the internet.

### 3.1.2 Serverless Architecture

A new upcomming trend in Cloud Computing is the concept of serverless architectures [Rob16]. This term was also used before in a different context. In earlier works not related to cloud computing, serverless architectures refer to peer-to-peer architectures

without a central server. Server functions or files are distributed on several clients which communicate with each other [Bol+00; Lee+02].

In the context of Cloud Computing, serverless architectures refer to compute services provided by cloud providers which abstract all server, operation systems, and runtime environment aspects. It runs the developer's piece of code (function) and full-automatically managed the compute resources. The developer just writes the code, but does not deal with the underlying environment which is completely managed by the cloud provider. Obviously, servers are still required but not considered by the developer. The execution of the code is triggered by an event and the customer just have to pay the execution time of the code. This kind of cloud service is also called function as a service (FaaS) [Rob16].

FaaS differs from the already established service model platform as a service (PaaS). PaaS refers to the offer of a runtime environment including hardware, middleware and partially software for a customer's application which used programming languages and tools supported by the cloud provider [Mel+11]. In contrast to FaaS the control of scaling and provisioning instances of the application remains with the customer such as the PaaS AWS Elastic Beanstalk provided by Amazon [Rob16; AWS16b]. FaaS is geared to event-based execution of functions which are automatically scaled to the amount of incoming events.

Several cloud providers provide FaaS such as AWS Lambda [AWS15], Google Cloud Functions [Pla16], Microsoft Azure Functions [Azu16], and IBM Bluemix OpenWhisk [IBM16]. The most popular and one of the first on the market is AWS Lamdba with Amazon API Gateway [AWS15]. The logic tier of a traditional three-tier architecture consisting of the *presentation tier*, e.g., a web client, the *logic tier*, e.g., a web server and the *data tier*, e.g., a database can be formed as a serverless logic tier. AWS Lambda is the compute service running the customer's piece of code, so called Lambda function which cover the required business logic. The function can be uploaded and executed when it is triggered by an event. An event can be for example an incoming HTTP request from the web client. AWS Lambda automatically scales to handle an arbitrary number of incoming request. The Amazon API Gateway is a service to define and manage APIs which deals with HTTP request by web clients. The API Gateway serves as bridge between the presentation tier and the Lambda function.

One of the key features of AWS Lambda in relation to function shipping is the ability to extend other AWS Services with business logic. As shown in Fig. 3.1 AWS Lambda and the Amazon API Gateway can be used for an Amazon Virtual Private Cloud (VPC) integration. The Lambda Function is directly integrated with the data-tier, e.g., a database storage service. The Amazon VPC facilitates to locally separate an area which serves as a private network and is not accessible over the internet [AWS16a]. Especially in case the data contains sensible business information the data have to be secured and

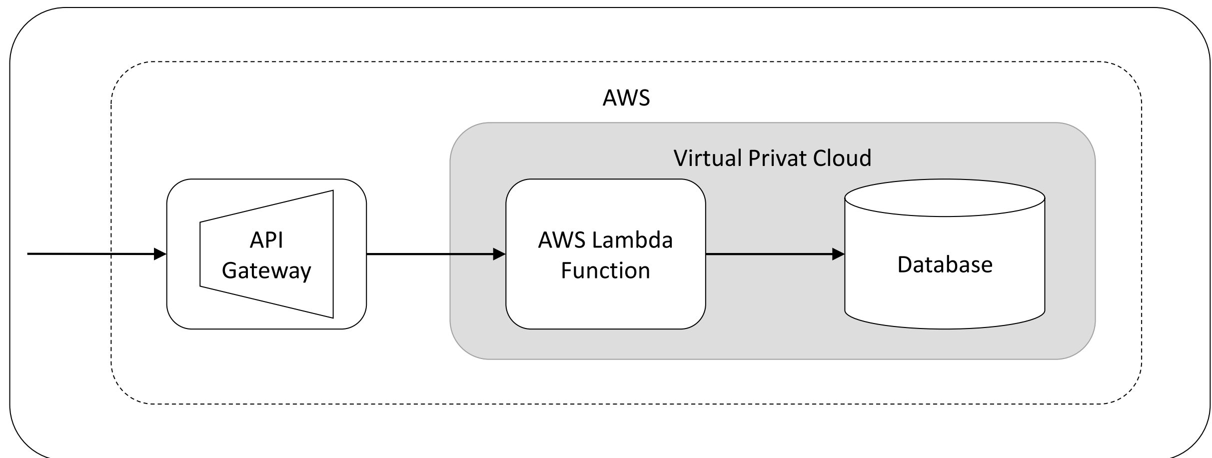



**Figure 3.1:** Examplary architecture with AWS Lambda adopted from [AWS15]

not accessible outside the private network. The data in the database are only accessible through the API Gateway and the Lambda Function.

Two aspects of the above described characteristics of serverless architectures in the context of cloud computing are useful for the development of modeling concepts for function shipping: (1) abstraction of the IT environment and (2) integration with storage services. If a serverless architecture is provided to run the processing logic required for the data, only the function has to be modeled instead of the complete application topology. It can be focused on the main purpose of the modeling concept: the processing logic, the data, and the relationship between them. The integration of the Lambda Function with storage services as provided by Amazon enables function shipping in terms of the execution close to the data. The code has to be uploaded and can run close to the data location. A similar approach is useful in case sensible business data which should be processed must not leave the private network of the company which owns the data. The processing logic has to run in the private network close to the data. To enable the shipping, deployment, and execution of a processing logic near to the data, a general cloud provider independent method is required.

## 3.2 Data Shipping Concepts

In various areas external data are required for data processing. In the following sections different approaches to access or migrate external data are described and evaluated in terms of their usefulness for the modeling concepts. These range from data access patterns in distributed applications to data portability in clouds and concrete mechanism to retrieve external data for business analytics or simulations.

### 3.2.1 Data Access Patterns

Fehling et al. [Feh+14] present patterns for cloud computing. Best practices solutions for cloud application designs are described in a cloud provider- and technology-independent manner. Cloud runtime modules as well as architecture styles and cloud application components are covered. In case of a distributed application the application components can be pooled in multiple tiers. In two-tier and three-tier applications, data can be stored in a separate data tier, which provides data for the business tier. The *data access component* provides a unified data access to different data sources. Data sources can be for example relational databases. It hides the complexity of data access and is responsible for creating, reading, updating, and deleting data elements. Only this component has to be adjusted in case of changes in the data source.

A similar concept is introduced by Fowler [Fow03] for enterprise applications. Like the data access component, a *gateway* encapsulates the access to a external data resource. The gateway serves as wrapper for the API of the data resource. It translates the method calls by the application logic into API specific calls. The same applies for the *proxy* pattern presented by Buschmann et al. [Bus+96]. The proxy is a representation of the component the client wants to communicate with. Hard-coded changes of the client are not required because access control, access optimization, and other additional processing steps are wrapped by the proxy. Such a component is required to enable the assignment of arbitrary data sources to the processing logic, which is not build to access different data sources. It can connect the processing logic to the respective data source.

### 3.2.2 Data Portability in Clouds

A rich research area in terms of cloud computing is portability of applications, services and data in or between clouds [Pet+14]. AWS [AWS16c] provides data shipping via hard drives to the AWS cloud. The hard drive can be ordered, the local data can be transferred, and after completion it is sent back to the data center. This can result in delays and service downtimes, which is often not manageable because of continuous data collection and analysis.

According to Petcu et al. [Pet+14] data portability is achieved if data can be exported from a cloud provider and imported to another. On short term platform-independent data representations are required, but on the long term standardized import and export functionality should be provided to reach data portability. The research effort in terms of data portability is focused on data management of data stored in the cloud environment and the development of abstraction layers and uniform APIs for cloud storage. The migration into cloud storage and between cloud storage of different providers are the

main purpose of standards and services. The connection between applications and data are not considered.

The modeling concepts for data shipping focus more on the data provisioning for the data processing and less on data migration. However, one option for data shipping is to package the data in an archive and to send them to the environment the processing logic is located. This can be realized either by shipping hard drives similar to the AWS service or by transferring the data over the internet. Thus, the data are migrated or copied but specific requirements for data migration and compatibility aspects between different data storage services are not considered in this thesis.

### 3.2.3 Accessing External Data for Business Analytics

There are several providers of business intelligence and analytics cloud platforms, which provide mechanisms to access external data sources for the data analysis. The two business analytics solution Birst [Bir16] and Logic Analytics [Ana16a], which are described in this section just serve as examples to demonstrate the data access functionality. Both provide ETL mechanisms to ship the data to the cloud to facilitate data processing.

Birst [Bir16] provides an Infinite Connectivity Framework to specify connectors to access any data source. Pre-defined connectors to connect data source types that are often used are available, which can be configured and used to extract data. Additionally, real-time queries can be executed on on-premise data sources. A detailed documentation about the Infinite Connectivity Framework is not available.

A similar mechanism is used by Logic Analytics [Ana16a]. Logic Analytics offers a platform for analytic applications, dashboards and reports used to support business decisions. To analyze data from different data sources *Connection* elements are defined to communicate with the data sources to retrieve the data. For this purpose different Connection elements are pre-defined, which can be used. Three Connection elements are distinguished: vendor-specific Connection, generic Connection, and special-purpose Connection. The vendor-specific Connection elements make it simple to configure a connection for example with DB2, MySQL or Google Docs data sources, whereby the necessary drivers are already available. In case of a MySQL Connection element the following attributes have to be specified: server name, database name, access credentials, and a unique ID which identifies the Connection element. With this information the connection string required to establish a connection with the database is automatically created. Generic connection elements are defined for connections for example using

a JDBC driver or ODBC driver. The last category of Connection elements covers connections to web services and pre-defined elements for HTTP, REST, or SOAP-based communications.

To retrieve the data, a temporary data container called *Datalayer* is defined [Ana16b]. After the data are retrieved they are cached in memory or in XML files on the web server. For different kinds of data sources different Datalayers are defined, which often match with the Connection element. Datalayers to retrieve data from CSV, XML, JSON, and Excel files as well as from SQL databases and further more are defined. For a SQL Datalayer the used Connection element can be referenced and a SQL query is defined retrieving the required data from the data source. The different Datalayers can be assigned for example to parts of a report.

A modeling element to define the connection requirements and the scope of the retrieved data is one approach to specify data that should be processed. Additionally, the linking of retrieved data to an element using this data is also required for the development of the modeling concepts to assign the processing logic and the data sources. This procedure is suitable if data should be retrieved from a remote location. It does not cover the shipping of data that are packaged in an archive and the provisioning of the data close to the processing logic.

### 3.2.4 Accessing External Data in Simulation Workflows

Simulation workflows often require data provided by different data sources. The SIMPL (SimTech – Information Management, Processes, and Languages) Framework introduced by Reimann et al. [Rei+11] provides an abstraction for data provisioning activities in simulation workflows and an extension for workflow management systems. They defined a set of generic extraction, transformation, and load operations (ETL) as an extension of the Business Process Execution Language (BPEL) to access arbitrary external data sources through unified logical interfaces. SIMPL defines three main operation: IssueCommand to manipulate data, RetrieveData to query data from a data source, and WriteDataBack to write data from the workflow to a data source.

However, since the external data sources are heterogeneous with respect to access, authentication, and query mechanisms, the unified logical interfaces have to be mapped to the concrete data source and the corresponding mechanisms. Therefore, SIMPL defines four metadata classes to describe this mapping as shown in Fig. 3.2.

A *Data Source* is a system to store and manage data like a file system or a database. Each Data Source contains several *Data Containers*, which represent identifiable datasets within the associated data source, e.g., a file in a file system or a table in a database. A Data Source has a unique *Logical Source Name* and an *Interface Description* containing

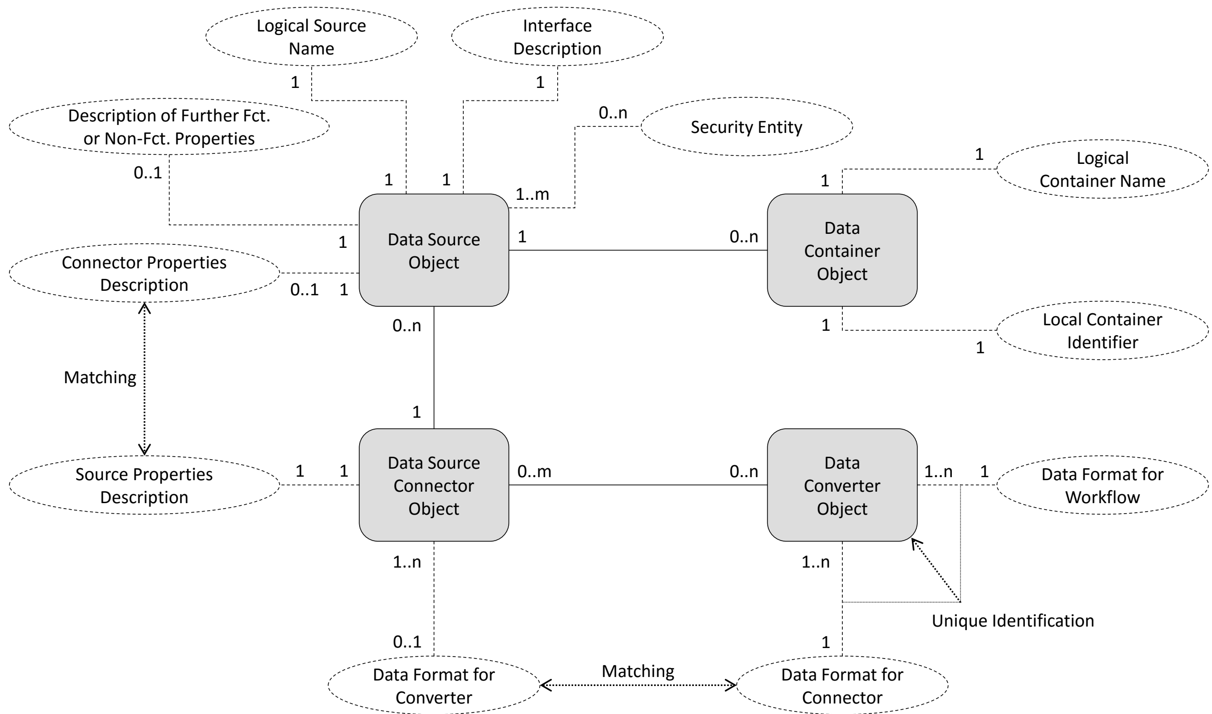



**Figure 3.2:** SIMPL Metadata classes adopted from [Rei+11]

information especially about the data source endpoint. Additionally, further functional and non-functional properties like the expected response time and security entities such as usernames and passwords can be defined for a data source. A Data Container is uniquely identified within the data source by the *Local Container Identifier*. The *Logical Container Name* is used in the workflow to address the data container and is mapped to the Local Container Identifier.

A *Data Source Connector* implements the generic operations for a concrete data source or a collection of data sources, e.g., one connector for all databases using JDBC. The connector establishes a connection to the data source, loads the driver, if necessary, and performs the operation on the data source. For instance, in case of the RetrieveData operation the query will be executed and the result returned. The *Source Properties Description* describes the properties a data source must have to be able to use this connector. These properties have to match with the *Connector Properties Description* associated with a Data Source to find the appropriate connector for a certain data source.

Furthermore, a *Data Format for Converter* description is associated to a Data Source Connector. It specifies the data output format or data input format the connector expects and indicates which *Data Converter* matches. Besides the *Data Format for Connector* description the Data Converter has a *Data Format for Workflow* description. This is the

format in which a workflow delivers data and expects the result back. Both format descriptions together describe the transformation capability of the data converter and uniquely identify the converter.

If an external data source has to execute an operation a *Logical Data Source Descriptor* is passed to the operation as input. This is either a Logical Source Name of a Data Source or a requirement description of functional or non-functional properties, which can be used to find and bind an appropriate Data Source at runtime. The operations IssueCommand and WriteDataBack deliver a notification of success or failure to the workflow, whereas the RetrieveData operation returns the requested data. These data may be stored in a variable specified by the workflow engine.

The focus of SIMPL is on extending the workflow activities with abstract and generic data management operations. Reimann et al. [Rei+14] presents a solution for the data management in the cloud which integrates the SIMPL operations in TOSCA. With TOSCA the simulation software can be defined and deployed in the cloud. For the data provisioning, i.e., the data shipping to the simulation software, the SIMPL data management operations are integrated in a TOSCA Plan to transfer the data to the simulation software. In the same way the data are written back after the simulation. Thus, the TOSCA Plans are used to fulfill the data shipping to the application by means of workflow operations based on SIMPL. For this, the workflow engine of the TOSCA runtime environment is extended by the operations and the SIMPL framework is plugged in.

Although this approach enables data shipping, the data are not part of the Topology Template in TOSCA. In terms of the data provisioning the operations and not the data itself are modeled. However, the SIMPL metadata classes shown in Fig. 3.2 presents components to define the data as well as the connection to a data source, which serve as basis for the modeling concepts in Chapter 5.

### 3.2.5 Decoupling of Data Flow and Control Flow in Service Compositions

Hahn et al. [Hah+16] introduce the vision of a novel Transparent Data Exchange (TraDE) middleware to support the data exchange between choreographed services. Thereby, the data flow is decoupled from the control flow to avoid unnecessary data exchange between the participating services of the choreography. The data should be passed only to the participants which require the data.

Based on the traditional business process management life cycle phases a data-aware service choreography management life cycle is introduced in which new TraDE methods

are applied. An explicit data model and data flow is created in the *Modeling* phase. It defines the data exchange between the participants. The data model and data flow are transformed in abstract workflow models for each participant. In the following *Refinement* phase the abstract workflow models are refined into executable models which are packaged and deployed to the workflow middleware in the *Deployment* phase. After that the choreography is executed and monitored. During the execution the participants communicate to exchange data or to trigger the functions of other participants. When the vision of the TraDE Middelware is realized, it will among others integrate data shipping mechanism and enable the use of heterogeneous data sources realized by a plugin of the mentioned-above SIMPL framework.

It is shown that the efficient data exchange is getting more important in the industry, especially in context of Big Data and the Internet of Things [Hah+16]. The capabilities described for the TraDE middleware are also required to implement the data shipping modeling concepts developed in this work, although service compositions are not considered. The control flow in terms of the trigger for the execution of a specific processing logic is independent of the definition of the data flow. The data flow defines the assignment between processing logic and data resources which are required in case the processing logic is invoked. Instead of workflow models defining the data model, abstract concepts independent of their implementation as well as options to realize them in TOSCA are the main subject of this thesis.

# 4 Data Shipping and Function Shipping Use Cases

A concrete modeling scenario with different data shipping and function shipping use cases is used to illustrate the problem and the motivation for the modeling concepts. It should serve as example for the developed modeling concepts and TOSCA realization options. This supports the understanding of the applicability of each concept and option.

## 4.1 Modeling Scenario for Data Shipping and Function Shipping

The problem and the motivation of this thesis are illustrated by the scenario shown in Fig. 4.1. The modeling scenario presents a possible situation, which should be solved by data shipping or function shipping. Two main roles are distinguished in this scenario: *processing logic owner* and *data owner*. The roles can be fulfilled by a single person, a group of people, or an organization. At the processing logic owner's site a processing logic is available, which is required for a specific data processing task. The data, which should be processed are stored at the data owner's site. Either the data owner or the processing logic owner has a runtime environment to deploy and run the processing logic

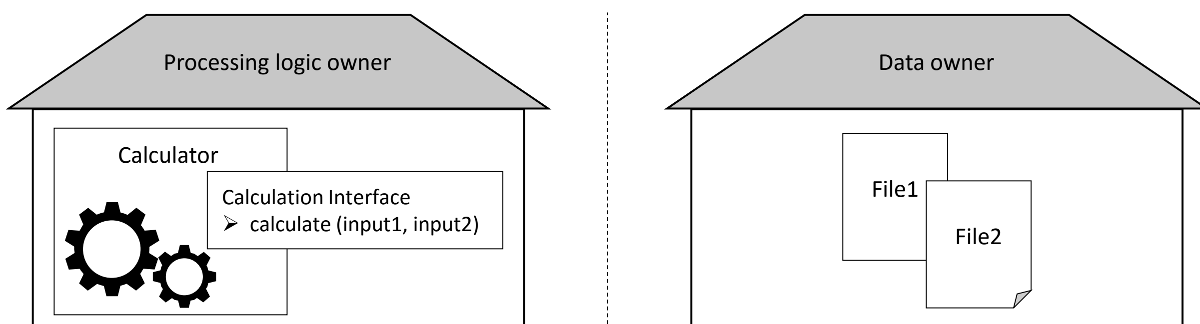



**Figure 4.1:** Modeling scenario

as well as to store and query the data. This depends on the shipping use case. A further assumption for this modeling scenario is that they are connected via the internet.

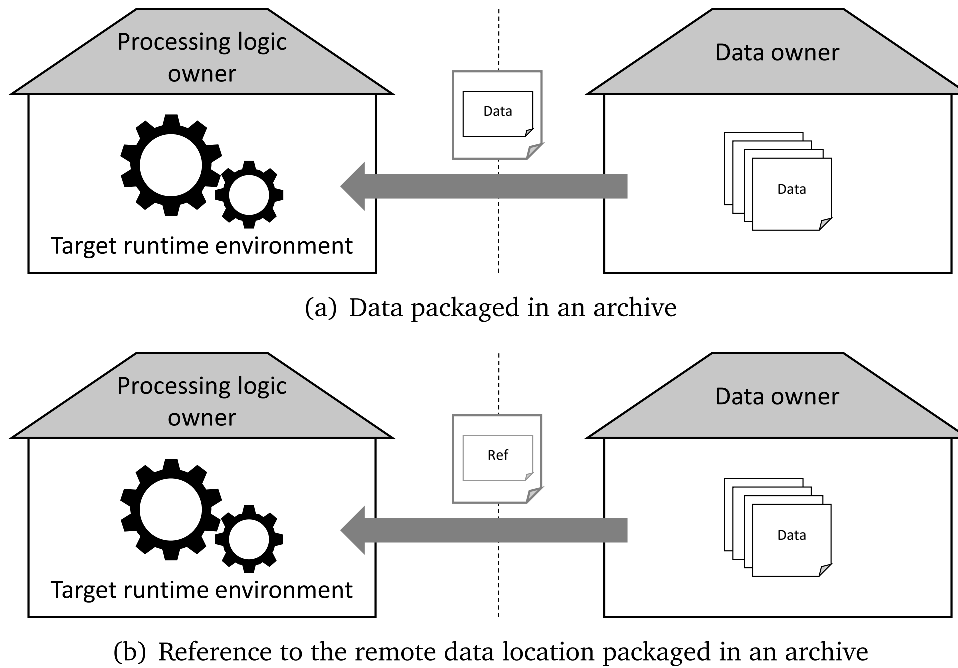
In this modeling scenario, the processing logic owner owns a processing logic *Calculator*. The processing logic provides the operation *calculate* via the interface *CalculationInterface*. To execute this operation two input parameters of type "string" are required. On the side of the data owner data are available, which have to be processed by the operation *calculate*. For instance, the operation calculate could compare sensor values like temperature and vibration of a production machine to analyze the optimal maintenance time. Processing logic and data should be brought together for the data processing either on the processing logic owner's site or the data owner's site depending on the shipping scenario.

The process of data shipping as well as function shipping is divided in modeling time, provisioning time and runtime. The modeling time is the point in time the model as well as other artifacts are defined and packaged in an archive. After the archive is transmitted to the target runtime environment the data are provisioned or the processing logic is deployed and instantiated during the provisioning time. Thus, it is ready for the execution during runtime. Such distinction is important to make clear which data shipping or, respectively, function shipping use case is suitable in a specific situation and at what point in time the data or processing logic is available at the target runtime environment. It is described more precisely for the data shipping and function shipping use cases in the following sections.

## 4.2 Use Cases for Data Shipping

There are two alternative ways of linking data and processing logic. First, data shipping and second, function shipping. As defined in Section 2.2, in case of data shipping the data are packaged, transmitted, and provided for the processing logic already deployed in the target runtime environment. Fig. 4.2 illustrates the two main data shipping use cases: (a) data are packaged in an archive and (b) reference to remote data location is packaged in an archive.

In case (a), the actual data are stored in the archive containing the whole service model and are shipped to the processing logic owner at provisioning time. In the target runtime environment the processing logic required for the data processing is already available and the data are deployed for the processing and assigned to the processing logic. For this use case the data have to be already available. For instance, historical production data over the last two years have to be analyzed. In this case all relevant data for




(a) Data packaged in an archive

(b) Reference to the remote data location packaged in an archive

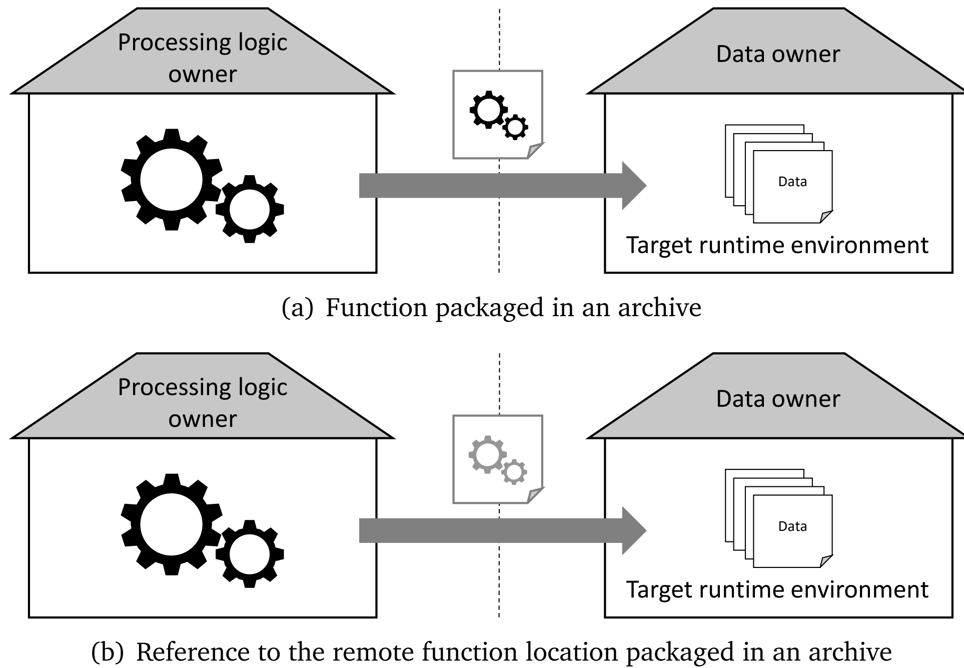
**Figure 4.2:** Use cases for data shipping

the processing are already available at modeling time. The data can be packaged and shipped together with the topology and other artifacts to the processing logic owner.

In case (b), a reference to the remote data location is shipped in an archive together with the topology and artifacts to retrieve the data at a later time. That means, the data remain at a remote location chosen by the data owner or the data are available at the remote location as soon as they are captured. At provisioning time the connection between the processing logic and the data location is established. At latest during runtime the data have to be retrieved. For instance, current weather data are required continuously, but future weather data are not available at modeling time. Thus, the data can not be packaged in an archive and shipped to the processing logic owner. In this case only a reference to the location at the data owner's site, where the data will be made available can be shipped. During the runtime up-to-date weather data can be retrieved and processed at the processing logic owner's site.

## 4.3 Use Cases for Function Shipping

The other alternative to link processing logic and data is function shipping. Instead of the data, the processing logic for the data processing is shipped to the data owner's site. Function shipping is an alternative approach to data shipping which is useful in different




(a) Function packaged in an archive

(b) Reference to the remote function location packaged in an archive

**Figure 4.3:** Use cases for function shipping

situations. For instance, in case a large amount of data should be processed and data shipping takes too much time or is too expensive or in case the data contain sensitive data which are not allowed to leave the company or data center, function shipping can be used.

The two main function shipping use cases are the shipping of the actual function in an archive and the shipping of a reference to the remote function location, illustrated in Fig. 4.3. In case (a) the processing logic is stored in an archive containing the whole service model and shipped to the data owner. In the target runtime environment the processing logic can be deployed and linked to the data, which are already available at provisioning time. In case (b) the processing logic remains at a remote location and just a reference to the actual processing logic is shipped. For the instantiation in the target runtime environment the processing logic can be retrieved. If the processing logic artifact is not available at modeling time, a reference to the remote location can be shipped and the processing logic can be retrieved as soon as they are available.

All modeling concepts, discussed in Chapter 5, relate to the modeling scenario illustrated in Fig. 4.1. It serves as example for the modeling to make clear how each modeling concept or TOSCA realization option can be used for a concrete scenario.

# 5 Modeling Concepts for Data Shipping and Function Shipping

The following chapter discusses eight modeling concepts enabling data and function shipping as well as options to realize the concepts in TOSCA. Some of these concepts are exclusively applicable for data shipping, or function shipping respectively, some of them for both. Applicability is indicated with each concept. The decision which modeling concept fits best mainly depends on

- the use case and the granularity of modeled information in particular
- the modeler's knowledge about the processing logic and the data
- and the runtime environment's capabilities.

The runtime environment's capabilities include for example the automated determination of the right data resource or processing logic required for the data processing, the establishment of a connection between the processing logic and the data resources, and the determination and execution of transformation operations on data to make them compatible with the processing logic. The granularity of explicitly modeled information determines the requirements for the modeler's knowledge and runtime capabilities. The finer the granularity, the higher the knowledge and the lower the runtime capabilities must be. For instance, a topology that describes a data resource is used for data shipping but the required processing logic is not explicit modeled. In this case the modeler does not require knowledge about the processing logic, but the runtime environment must be able to determine the right processing logic, to establish a connection, and to possibly execute data transformations. Thus, the requirements for the modeler's knowledge are low but for the runtime environment's capabilities are high.

An overview of the different modeling concepts and TOSCA realization options is shown in Fig. 5.1. Eight different concepts are considered, each representing an abstract modeling approach, which can be used for data shipping, function shipping, or both. The various options show how the abstract concepts can be realized with TOSCA. In some cases more than one option exists to implement the concept in TOSCA, depending on the utilized TOSCA language elements.

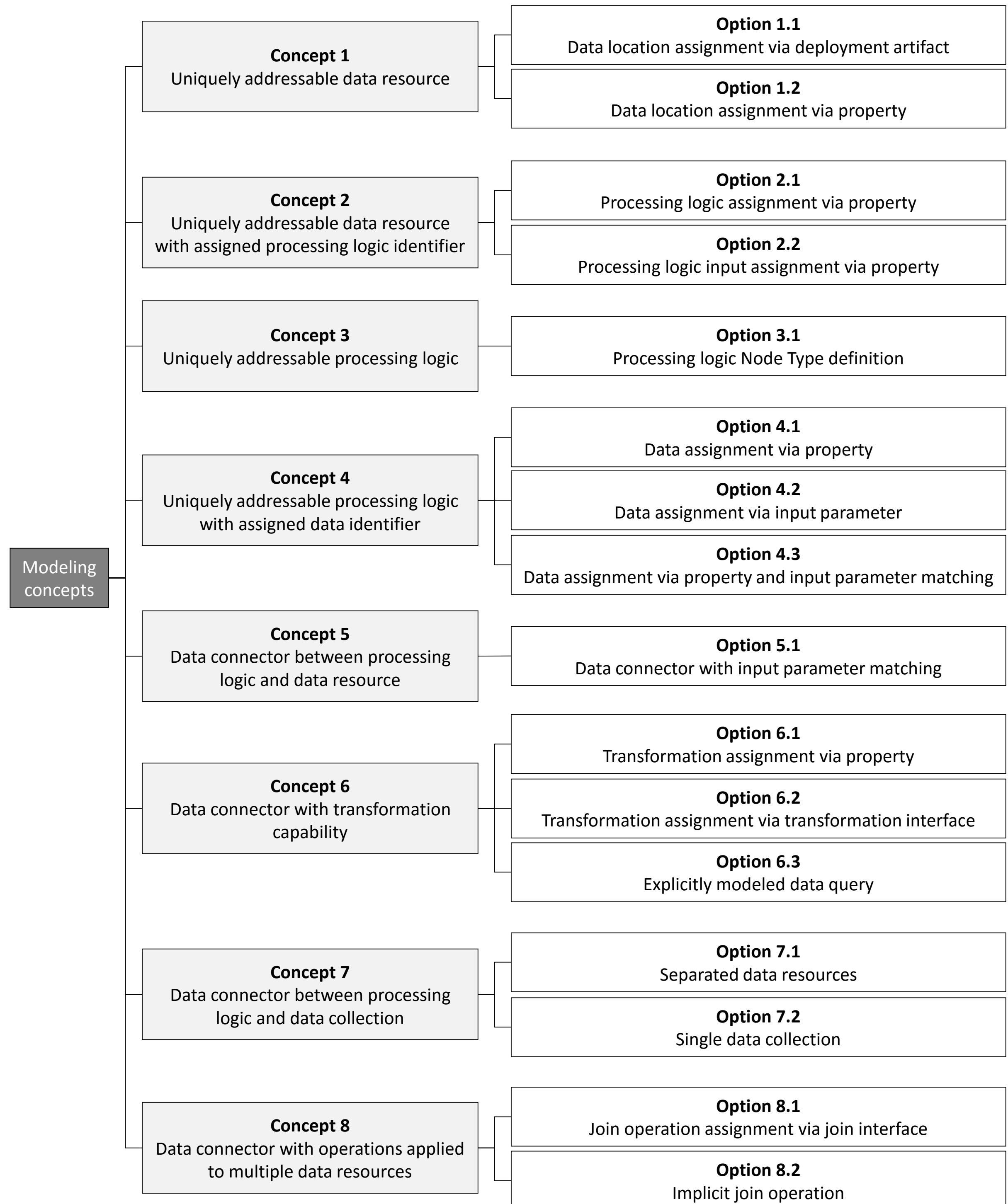



**Figure 5.1:** Overview of the modeling concepts and the related TOSCA realization options

Each modeling concept is described following a common structure. First, the concept is described in its four main characteristics, secondly, one or more sections describe the TOSCA realization option(s) related to the concept.

The first part, the concept description, is structured as follows:

- *Context* – Explanation of the circumstances and environment leading to the problem. Other concepts from the eight concepts may be referenced here to differentiate them.
- *Problem* – Short description of the problem which should be solved by this concept. It is expressed as a question.
- *Solution* – Description of the concept and how it solves the problem. Additionally, this part refers to the data and function shipping use cases the concept is suitable for.
- *Implications* – Explanation of the impacts of the usage of this concept.

In the second part, each of the TOSCA realization options of the respective concept is introduced in a separate section structured in four parts: Firstly, a detailed description of the TOSCA Service Template is given. The description is independent of the data and function shipping use cases. Secondly, the result part describes how the given data and function shipping use cases can be realized by this option in TOSCA. Thirdly, the assumptions made with regard to the modeler's knowledge and runtime environment's capabilities, as well as the restrictions of this option are explained. Finally, the extensions part addresses required TOSCA language extensions to enable the modeling and model processing and summarizes the domain specific elements used within this option. The illustrated TOSCA Service Template of each option bases on the assumption that the processing logic runs in the TOSCA runtime environment. As seen in Section 3.1.2, serverless architectures enable to abstract the complete topology stack and to focus only on the function used to apply the required business logic. It is assumed that this concept applies for the environment which runs the processing logic.

The abstract modeling concepts are not TOSCA-based. The basic elements the modeling concepts consist of are *Processing Logic*, *Data Resource*, *Data Collection*, and *Data Connector*. For a better understanding of the eight abstract modeling concepts the used elements in these models are defined in the following:

*Processing Logic:* The Processing Logic element represents an algorithm, function, or any other logic, which is used for data processing and requires data as input. A processing logic is owned by the processing logic owner.

*Data Resource* and *Data Collection:* A Data Resource represents data, which has an unique identifier and a location. This can be for example a file or table. A Data Collection can contain several Data Resources. The Data Resources are subelements of the Data Collection and can be addressed within the Data Collection. The Data Collection itself is also an addressable entity. Examples for a Data Collection are file directories or relational databases.

*Data Connector:* The Data Connector represents the connection between a Processing Logic and a Data Resource or Data Collection. Only 1:1 relationships can be modeled. Properties defining the connection between the processing logic and the data are encapsulated by the Data Connector.

Not all basic elements are used in every concept. Depending on the concept only a subset of basic elements are used. Additionally, attributes are assigned to the basic elements, which specify the properties of the elements. The used attributes are explained for each concept.

## 5.1 Concept 1: Uniquely Addressable Data Resource

The focus of this concept is on modeling the data resource, which should be processed.

*Context*

A data resource is available, which should be processed by a processing logic. This data resource has a location where it is physically stored. The required processing logic is either not known in advance or just one processing logic is available. In the first case the appropriate processing logic is determined by an external mechanism or person, in the second case the allocation is implicitly given because only one processing logic is available.

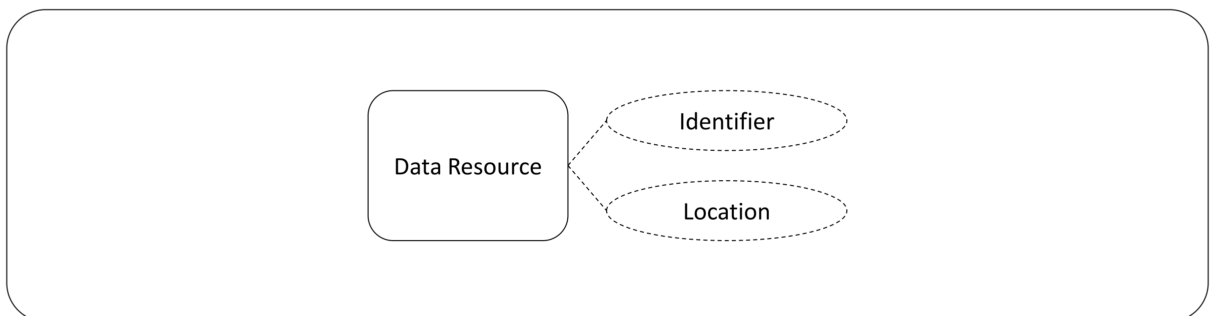



**Figure 5.2:** Concept 1: Uniquely addressable data resource

*Problem*

How can a data resource, which should be processed by a processing logic, be uniquely addressed?

*Solution*

Fig. 5.2 shows how a data resource can be modeled. A *Data Resource* entity is uniquely addressable by an *Identifier* and the physical location of the represented data resource is indicated by the *Location*. The *Location* can refer to local as well as remote data. Therefore, this concept can be used for shipping the actual data packaged in an archive as well as shipping of a reference to a remote location. As the processing logic is not part of this model, the concept is not suitable for function shipping.

*Implications*

Using this modeling concept, the location where the data are physically stored can be referenced. However, an allocation between the data resource and the processing logic cannot be specified. If it is necessary to integrate the allocation into the model, one of the other concepts, except of Concept 3 and Concept 4, can be used. Furthermore, other characteristics of the data such as the data format are not added to the *Data Resource*. Concept 6 demonstrates how such information can be associated to a *Data Resource*.

### 5.1.1 Option 1.1: TOSCA Realization with Data Location Assignment via Deployment Artifact

In this TOSCA option, illustrated in Fig. 5.3, the *Data Resource* entity of the concept is defined by a Node Type *DataResource*. The ID attribute of a Node Template serves as the *Identifier* and by means of a Deployment Artifact, the actual data storage location can be referenced.

*Description*

Listing 5.1 depicts the part of the Service Template specifying the Node Type and Node Template in pseudo-XML. A Node Type *DataResource* is defined (lines 1 to 3) and a Node Template *DataResourceFile* of this type is specified (lines 5 to 10). The Deployment Artifact *Data* references an Artifact Type *FileArtifact* and an Artifact Template *FileContent*, which is of type *FileArtifact*. The Artifact Template, shown in Listing 5.2, is required to indicate the location of the data. The placeholder *address* in line 8 can be replaced by an relative URI, in case it points to data in the CSAR containing the Service Template, or any other address if the data has to be retrieved from a remote location.

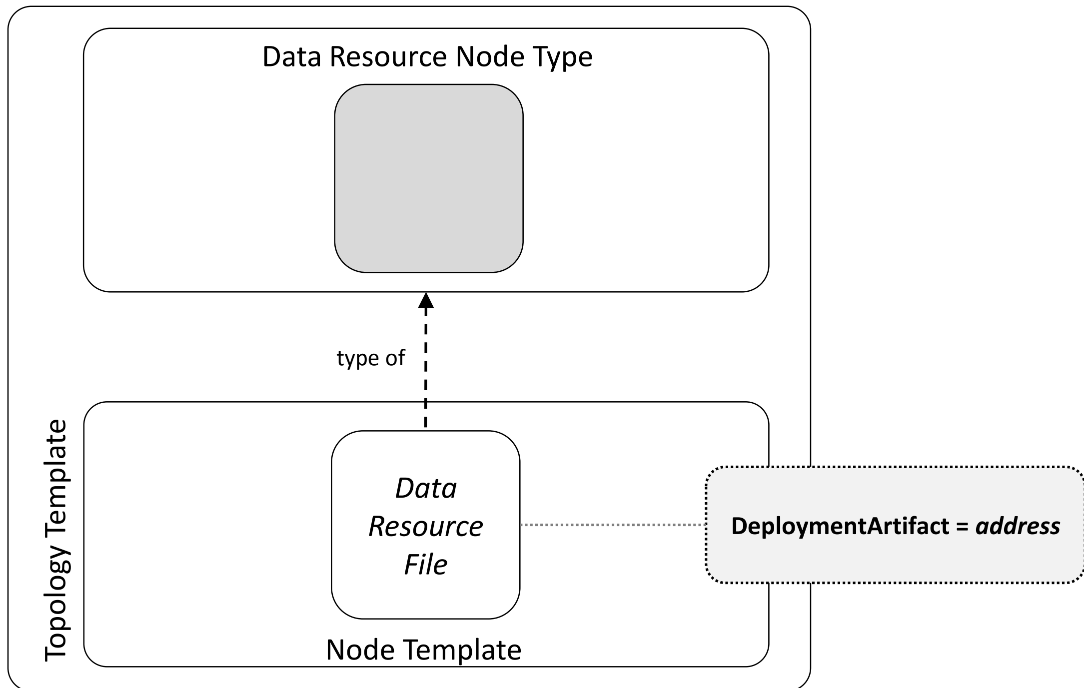



**Figure 5.3:** Option 1.1: TOSCA realization with data location assignment via deployment artifact

**Listing 5.1** Option 1.1: Node Type definition *DataResource* and Node Template *DataResourceFile*

```
1 <NodeType name="DataResource">
2   ...
3 </NodeType>
4 ...
5 <NodeTemplate id="DataResourceFile" type="DataResource">
6   ...
7   <DeploymentArtifacts>
8     <DeployomentArtifact name="Data" artifactType="FileArtifact"
          artifactRef="FileContent"/>
9   </DeploymentArtifacts>
10 </NodeTemplate>
```

**Listing 5.2** Option 1.1: Artifact Type definition *FileArtifact* and Artifact Template *FileContent*

```
1 <ArtifactType name="FileArtifact">
2   ...
3 </ArtifactType>
4 ...
5 <ArtifactTemplate id="FileContent" type="FileArtifact">
6   ...
7   <ArtifactReferences>
8     <ArtifactReference reference="address"/>
9   </ArtifactReferences>
10 </ArtifactTemplate>
```

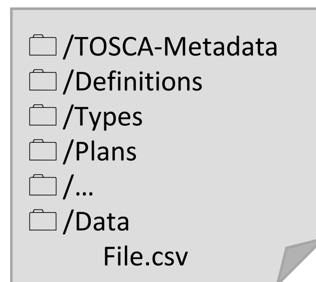



**Figure 5.4:** Exemplary structure of an CSAR containing data

*Result*

This TOSCA realization can be applied for both data shipping use cases illustrated in Fig. 4.2:

(a) Data packaged in an archive: the data are stored in the CSAR file containing the Service Template. The CSAR is transferred to the target TOSCA runtime environment where the processing logic is already deployed. In this case the Artifact Template *FileContent* references an relative URI, which is interpreted relative to the root directory of the CSAR. An exemplary structure of a CSAR is shown in Fig. 5.4. The data, represented by the Node Template *DataResourceFile*, are stored in the subdirectory *Data* of the CSAR. The corresponding relative URI in the Artifact Template is "Data/File.csv". That way the data are shipped in an archive.

(b) Reference to the remote data location packaged in an archive: the data remain on a remote data location and the Artifact Template *FileContent* references the address of the remote location. If the data are stored on an FTP-server, the address could be for example “ftp://ftp.example.com/File.csv”. The TOSCA Service Template is sent within a CSAR to the target runtime environment where the processing logic is deployed. The data can be retrieved at a later time when they are needed.

*Assumptions and Restrictions*

This option is restricted to the data shipping use cases. An explicit modeling of the relationship between the data and the processing logic is not intended. Concept 2 shows an extension to specify the required processing logic. In order to model the data shipping as shown for this option three assumptions are made:

- The processing logic is already deployed in the target runtime environment.
- Either an external mechanism or a person can determine the right processing logic, in case several processing logics run in the target runtime environment, or the allocation is implicitly given because just one processing logic is available.
- There is no need for further information about data format, data semantic, or data transformation operations to enable the mapping between data and processing logic and the accurate processing of the data. Either the TOSCA runtime environment or the processing logic can handle potential adaptations.

*Extensions*

The above-proposed option can be realized without any extensions of the TOSCA meta-model. However, in order to process this model, the target runtime environment has to be able to interpret the domain specific elements. A Node Template of type *DataResource* has a Deployment Artifact, which references data contained in the subdirectory *Data* in the CSAR or a remote data location. The target runtime environment has to know how to connect the referenced data to the processing logic appropriately.

### 5.1.2 Option 1.2: TOSCA Realization with Data Location Assignment via Property

As in Option 1.1, the data resource is defined by a Node Type *DataResource*. Instead of a Deployment Artifact the property *Location* is defined by the Node Type to address the location of the data. The model of this TOSCA realization option is depicted in Fig. 5.5.

*Description*

The Node Type *DataResource*, shown in Listing 5.3 in pseudo-XML, has a property *Location* defined in Listing 5.4. The property *Location* refers to the address where the data are physically stored. Listing 5.5 depicts in pseudo-XML the Node Template *DataResourceFile* of type *DataResource*. In this example *address* (line 5) is used as placeholder and can be replaced by an relative URI or any other address. For each Node Template of type *DataResource* one *Location* property is declared.

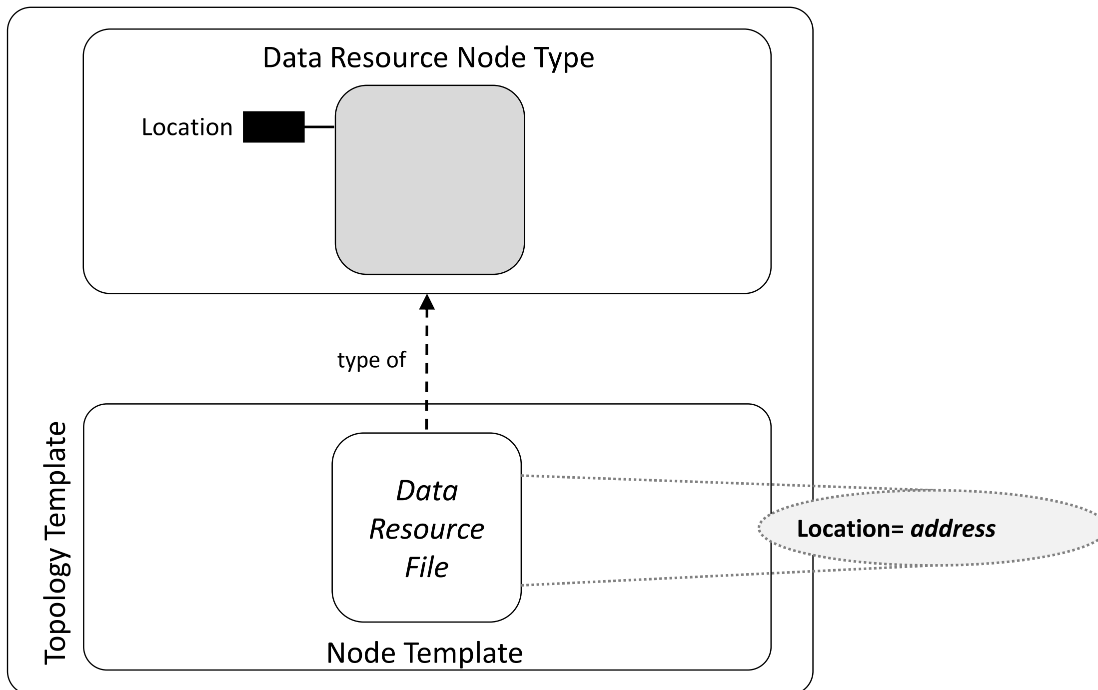



**Figure 5.5:** Option 1.2: TOSCA realization with data location assignment via property

**Listing 5.3** Node Type definition *DataResource* with properties

```
1 <NodeType name="DataResource">
2   ...
3   <PropertiesDefinition element="DataResourceProperties"/>
4 </NodeType>
```

*Result*

As well as Option 1.1, this TOSCA realization option can be used for both, the shipping of the data contained in an archive and the shipping of a reference to a remote data location.

(a) Data packaged in an archive: Fig. 5.4 illustrates an exemplary structure of the CSAR containing the actual data and the Service Template. In this case the property *Location* of the Node Template *DataResourceFile* references the relative URI "Data/File.csv". That way the data are packaged in an archive and shipped.

(b) Reference to the remote data location packaged in an archive: the data are not packaged in the CSAR and the property *Location* references the address of the remote location. The remote location could be for example an FTP-server with the address “urlftp://ftp.example.com/File.csv”. The CSAR without the data is sent to the target runtime environment and the data can be retrieved later.

**Listing 5.4** *Location* property definition for Node Type *DataResource*

```
1 <xs:element name="DataResourceProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="Location">
5         <xs:complexType>
6           <xs:attribute name="ref" type="xs:string"/>
7         </xs:complexType>
8       </xs:element>
9     </xs:sequence>
10  </xs:complexType>
11 </xs:element>
```

**Listing 5.5** Node Template *DataResourceFile* with property *Location*

```
1 <NodeTemplate id="DataResourceFile" name="Data Resource File" type="DataResource">
2   ...
3   <Properties>
4     <DataResourceProperties>
5       <Location ref="address"/>
6     </DataResourceProperties>
7   </Properties>
8 </NodeTemplate>
```

*Assumptions and Restrictions*

This TOSCA realization option is based on the same assumptions and restrictions made for Option 1.1. The processing logic is already available in the target runtime environment, but is not part of the model. The assignment of data and processing logic is done by the runtime environment or an external mechanism, in case more than one processing logic is available. Further adaptations, e.g., data format or data transformation operations are also made by the runtime environment or the processing logic itself.

*Extensions*

For this option a TOSCA metamodel extension is not required. However, the defined Node Type *DataResource* with the related property *Location* has to be correctly interpreted by the target runtime environment, i.e., the Node Template of type *DataResource* represents data that can be retrieved from the address specified by the property *Location*. The target runtime environment has to connect the data to the right processing logic.

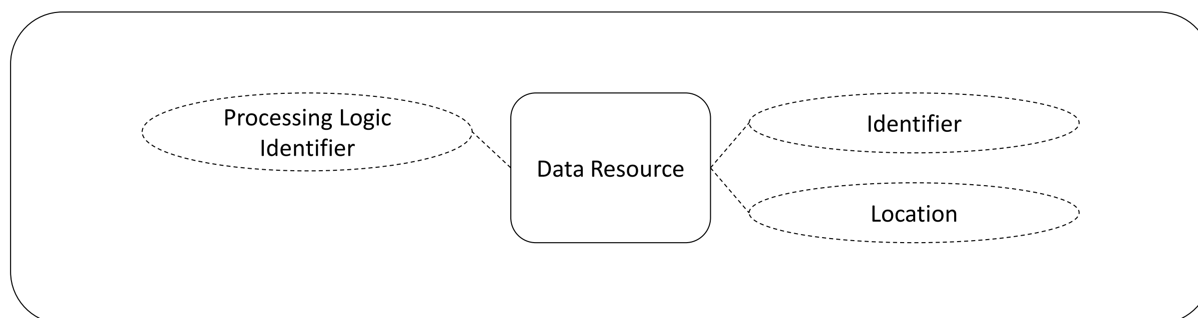



**Figure 5.6:** Concept 2: Uniquely addressable data resource with assigned processing logic identifier

## 5.2 Concept 2: Uniquely Addressable Data Resource with Assigned Processing Logic Identifier

The focus of this concept is on modeling the data resource, which should be processed, similar to Concept 1. Additionally, the assignment of the necessary processing logic to the data resource is considered.

*Context*

The data resource owned by the data owner has to be processed by a specific processing logic of the processing logic owner. In case that more than one processing logic is available and the runtime environment cannot determine the right processing logic for the data processing, the modeler has to specify the mapping between data and processing logic. Concept 1 does not facilitate to explicitly determine which processing logic should be used to process the data resource.

*Problem*

How can the modeler specify which processing logic should process the data resource?

*Solution*

Fig. 5.6 shows how the assignment of a processing logic to a data resource can be modeled. A *Data Resource* is uniquely addressable by an *Identifier* and the physical location of the represented data is indicated by the *Location* attribute. The *Processing Logic Identifier* associated with the *Data Resource* indicates the mapping between the data resource and the processing logic, which should be invoked with the data. The *Processing Logic Identifier* refers to the ID of the respective processing logic. This solves the problem on how the modeler can specify which processing logic should be used for a data resource. The *Location* can reference a local or a remote data location. Thus, the concept can be used for both data shipping use cases. Since the *Processing Logic Identifier*

contains the ID of the processing logic and not the location of the actual deployment artifact, this concept is not suitable for function shipping. The ID of the processing logic can be processed by the runtime environment in which this processing logic is available.

*Implications*

This concept enables the mapping of a processing logic to a data resource and the determination of the data storage location. The concept is not applicable for function shipping, for which one of the other concepts (except Concept 1) can be used. How additional information can be added to the data resource is elucidated in Concept 6.

### 5.2.1 Option 2.1: TOSCA Realization with Processing Logic Assignment via Property

In this option, depicted in Fig. 5.7, a data resource is defined by a Node Type *DataResource* with a property *Location* referencing the actual data location. The property *ProcLogicID* determines the ID of a Node Template representing the processing logic which should process the data and which is deployed in the target runtime environment.

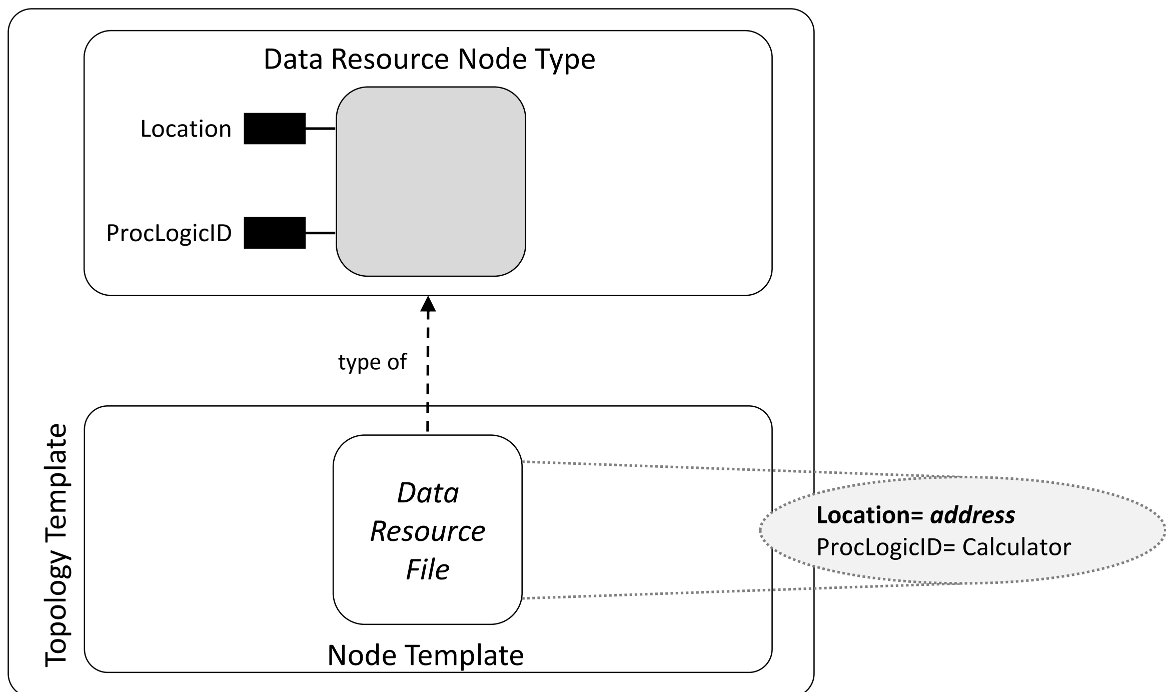



**Figure 5.7:** Option 2.1: TOSCA realization with processing logic assignment via property

**Listing 5.6** Option 2.1: Properties definition for Node Type *DataResource*

```
1 <xs:element name="DataResourceProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="Location">
5         <xs:complexType>
6           <xs:attribute name="ref" type="xs:string"/>
7         </xs:complexType>
8       </xs:element>
9       <xs:element name="ProcLogicID" type="xs:string" maxOccurs="unbounded"/>
10    </xs:sequence>
11  </xs:complexType>
12 </xs:element>
```

**Listing 5.7** Option 2.1: Node Template *DataResourceFile*

```
1 <NodeTemplate id="DataResourceFile" name="Data Resource File" type="DataResource">
2   ...
3   <Properties>
4     <DataResourceProperties>
5       <Location ref="address"/>
6       <ProcLogicID>Calculator</ProcLogicID>
7     </DataResourceProperties>
8   </Properties>
9 </NodeTemplate>
```

*Description*

Listing 5.3 illustrates the Node Type *DataResource* definition in pseudo-XML. The Node Type references an externally defined set of properties shown in Listing 5.6. For the property *Location* an attribute of type "string" is defined and should be utilized to identify the location of the data. The property *ProcLogicID* of type "string" should be used to determine the processing logic required for this data resource. This property can appear multiple times because the attribute *maxOccurs* is set to "unbounded" (line 9). Due to this setting, as much processing logics as required can be assigned to one data resource. In this example the *ProcLogicID* of the Node Template *DataResourceFile*, defined in Listing 5.7, refers to the processing logic *Calculator*, which has to process the data. The placeholder *address* (line 5) is replaced by a concrete relative URI or any other address depending on whether it is used for the shipping of the actual data or a reference to a remote data location in an archive.

*Result*

This TOSCA realization can be applied for both data shipping use cases illustrated in Fig. 4.2:

(a) Data packaged in an archive: the data are stored in the CSAR file, which is transferred to the target runtime environment where the processing logic is deployed. Therefore, the *Location* property contains the data path in the CSAR. An example is shown in Fig. 5.4. In this case the value of the *Location* property is "Data/File.csv". That way, the actual data are shipped in an archive and can be processed by the referenced processing logic.

(b) Reference to the remote data location packaged in an archive: the data remain at a remote data location and the property *Location* references the address of the remote location. The TOSCA Service Template is packaged in a CSAR and sent to the target runtime environment where the processing logic is already deployed. The data can be retrieved at the latest during runtime.

*Assumptions and restrictions*

As mentioned above, a data resource can be linked with certain processing logics but it cannot address a certain interface, operation, or input parameter of the processing logic. If the assigned processing logic provides several interfaces, operations, or input parameters it is not possible to distinguish between them when assigning a processing logic to a data resource. Either this modeling variant can just be used in case each processing logic only provides one interface with one operation requiring one input parameter or the runtime environment respectively the processing logic can handle it. In case the precise addressing of the input parameter is necessary, Option 2.2 can be used. In the example shown in Fig. 5.7 just one processing logic is associated with the data resource. Due to the properties definition in Listing 5.6 it is possible to associate more than one processing logic with one data resource, but the semantic of multiple links is not defined. For instance, it could imply either the assigned processing logics should process the data simultaneously or sequentially.

In order to model the data shipping in this way, three assumptions are made:

- The referenced processing logic(s) is/are already deployed in the target runtime environment and the ID(s) of the Node Template(s) representing the processing logic(s) is/are unique in this environment.
- The ID of the Node Template(s) representing the processing logic(s) in the target runtime environment is/are known. This is needed to specify the right processing logic(s) for the data resource.
- There is no need for further information about data format, data semantic, or data transformation operations to enable the mapping between data and processing logic and the accurate processing of the data. Either the TOSCA runtime environment or the indicated processing logic can handle potential adaptations.

*Extensions*

To realize this option no extensions of the TOSCA metamodel are required. However, the target environment has to be able to interpret the domain specific elements. A Node Type *Data Resource* has a property *Location*, which refers to the actual data contained in the subdirectory *Data* in the CSAR or available at a remote data location. The property *ProcLogicID* indicates the already deployed processing logic in the target runtime environment. The target runtime environment has to know how to connect the data resource to the concrete input parameter of the operation provided by the processing logic. In case the data remains at a remote location, they have to be retrieved at the latest during runtime.

### 5.2.2 Option 2.2: TOSCA Realization with Processing Logic Input Parameter Assignment via Property

As in Option 2.1, the data resource is defined by a Node Type *DataResource* with a property *ProcLogicID* referencing the processing logic deployed in the target runtime environment, which should process the data. In addition to the processing logic ID the exact interface, operation, and the input parameter of the operation, which should use

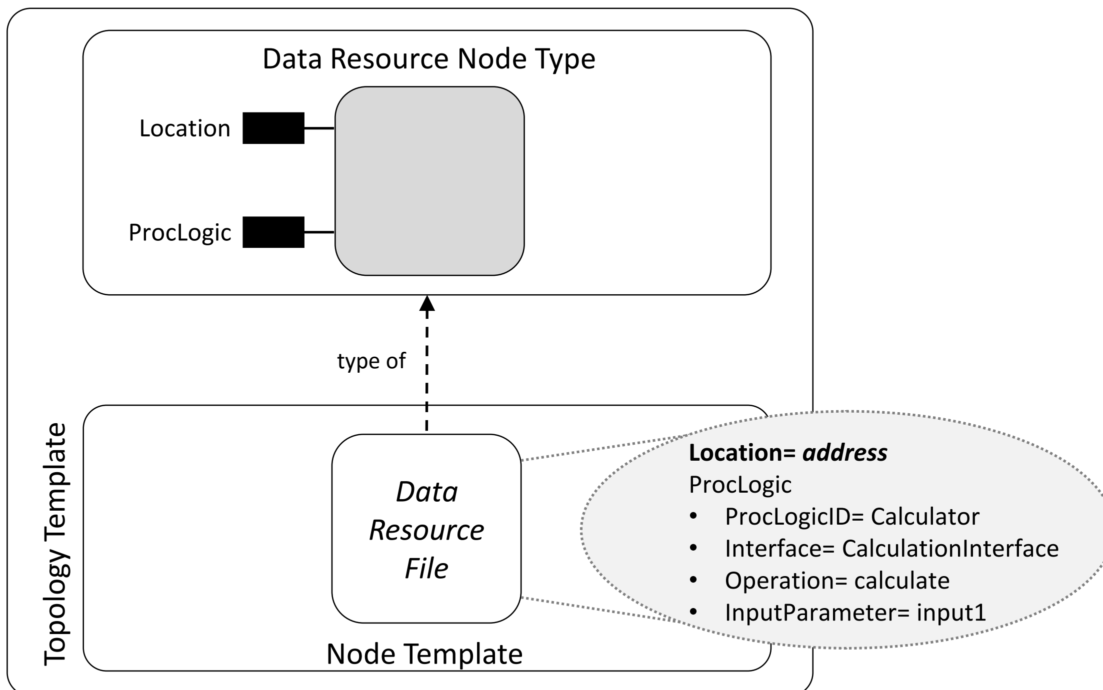



**Figure 5.8:** Option 2.2: TOSCA realization with processing logic input parameter assignment via property

**Listing 5.8** Option 2.2: Properties definition for Node Type *DataResource*

```
1 <xs:element name="DataResourceProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="Location">
5         <xs:complexType>
6           <xs:attribute name="ref" type="xs:string"/>
7         </xs:complexType>
8       </xs:element>
9       <xs:element name="ProcLogic" maxOccurs="unbounded">
10        <xs:complexType>
11          <xs:sequence>
12            <xs:element name="ProcLogicID" type="xs:string"/>
13            <xs:element name="Interface" type="xs:string"/>
14            <xs:element name="Operation" type="xs:string"/>
15            <xs:element name="InputParameter" type="xs:string"/>
16          </xs:sequence>
17        </xs:complexType>
18      </xs:element>
19    </xs:sequence>
20  </xs:complexeType>
21 </xs:element>
```

this data resource as input is specified. Fig. 5.8 illustrates how a corresponding Service Template can look like.

*Description*

The Node Type *DataResource* is depicted in Listing 5.3. Compared to Option 2.1, additional properties are defined to specify the interface, operation and input parameter, the data resource is used for (lines 12 to 15 in Listing 5.8). As many processing logics as required can be attached to one data resource, because the attribute *maxOccurs* is set to "unbounded". A Node Template *DataResourceFile* of type *DataResource* is specified in Listing 5.9. For the property *Location* the placeholder *address* is used, which can be replaced by a relative URI in case of a shipping of the actual data, or any other address if a reference to a remote data location is shipped. In this example the property *ProcLogicID* of the Node Template *DataResourceFile* refers to the processing logic *Calculator*, specifically to the interface *CalculationInterface* and its operation *calculate*, which should process the data. The data serve as the input for the input parameter *input1*. In case the data resource should be processed by more than one processing logic, additional processing logics can be assigned because the *ProcLogic* element can occur multiple times.

**Listing 5.9** Option 2.2: Node Template *DataResourceFile*

```
1 <NodeTemplate id="DataResourceFile" name="Data Resource File" type="DataResource">
2   ...
3   <Properties>
4     <DataResourceProperties>
5       <Location ref="address"/>
6       <ProcLogic>
7         <ProcLogicID>Calculator</ProcLogicID>
8         <Interface>CalculationInterface</Interface>
9         <Operation>calculate</Operation>
10        <InputParameter>input1</InputParameter>
11      </ProcLogic>
12    </DataResourceProperties>
13  </Properties>
14 </NodeTemplate>
```

*Result*

This TOSCA realization option can be used for both data shipping use cases, the shipping of the actual data as well as the shipping of the reference to a remote data location, similar to Option 2.1. Either the property *Location* references a data resource in a subdirectory of the CSAR, which also contains the Service Template or it references a remote location.

*Assumptions and restrictions*

In contrast to Option 2.1, a more detailed addressing of the processing logic used for the data resource is possible. The interface, operation and input parameter of the processing logic is modeled. In this Option, the semantic of multiple links is also not defined.

In order to model the data shipping in this way four assumptions are made:

- The Node Template representing the processing logic refers to a Node Type which uses the Application Interface extension for TOSCA [Zim16]. This is required in order to explicitly address the input parameter.
- The referenced processing logic(s) is/are already deployed in the target runtime environment and the ID(s) of the Node Template(s) representing the processing logic(s) is/are unique in this environment.
- The modeler knows the exact name and the details of the Application Interface of the Node Template representing the processing logic in the target runtime environment.

- There is no need for further information about data format, data semantics, or data transformation operations to enable the mapping between data and processing logic and the accurate processing of the data. Either the TOSCA runtime environment or the indicated processing logic can handle potential adaptations.

*Extensions*

For modeling the Service Template as shown in Fig. 5.8 no extension is required, but for the modeling of the processing logic, which is not part of this model, the Application Interface extension have to be used. To process this model the *Location* has to be interpreted as the data storage location. The defined properties of the Node Template, which indicates the required processing logic have to be mapped to a Node Template with the corresponding ID and interface, operation, and input parameter name. If the data are stored at a remote location, they have to be retrieved at the latest during runtime.

## 5.3 Concept 3: Uniquely Addressable Processing Logic

In contrast to Concept 1 and Concept 2, the main subject of this concept is the modeling of the processing logic.

*Context*

A processing logic which should be used for data processing is provided by the processing logic owner. The required data resources are either not known or just one data resource is available. In the first case, an external mechanism or a person has to map the processing logic with the right data resource. In the second case, the assignment can be implied because only one data resource exists.

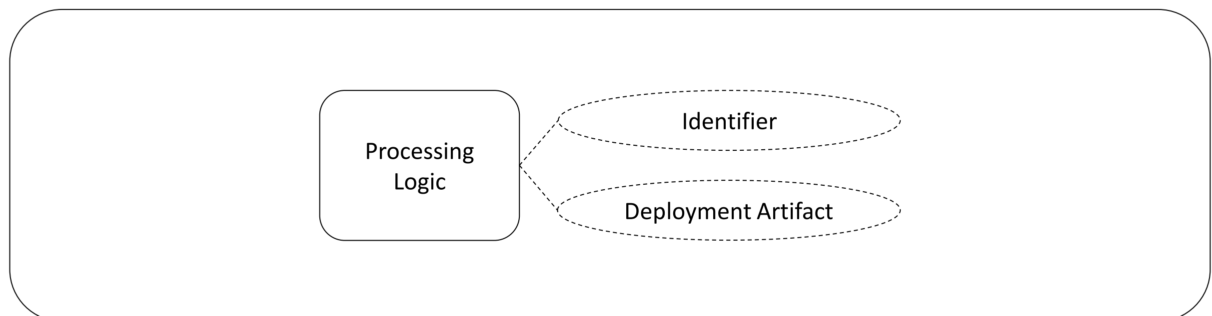



**Figure 5.9:** Concept 3: Uniquely addressable processing logic

*Problem*

How can a processing logic, which is used for data processing, be specified?

*Solution*

The modeling concept is shown in Fig. 5.9. A *Processing Logic* entity is defined, which is uniquely addressable by an *Identifier* and requires a *Deployment Artifact* to install and run the actual processing logic. The deployment artifact can be stored locally or at a remote location. Therefore, this concept is applicable for both function shipping use cases, illustrated in Fig. 4.3: shipping of the actual function as well as shipping of a reference to the remote location. Due to missing modeling elements representing the data, this concept is not suitable for data shipping.

*Implications*

This concept enables the specification of a processing logic with the storage location of the actual deployment artifact. It can not integrate the data, which should be processed by this processing logic. If it is necessary to additionally model the mapping of the data to the respective processing logic, Concept 4 or Concept 5 should be used.

### 5.3.1 Option 3.1: TOSCA Realization with Processing Logic Node Type Definition

In this TOSCA option, illustrated in Fig. 5.10, a Node Type *ProcessingLogic* is defined, which is used to specify a Node Template representing a concrete processing logic. A Deployment Artifact references the actual artifact that contains the processing logic.

*Description*

The parts of the Service Template defining the Node Type *ProcessingLogic* and specifying the Node Template *Calculator* of type *ProcessingLogic* are shown in Listing 5.10 and Listing 5.11 in pseudo-XML. The Node Template contains a Deployment Artifact *MyCalculatorArtifact*, referencing the Artifact Template *CalculatorInstallable*, which is exemplarily of type *JARArtifact*. This Artifact Type as well as the Artifact Template is shown in Listing 5.12. The Artifact Template *CalculatorInstallable* references the location of the actual deployment artifact. The placeholder *anyURI* should underline that the location depends on the function shipping use case this TOSCA realization option is used for. The actual deployment artifact is either contained in the CSAR with the Service Template or available at a remote location. The placeholder is replaced accordingly.

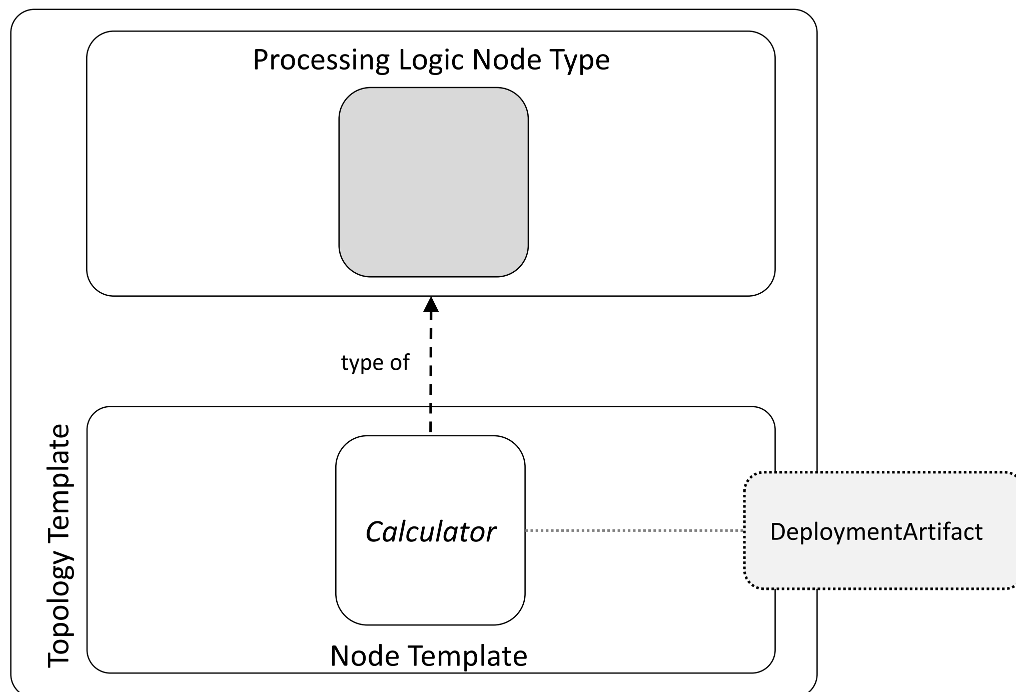



**Figure 5.10:** Option 3.1: TOSCA realization with processing logic Node Type definition

**Listing 5.10** Option 3.1: Node Type definition *ProcessingLogic*

```
1 <NodeType name="ProcessingLogic">
2   ...
3 </NodeType>
```

**Listing 5.11** Option 3.1 and 4.2: Node Template *Calculator*

```
1 <NodeTemplate id="Calculator" type="ProcessingLogic">
2   ...
3   <DeploymentArtifacts>
4     <DeplyomentArtifact name="MyCalculatorArtifact" artifactType="JARArtifact"
          artifactRef="CalculatorInstallable"/>
5   </DeploymentArtifacts>
6 </NodeTemplate>
```

**Listing 5.12** Artifact Type definition *JARArtifact* and Artifact Template *CalculatorInstallable*

```
1 <ArtifactType name="JARArtifact">
2   ...
3 </ArtifactType>
4 ...
5 <ArtifactTemplate id="CalculatorInstallable" type="JARArtifact">
6   ...
7   <ArtifactReferences>
8     <ArtifactReference reference="anyURI"/>
9   </ArtifactReferences>
10 </ArtifactTemplate>
```

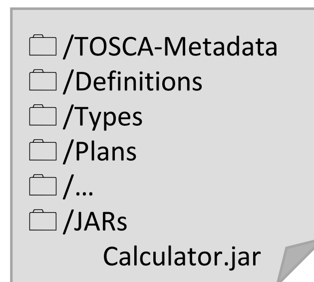



**Figure 5.11:** Exemplary structure of a CSAR containing the processing logic

*Result*

This TOSCA realization can be applied for both function shipping use cases illustrated in Fig. 4.3:

(a) Function packaged in an archive: the actual deployment artifact is contained in the CSAR together with the overall Service Template. The CSAR is transferred to the target runtime environment where the data are already available. Therefore, the Artifact Template references the location in the CSAR, for example "JARs/Calculator.jar" with the corresponding CSAR shown in Fig. 5.11. That way, the actual processing logic is shipped and can be deployed in the target runtime environment for the data processing.

(b) Reference to the remote function location packaged in an archive: the deployment artifact of the processing logic is available at a remote location. This could be for example an FTP-server. The TOSCA Service Template containing a reference to the location is sent within a CSAR to the target runtime environment. For the instantiation the deployment artifact can be retrieved from the remote location.

*Assumptions and restrictions*

This option is only applicable in situations where the data are not considered. The relationship between processing logic and data is not part of the model. If an explicit assignment is required, Concept 4 or Concept 5 should be chosen. In order to model the function shipping in this way three assumptions are made:

- The data which should be processed are already available in the target runtime environment.
- Either an external mechanism or a person can map the processing logic to the data in case several data resources are available, or the allocation is implicitly given because just one data resource is available.
- The runtime environment or the processing logic itself can perform required transformation operations or other adaptations.

*Extensions*

This option can be realized without any extensions of the TOSCA metamodel. In this case the TOSCA language is used as it is initially intended without considering data resources. However, the target environment has to be able to assign the data to the processing logic.

## 5.4 Concept 4: Uniquely Addressable Processing Logic with Assigned Data Identifier

As well as Concept 3, this concept focuses on the processing logic. In addition, the assignment of data resources to the processing logic is modeled.

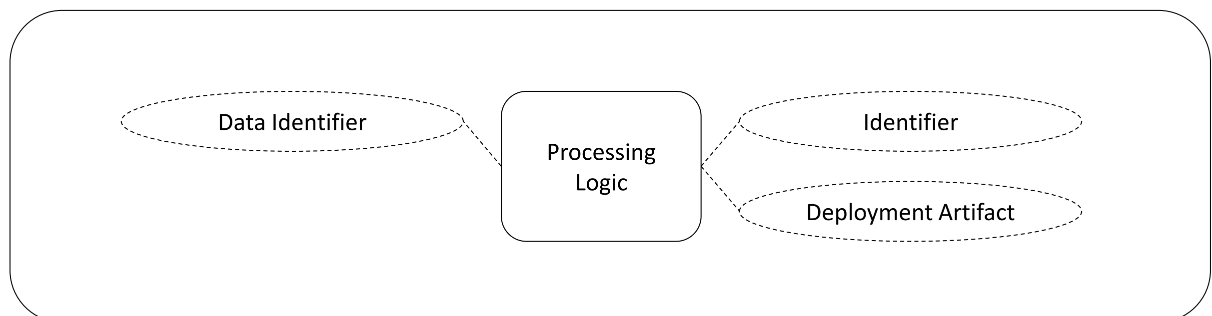



**Figure 5.12:** Concept 4: Uniquely addressable processing logic with assigned data identifier

*Context*

A processing logic owned by a processing logic owner should be used to process a specific data resource. In case that more than one data resource is available and the target runtime environment cannot map the right data resource to the processing logic, the modeler has to specify the assignment. The required data resource is known by the modeler. Concept 3 does not allow to determine the assignment. The mapping between processing logic and data resource is also considered by Concept 2, but it is from the data owner's point of view and concentrates on the modeling of the data resource instead of the processing logic.

*Problem*

How can the modeler specify which data resource should be processed by the processing logic?

*Solution*

Fig. 5.12 shows how the data resource can be attached to the processing logic. The processing logic is represented by the *Processing Logic* entity, which is uniquely addressable by its *Identifier* and requires an *Deployment Artifact* to deploy and run the processing logic. The *Data Identifier* indicates the data resource, which should be processed by this processing logic. The *Data Identifier* refers to the ID of the respective data resource. The actual deployment artifact referenced by the *Deployment Artifact* attribute can be stored locally or at a remote location. Thus, the concept can be used for both function shipping use cases. Since the *Data Identifier* refers to the ID of the data resource and not to the data storage location, this concept is not suitable for data shipping.

*Implications*

This concept enables the mapping of a data resource to a processing logic based on the processing logic owner's perspective. The processing logic is the main subject of this concept and thus it is applicable for function shipping. To realize data shipping one of the other concepts, except Concept 3, can be used.

### 5.4.1 Option 4.1: TOSCA Realization with Data Assignment via Property

Fig. 5.13 depicts a realization option in TOSCA to attach a data identifier to a processing logic. Just like in Option 3.1, a Node Type *ProcessingLogic* is defined and a Deployment Artifact is attached to a Node Template of type *ProcessingLogic*. Additionally, a property *DataResID* for the Node Type is defined to reference the data resource.

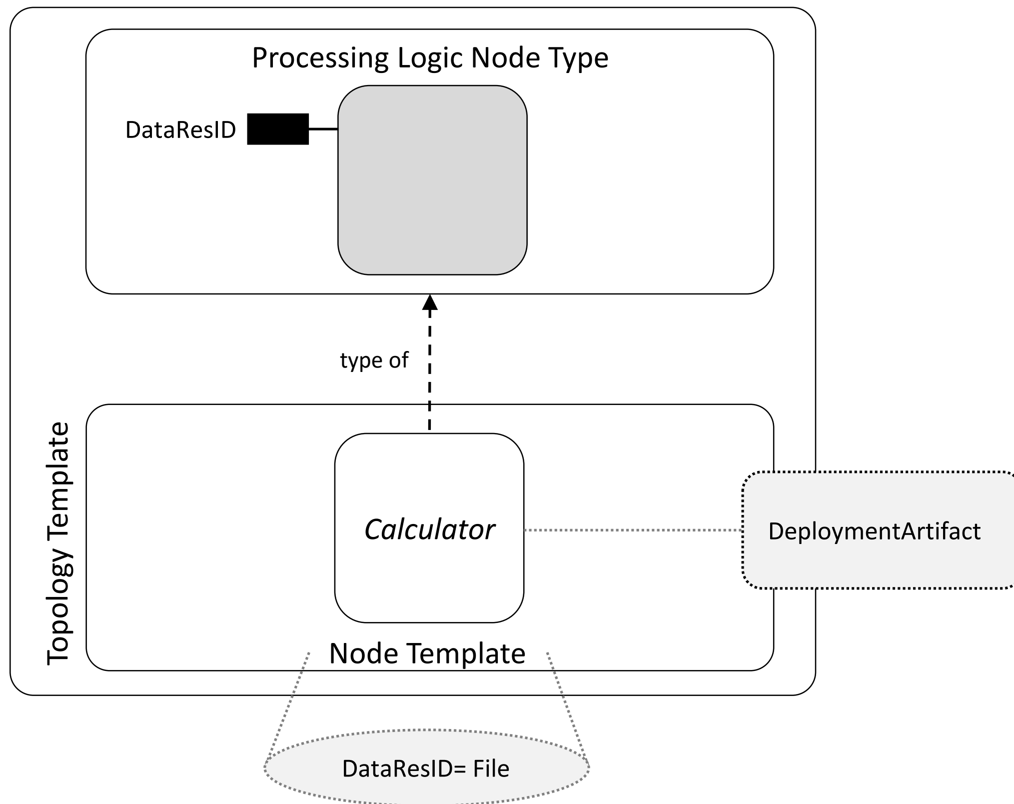



**Figure 5.13:** Option 4.1: TOSCA realization with data assignment via property

**Listing 5.13** Option 4.1: Node Type definition *ProcessingLogic*

```
1 <NodeType name="ProcessingLogic">
2   ...
3   <PropertiesDefinition element="ProcessingLogicProperties"/>
4 </NodeType>
```

*Description*

Listing 5.13 illustrates the Node Type *ProcessingLogic* definition in pseudo-XML. The Node Type definition includes properties defined in Listing 5.14. The property *DataResID* of type "string" should be used to reference the Node Template representing the data resource, which is already available in the target runtime environment. This property can appear multiple times because the attribute *maxOccurs* is set to "unbounded". In this example the property *DataResID* of the Node Template *Calculator*, defined in Listing 5.15, refers to the data resource named *File* (line 5). The attached Deployment Artifact refers to the Artifact Template *CalculatorInstallable* specified in Listing 5.12. The placeholder *anyURI* in line 8 can be replaced by a relative URI in case the actual deployment artifact is contained in the CSAR containing the Service Template, or an absolute URI in case it is available at a remote location.

**Listing 5.14** Option 4.1: Properties definition for Node Type *ProcessingLogic*

```
1 <xs:element name="ProcessingLogicProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="DataResID" type="xs:string" maxOccurs="unbounded"/>
5     </xs:sequence>
6   </xs:complexeType>
7 </xs:element>
```

**Listing 5.15** Option 4.1: Node Template *Calculator*

```
1 <NodeTemplate id="Calculator" name="Calculator" type="ProcessingLogic">
2   ...
3   <Properties>
4     <ProcessingLogicProperties>
5       <DataResID>File</DataResID>
6     </ProcessingLogicProperties>
7   </Properties>
8   <DeploymentArtifacts>
9     <DeplyomentArtifact name="MyCalculatorArtifact" artifactType="JARArtifact"
          artifactRef="CalculatorInstallable"/>
10  </DeploymentArtifacts>
11 </NodeTemplate>
```

*Result*

This TOSCA realization can be applied for both function shipping use cases in the same way as Option 3.1:

(a) Function packaged in an archive: the actual deployment artifact is contained in a subdirectory of the CSAR containing the Service Template and the Artifact Template that references the path in the CSAR. An exemplary structure of a respective CSAR is shown in Fig. 5.11. In this example the artifact is referenced by "JARs/Calculator.jar".

(b) Reference to the remote function location packaged in an archive: the deployment artifact of the processing logic is available at a remote location and can be retrieved when needed. The Service Template contains a reference to the remote location and is shipped within the CSAR. At the latest during provisioning time the actual artifact is retrieved.

*Assumptions and restrictions*

This option enables to link a data resource to a processing logic, but a specific input parameter for which the data should be used cannot be specified. If a more accurate allocation is required, one of the other two TOSCA realization options of this concept,

Option 4.2 or Option 4.3, can be used. In the example shown in Fig. 5.13 just one data resource is associated with the processing logic. According to the property definition as seen in Listing 5.14, several data resources can be linked to one processing logic. However, the semantics of multiple links is not defined. Either the assigned data resources should be joined and serve as a single input or each data resource is used for different invocations of the processing logic.

In order to model the function shipping in this way three assumptions are made:

- The referenced data resource(s) is/are already available in the target runtime environment and the ID(s) of the Node Template(s) representing the data resource(s) is/are unique in this environment.
- The modeler knows the exact name of the Node Template representing the data resource(s) in the target runtime environment.
- The runtime environment or processing logic, respectively, assigns the data to the concrete input parameter of the processing logic and can apply required transformation operations and further adaptations.

*Extensions*

A TOSCA metamodel extension is not required for this option. However, the value of the property *DataResID* has to be mapped to the ID of a Node Template representing a data resource already available in the target runtime environment. This data have to be linked to the processing logic.

### 5.4.2 Option 4.2: TOSCA Realization with Data Assignment via Input Parameter

Similar to Option 4.1, a Node Type *ProcessingLogic* is defined and a Node Template of this type is specified. However, this option differs from Option 4.1 in terms of the assignment of data resources to the processing logic. The data resources are directly assigned to the input parameter, as seen in Fig. 5.14.

*Description*

In this option the data resources used for this processing logic are directly associated with the respective input parameter. Listing 5.16 illustrates the Node Type *ProcessingLogic*. Lines 4 to 12 define an Application Interface *CalculationInterface* with an operation *calculate*. This operation requires two input parameters *input1* and *input2*, both of type "string". The attribute *value* (lines 8 and 9) indicates that a data resource with the ID *File1* is linked to the input parameter *input1* and the data resource *File2* to

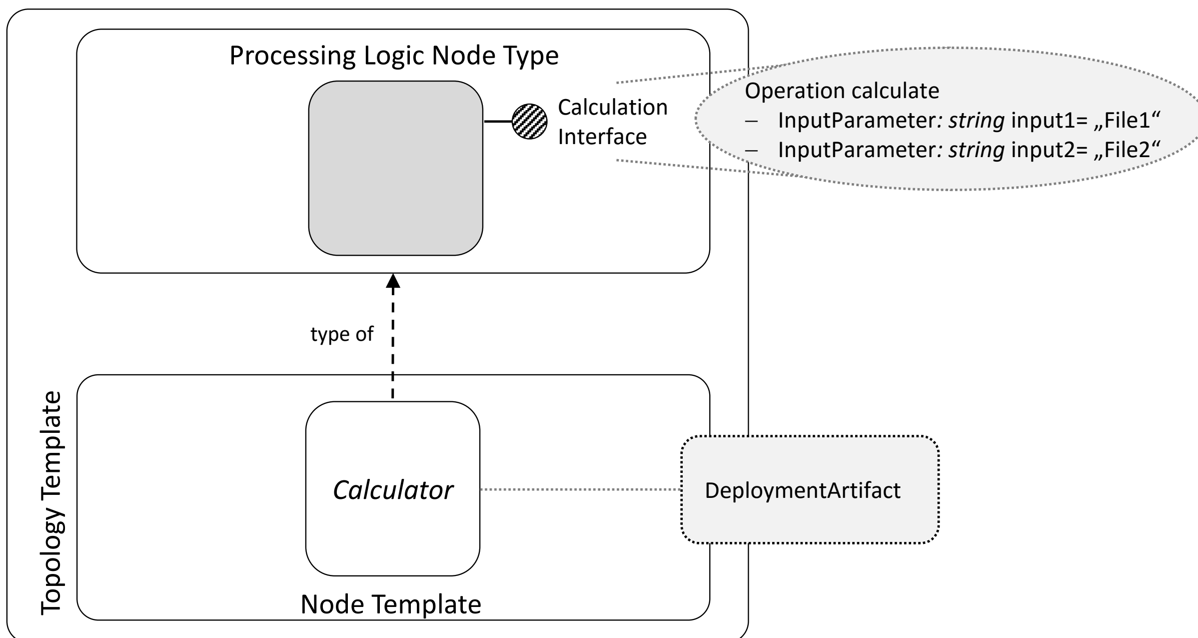



**Figure 5.14:** Option 4.2: TOSCA realization with data assignment via input parameter

**Listing 5.16** Option 4.2: Node Type definition *ProcessingLogic*

```
1 <NodeType name="ProcessingLogic">
2   ...
3   <opentosca:ApplicationInterfaces
        xmlns:opentosca="http//www.uni-stuttgart.de/opentosca">
4     <Interface name="CalculationInterface">
5       <Operation name="calculate">
6         <documentation>calculates a result from two input values</documentation>
7         <InputParameters>
8           <InputParameter name="input1" type="xs:string" value="File1"/>
9           <InputParameter name="input2" type="xs:string" value="File2"/>
10        </InputParameters>
11      </Operation>
12    </Interface>
13  </opentosca:ApplicationInterfaces>
14 </NodeType>
```

input parameter *input2*. The attribute *value* is a new extension and is explained later. The specification of Node Template *Calculator* corresponds to the specification seen in Listing 5.11 with the associated Deployment Artifact referencing the actual deployment artifact of the processing logic, illustrated in Listing 5.12.

*Result*

This TOSCA realization option can be used for both function shipping use cases: the actual processing logic is shipped with the CSAR or is available at a remote location. Both use cases can be realized in the same way as described for Option 4.1. Either the Deployment Artifact references the artifact in a subdirectory of the CSAR or at a remote location.

*Assumptions and restrictions*

In contrast to Option 4.1, the data resource can be assigned to a particular input parameter. However, the number of data resources assigned to an input parameter is limited to one. The assignment of Node Template IDs representing the data to an *Application Interface* of a Node Type restricts the reusability of the Node Type. The assigned Node Template IDs to the attribute *value* are the same for each Node Template of this Node Type.

In order to model the function shipping in this way three assumptions are made:

- The referenced data resources are already available in the target runtime environment and the IDs of the Node Templates representing the data resources are unique in this environment.
- The modeler knows the exact name of the Node Template representing the data resources in the target runtime environment.
- The runtime environment or processing logic can perform required transformation operations.

*Extensions*

In this case the TOSCA Application Interface extension introduced by Zimmermann [Zim16] is required. This extension enables the definition of the application interface, operations, and the input parameters in TOSCA. It does not provide the possibility to assign a fixed value to the parameter. Therefore, an additional extension is required to enable the assignment of a fixed value to an input parameter. The attribute *value* defined for the input parameter extends the Application Interface definition. The ID given by the attribute *value* has to be mapped to a Node Template with the same ID representing a data resource. This specific data resource has to be linked to the input parameter.

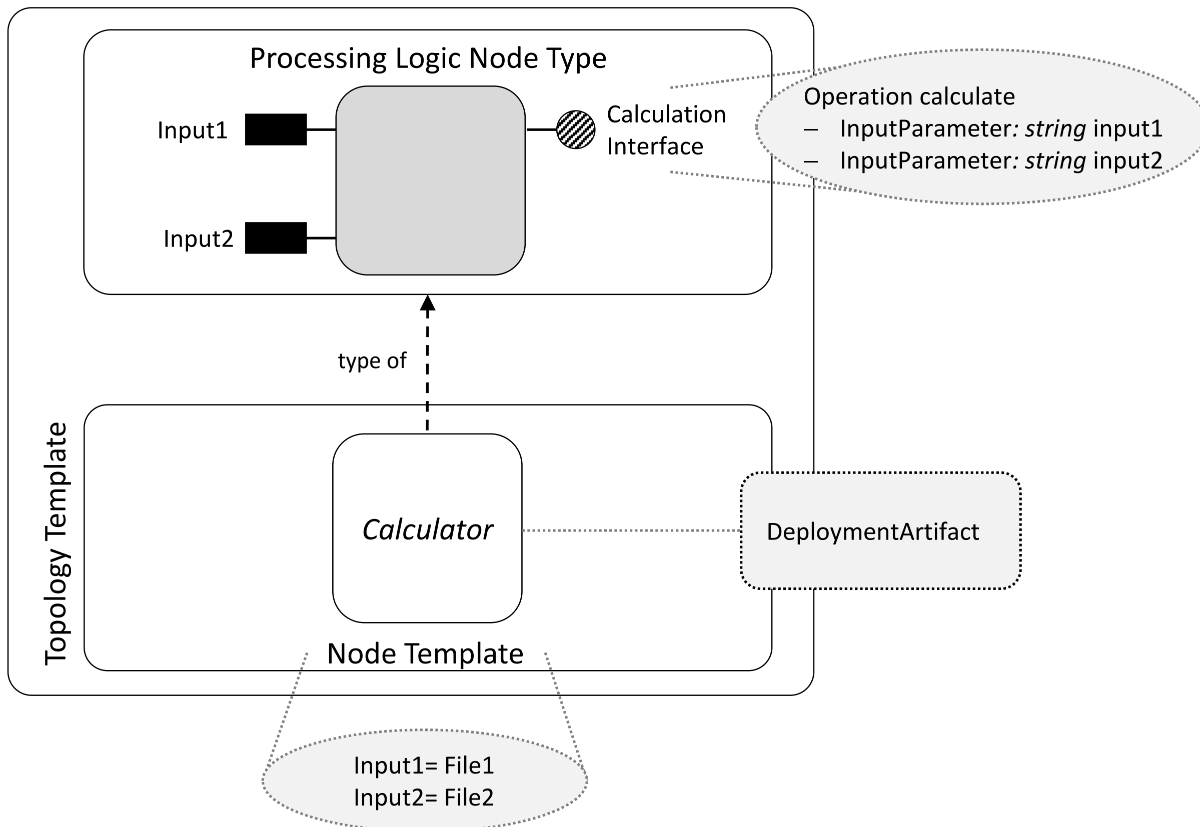



**Figure 5.15:** Option 4.3: TOSCA realization with data assignment via property and input parameter matching

### 5.4.3 Option 4.3: TOSCA Realization with Data Assignment via Property and Input Parameter matching

This option combines elements from Option 4.1 and Option 4.2. As depicted in Fig. 5.15, an Application Interface with an operation and input parameters is modeled. Contrary to Option 4.2, the data resources are not assigned directly to the parameter, but attached to the Node Template as properties. Therefore this option is more flexible in terms of the allocation of data resources to the input parameters. A data resource can be assigned to a Node Template representing a processing logic and not to the Node Type like it is shown in Option 4.2.

*Description*

In this option, a Node Type *ProcessingLogic* with an Application Interface and properties is defined. Listing 5.17 illustrates the definition of Node Type *ProcessingLogic*. Line 3 refers to the properties, which are declared in Listing 5.18. Lines 5 to 13 define an Application Interface *CalculationInterface* with the operation *calculate* and the two input

**Listing 5.17** Node Type definition *ProcessingLogic* with application interface

```
1 <NodeType name="ProcessingLogic">
2   ...
3   <PropertiesDefinition element="ProcessingLogicProperties"/>
4   <opentosca:ApplicationInterfaces
        xmlns:opentosca="http//www.uni-stuttgart.de/opentosca">
5     <Interface name="CalculationInterface">
6       <Operation name="calculate">
7         <documentation>calculates a result from two input values</documentation>
8         <InputParameters>
9           <InputParameter name="input1" type="xs:string"/>
10          <InputParameter name="input2" type="xs:string"/>
11        </InputParameters>
12      </Operation>
13    </Interface>
14  </opentosca:ApplicationInterfaces>
15 </NodeType>
```

**Listing 5.18** Option 4.3: Properties definition for Node Type *ProcessingLogic*

```
1 <xs:element name="ProcessingLogicProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="Input1" type="xs:string" maxOccurs="unbounded"/>
5       <xs:element name="Input2" type="xs:string" maxOccurs="unbounded"/>
6     </xs:sequence>
7   </xs:complexeType>
8 </xs:element>
```

parameters *input1* and *input2*. Listing 5.18 defines the two properties *Input1* and *Input2* of type "string" for the Node Type *ProcessingLogic*. Both elements can occur multiple times. The semantics of multiple properties with the same name is explained later. A Node Template *Calculator* is specified in Listing 5.19. The properties get the values *File1* and *File2*, which reference the data resource IDs. These IDs relate to Node Templates representing the data resources. Furthermore, a Deployment Artifact referencing the actual deployment artifact is specified. The refered Artifact Type and Artifact Template is already seen in Listing 5.12. To achieve the mapping between input parameter and data resource, a naming convention is introduced: the name of the input parameter and the name of the property referencing the required data resource are equal. In this example, the data resource referenced by the the property *Input1* serves as input for the input parameter with the name *input1*.

**Listing 5.19** Option 4.3: Node Template *Calculator*

```
1 <NodeTemplate id="Calculator" type="ProcessingLogic">
2  ...
3   <Properties>
4    <ProcessingLogicProperties>
5      <Input1>File1</Input1>
6      <Input2>File2</Input2>
7    </ProcessingLogicProperties>
8   </Properties>
9   <DeploymentArtifacts>
10    <DeplyomentArtifact name="MyCalculatorArtifact" artifactType="JARArtifact"
        artifactRef="CalculatorInstallable"/>
11  </DeploymentArtifacts>
12 </NodeTemplate>
```

*Result*

In the same way as Option 4.1, both function shipping use cases can be realized by this option. Either the actual deployment artifact is shipped with the CSAR containing the Service Template or it remains at a remote location.

*Assumptions and restrictions*

This option combines the benefits of Option 4.1 and Option 4.2. The data resource can be assigned to a specific input parameter and as much data resources as required can be associated with one parameter. The fact remains, that the semantics of multiple data resources assigned to one input parameter is not defined. It could imply either a join of the associated data resources or separate invocations, each with one associated data resource.

In order to model the function shipping in this way three assumptions are made:

- The referenced data resources are already available in the target runtime environment and are known by the modeler.
- The modeler knows the exact name of the Node Template representing the data resources in the target runtime environment and the IDs of these Node Templates are unique.
- The runtime environment or processing logic can perform required transformation operations.

*Extensions*

For modeling the application interface the extension introduced by Zimmermann [Zim16] is required, but no further metamodel extensions are required. However, the domain specific elements have to be processed by the runtime environment. The

defined properties have to be mapped to the input parameter with the same name, by convention. The data resource IDs declared for the properties have to be aligned with the existing data resources in the runtime environment. The corresponding data have to be linked to the input parameter. As depicted in Fig. 5.15, the property *Input1* are assigned to the data resource ID *File1*. Therefore, this data resource is used as input for the input parameter *input1*.

## 5.5 Concept 5: Data Connector between Processing Logic and Data Resource

This concept pursues another approach than the first four concepts. Instead of considering the scenario either from the data owner's or the processing logic owner's perspective, it focuses on the connection between processing logic and data resource.

*Context*

Either the data owner or the processing logic owner wants to ship the data to the processing logic, or the processing logic to the data, respectively. The relationship between them is important. All concepts explained before are applicable only for data shipping or function shipping. Therefore, a flexible model is required, which does not differ whether it is used for data shipping or function shipping.

*Problem*

How can the modeler specify the relationship between the processing logic and the data resource?

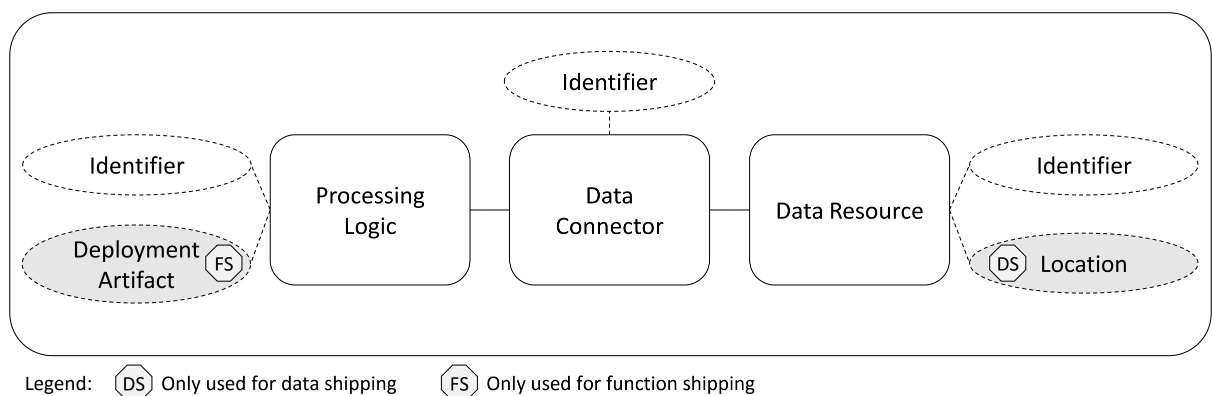



**Figure 5.16:** Concept 5: Data connector between processing logic and data resource

*Solution*

Fig. 5.16 illustrates the abstract modeling concept. The *Processing Logic* entity on the left side is uniquely addressable by an *Identifier*. The same applies to the *Data Resource* entity on the right side. The *Data Connector* represents an 1:1 relationship between a *Processing Logic* and a *Data Resource*. It can also be identified by an unique *Identifier*. The gray-shaded elements with the icon mark the attributes, which usage depends on the shipping use case the concept is used for. Two cases can be distinguished: Either the *Deployment Artifact* or the *Location* element is part of the model. The first case occurs if the model is applied for function shipping, the second case if data shipping is realized. Therefore, this modeling concept is applicable for all four shipping use cases, both data shipping as well as both function shipping use cases, presented in Fig. 4.2 and Fig. 4.3.

*Implications*

This concept enables the modeler to specify the relationship between a processing logic and a data resource. The location of the deployment artifact, in case of function shipping, or the actual data location, in case of data shipping, can be referenced. Thus, one single modeling concept covers all use cases and can be adapted flexibly according to the underlying situation. Contrary to the Concept 1 to Concept 4, this concept promotes a common understanding of the scenario, which should be modeled, independent of the use case. It can be used either for the data shipping or the function shipping use cases. The subsequent concepts Concept 6 and Concept 8 demonstrate how additional details such as transformation operations can be added to the *Data Connector*.

### 5.5.1 Option 5.1: TOSCA Realization with an Data Connector with Input Parameter Matching

Fig. 5.17 illustrates how the *Data Connector* can be realized with TOSCA. A Node Type *ProcessingLogic*, representing the processing logic, as well as a Node Type *DataResource*, representing the data, are defined. In addition a Relationship Type *DataConnector* is defined, which can be used to specify Relationship Templates connecting Node Templates of type *ProcessingLogic* and *DataResource*.

*Description*

Both Node Type definitions are presented above. Listing 5.17 shows the definition of the Node Type *ProcessingLogic* with the Application Interface *CalculationInterface* (the properties are not used in this case). Listing 5.3 shows the definition of the Node Type *DataResource* with the associated properties definition in Listing 5.4. The property *Location* is utilized to specify the actual storage location. The Relationship Type *DataConnector* is defined in Listing 5.20 with the respective properties shown in

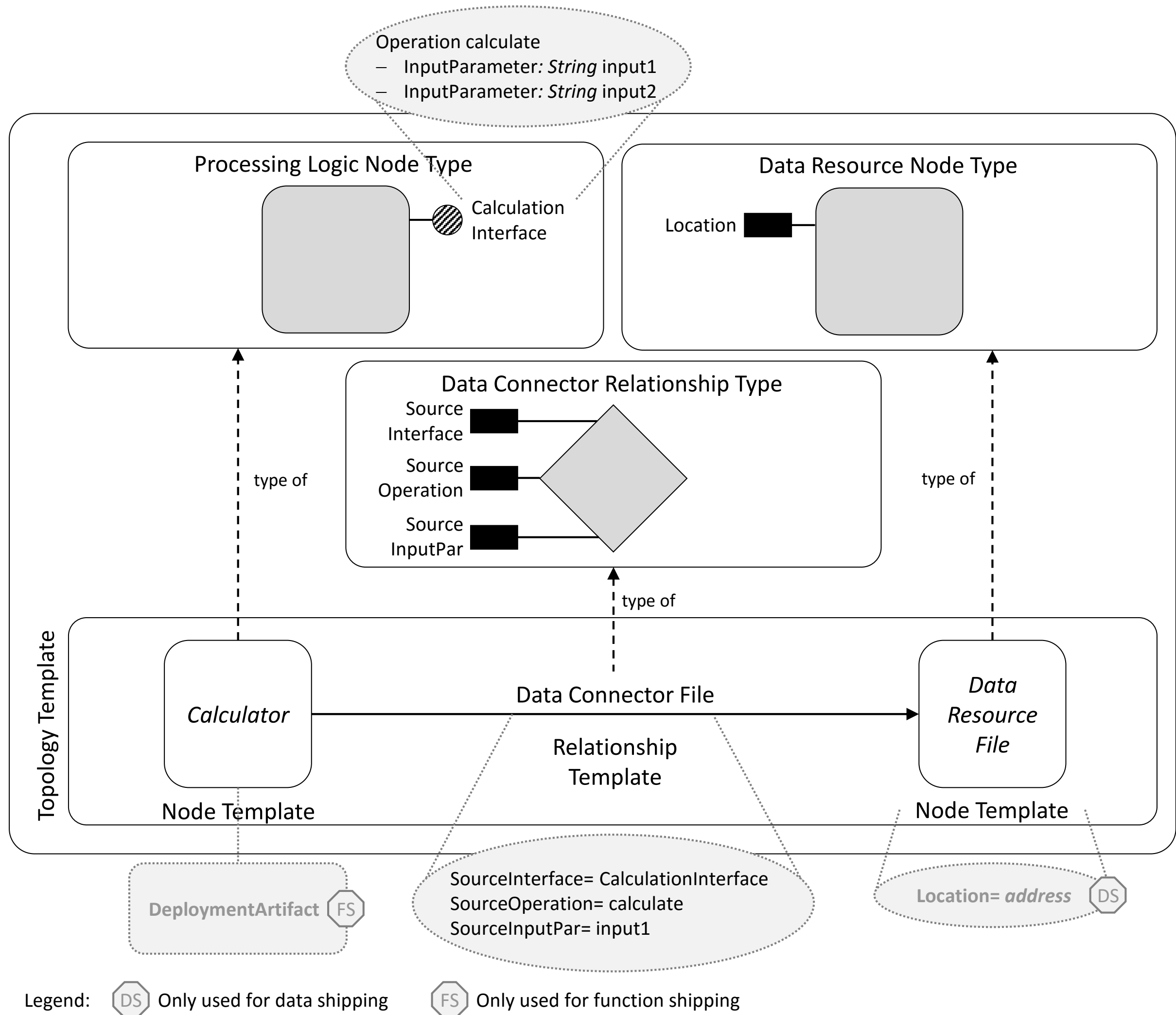



**Figure 5.17:** Option 5.1: TOSCA realization with data connector with input parameter matching

Listing 5.21. Since the properties *SourceElement* and *TargetElement* of a Relationship Template can only reference Node Templates, further properties to specify the exact interface, operation, and input parameter the data relates to are required. Due to the fact that the indicator of an input parameter is only unique within an operation element and an operation is only unique within the interface element, the interface name as well as the operation name have to be declared. Listing 5.22 shows the Relationship Template *DataConnectorFile* of type *DataConnector*. The Node Template with the ID *Calculator* is defined as the *SourceElement* (line 10) and the Node Template with the ID *DataResourceFile* as *TargetElement* (line 11). The properties are used to specify the exact input parameter the data are used for. In this case the data represented by the Node Template *DataResourceFile* serve as input for the input parameter *input1* required by the

**Listing 5.20** Relationship Type definition *DataConnector* with properties

```
1 <RelationshipType name="DataConnector">
2   ...
3   <RelationshipTypeProperties element="DataConnectorProperties"/>
4 </RelationshipType>
```

**Listing 5.21** Properties definition for Relationship Type *DataConnector*

```
1 <xs:element name="DataConnectorProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="SourceInterface" type="xs:string"/>
5       <xs:element name="SourceOperation" type="xs:string"/>
6       <xs:element name="SourceInputPar" type="xs:string"/>
7     </xs:sequence>
8   </xs:complexType>
9 </xs:element>
```

operation *calculate*. Furthermore, two variations of this option can be distinguished: either a Deployment Artifact is assigned to the Node Template *Calculator* or the property *Location* is specified for the Node Template *DataResourceFile*. The first variation is used in case the TOSCA realization option is used for function shipping, the second one in case of data shipping.

*Result*

Data shipping and function shipping can be realized by this option, in each case the shipping of the actual data or function as well as the shipping of a reference to a remote location. The following differences for the realization have to be considered:

**Listing 5.22** Option 5.1: Relationship Template *DataConnectorFile*

```
1 <RelationshipTemplate id="DataConnctorFile" name="Data Connector File"
      type="DataConncetor">
2   ...
3   <Properties>
4     <DataConncectorProperties>
5       <SourceInterface>CalculationInterface</SourceInterface>
6       <SourceOperation>calculate</SourceOperation>
7       <SourceInputPar>input1</SourceInputPar>
8     </DataConncetorProperties>
9   </Properties>
10  <SourceElement ref="Calculator"/>
11  <TargetElement ref="DataResourceFile"/>
12 </RelationshipTemplate>
```

1. Data shipping: in this case the Node Template *Calculator* is abstract, i.e., neither implementation artifacts nor deployment artifacts are assigned and no instances of this Node Template can be created. The Node Template has to be substituted with another one having a specialized, derived Node Type. The one substituting the Node Template *Calculator* is already available in the target runtime environment. The property *Location* of the Node Template *DataResourceFile* references the location of the data. In case the data are contained in the CSAR, which contains the whole Service Template the actual data are shipped in the archive. Fig. 5.4 shows an example of a CSAR. In this case the placeholder *address* of the property *Location* is replaced by "Data/File.csv". For the second data shipping use case, at which the data remain at a remote location, the address identifies a file for example on an FTP-server.

2. Function shipping: the processing logic instead of the data resource is shipped. In this case the Deployment Artifact of the processing logic is assigned to the Node Template *Calculator*. The actual deployment artifact is either contained in the CSAR or available at a remote location. The Node Template *DataResourceFile* is abstract and the property *Location* is not defined. The Node Template *DataResourceFile* is substituted by a Node Template representing data already available in the target runtime environement.

*Assumptions and restrictions*

The Relationship Template *DataConnectorFile* can only connect two Node Templates. How multiple data resources can be connected to one processing logic is shown in Concept 7 and Concept 8. The following assumptions are made to realize data shipping or function shipping:

- The data and the processing logic have to be known by the modeler.
- The data, which should be processed, or the processing logic, which should process the data, are already available in the target runtime environment.
- A mechanism for the substitution either of the abstract Node Template *Calculator* or the Node Template *DataResourceFile* with the appropriate Node Template available in the target runtime environment exists.
- The runtime environment can perform required transformation operations for example data conversion, or data selection operations.

*Extensions*

An additional TOSCA metamodel extension beside the TOSCA Application Interface extension is not required for this option. However, the domain specific defined Node Types has to be interpreted correctly. The Relationship Type *DataConnector* maps the

processing logic with the data. The data serve as input for a specific input parameter of the processing logic. Accordingly, the Node Type *ProcessingLogic* represents the processing logic and the Node Type *DataResource* the data. Furthermore, the substitution of the Node Template *Calculator*, or Node Template *DataResourceFile* respectively, has to be executed.

## 5.6 Concept 6: Data Connector with Transformation Capability

Similarly to Concept 5, this concept focuses on the connection between the processing logic and the data resource. Additionally, transformation capabilities are added to the connection and can be modeled explicitly.

*Context*

The data owner or the processing logic owner possesses a data resource or a processing logic, respectively, which should be shipped. The processing logic cannot process the original data resource, i.e., either the format of the data cannot be processed or just a subset of the data contained in the data resource should be processed. Required transformation operations like format conversion, data conversion, or data selection are not provided by the target runtime environment or the processing logic. Format conversion means that the data format, for example XML or CSV, has to be changed. In case of data conversion the measuring unit or representation of the data has to be converted, for example from Celsius to Fahrenheit or from a "float" to an "integer" data type. The modeler has to specify the required transformation operations, which cannot be realized by Concept 5.

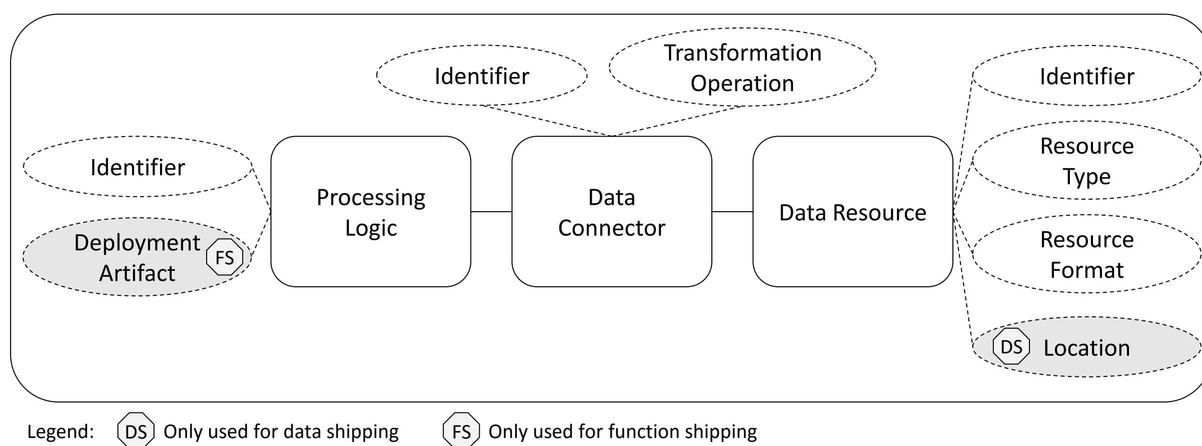



**Figure 5.18:** Concept 6: Data connector with transformation capability

*Problem*

How can the modeler specify the relationship between the processing logic and the data resource, and how can transformation operations be added to the model?

*Solution*

The modeling concept is shown in Fig. 5.18. The *Processing Logic* entity on the left side is identified by an unique *Identifier*. The *Data Resource* is also addressed by an *Identifier*. Additionally, further characteristics of the data are attached to the *Data Resource*: *Resource Type* and *Resource Format*. The first attribute identifies what kind of data resource it is, for example a flat file or a table. The second one determines the storage format of the data resource, e.g, CSV or JSON. The relationship between the processing logic and the data resource is represented by the *Data Connector*, which has a *Transformation Operation* attached in addition to the *Identifier*. The *Transformation Operation* references programs, which can perform the required transformation. Both use cases, data shipping and function shipping, are covered by this modeling concept. The grey-shaded elements *Deployment Artifact* and *Location* with the icon *FS* or *DS* are only required either in case of function shipping or data shipping. In case of function shipping solely the *Deployment Artifact* is needed, in case of data shipping the *Location* attribute is needed.

*Implications*

In addition to Concept 5, transformation operations can be added to the relationship between the processing logic and the data resource. Instead of external mechanisms, which automatically perform required transformations, the modeler explicitly specifies the transformation operations. For any other concept described before, these transformations are not explicitly modeled. The transformation operations can also be combined with data collections, introduced in Concept 7. For instance, the operation can be applied not only to single files or tables but also to multiple tables of one database. In Concept 8 operations in terms of joining data from different data resources are described in detail.

### 5.6.1 Option 6.1: TOSCA Realization with Transformation Assignment via Property

As shown in Fig. 5.19, two Node Types are defined: *ProcessingLogic*, representing the processing logic, and *DataResource*, representing the data. The Relationship Type *DataConnector* represents the connection between the two Node Types. Compared to Option 5.1, a *TransformOp* property is added to the Relationship Type.

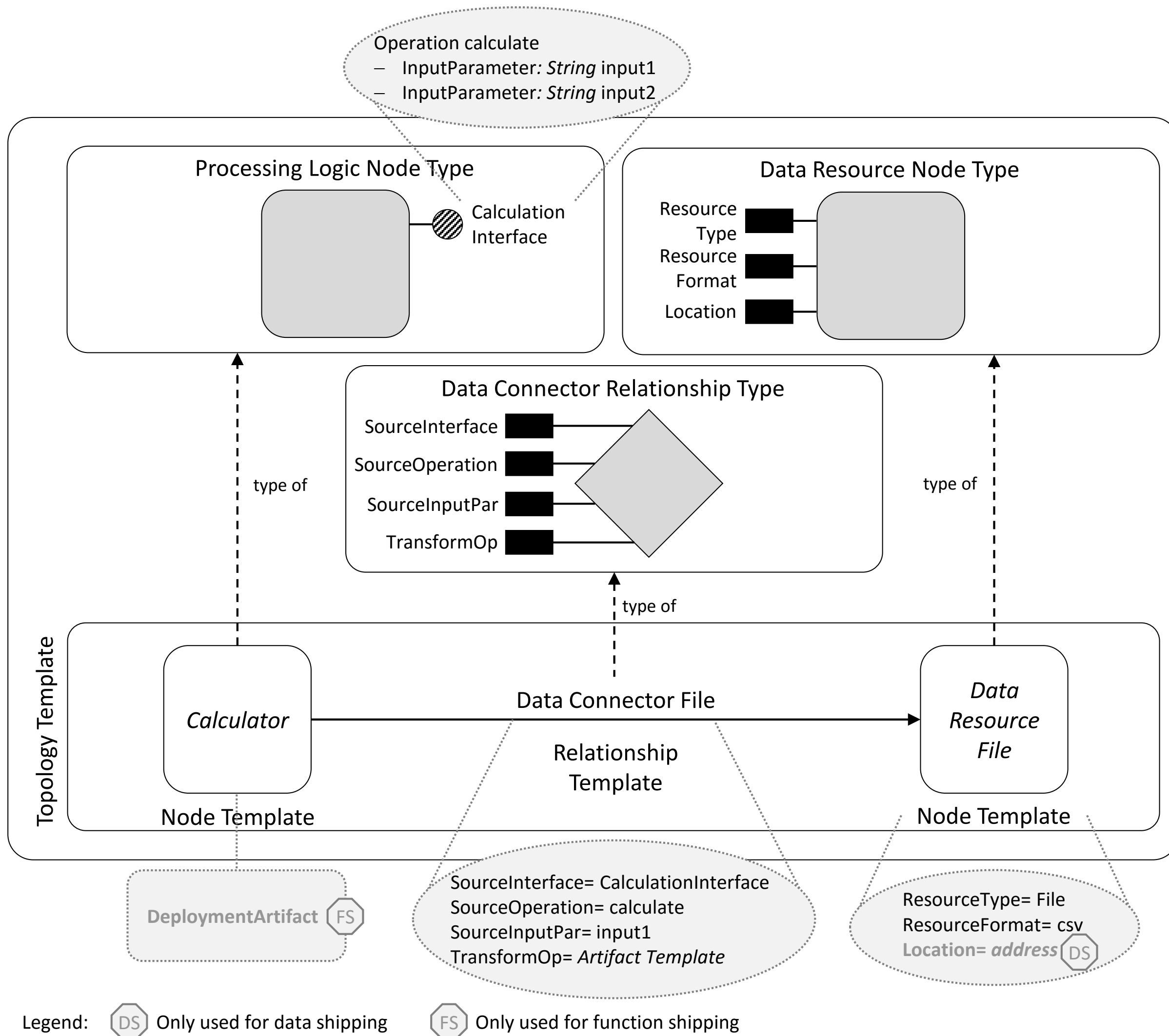



**Figure 5.19:** Option 6.1: TOSCA realization with transformation assignment via property

*Description*

The Node Type *ProcessingLogic* is defined as in Option 5.1, illustrated in Listing 5.17 without the properties in line 4. The processing logic provides an Application Interface *CalculationInterface* with the operation *calculate* that exposes two input parameters. The Node Type *DataResource* is defined in Listing 5.3 with the properties defined in Listing 5.23. Three properties are defined: *ResourceType*, *ResourceFormat*, and *Location*. The first two properties characterize the data. *ResourceType* indicates what kind of resource it is, e.g., a file or a database table. *ResourceFormat* specifies the storage format. Both properties could be used to enable an automated identification of a required format transformation operation by a modeling tool which proposes transformation

**Listing 5.23** Extended properties definition for Node Type *DataResource*

```
1 <xs:element name="DataResourceProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="ResourceType" type="string"/>
5       <xs:element name="ResourceFormat" type="string"/>
6       <xs:element name="Location">
7         <xs:complexType>
8           <xs:attribute name="ref" type="xs:string"/>
9         </xs:complexType>
10      </xs:element>
11    </xs:sequence>
12  </xs:complexeType>
13 </xs:element>
```

operations to the the modeler or by the runtime environment. The *Location* references the actual data, as seen in almost all other options. The Relationship Type *DataConnector* (Listing 5.20) has four properties assigned. The properties definition is shown in Listing 5.24. The three properties *SourceInterface*, *SourceOperation*, and *SourceInputPar* specify the concrete input parameter the data should be used for. All three properties are required to uniquely address a specific input parameter. The property *TransformOp* is used to refer to an Artifact Template, representing the actual transformation operation. The actual transformation operation is assigned to a specific Relationship Template. Only the property definition is done on the level of types. In this example the Node Template *Calculator* of type *ProcessingLogic* and the Node Template *DataResourceFile* of type *DataResource* are connected by the Relationship Template *DataConnectorFile* defined by the Relationship Type *DataConnector*. In case the model is used for function shipping the Deployment Artifact is assigned to the Node Template *Calculator* and the property *Location* is not used. If it is used for data shipping, the property *Location* to address the data is used and the Deployment Artifact is not assigned to the Node Template *Calculator*.

*Result*

This TOSCA realization option can be applied for data shipping as well as function shipping. How the data shipping and function shipping use cases can be realized is shown below:

1. Data shipping: the property *Location* defined for the Node Type *DataResource* is used to reference the actual data. The data can either be contained in the CSAR itself or are available at a remote location, for example an FTP-server. The processing logic is already available in the target runtime environment. Therefore, the Node Template *Calculator* in the Topology Template contained in the CSAR is

**Listing 5.24** Option 6.1: Properties definition for Relationship Type *DataConnector*

```
1 <xs:element name="DataConnectorProperties">
2  <xs:complexType>
3   <xs:sequence>
4    <xs:element name="SourceInterface" type="xs:string"/>
5    <xs:element name="SourceOperation" type="xs:string"/>
6    <xs:element name="SourceInputPar" type="xs:string"/>
7    <xs:element name="TransformOp">
8     <xs:complexType>
9      <xs:attribute name="artifactType" type="xs:QName"/>
10     <xs:attribute name="artifactRef" type="xs:QName"/>
11    </xs:complexType>
12   </xs:element>
13  </xs:sequence>
14 </xs:complexType>
15 </xs:element>
```

declared as abstract. It is substituted by the Node Template in the target runtime environement representing the concrete processing logic, which should process the data.

2. Function shipping: the actual processing logic instead of the data is shipped, either contained in the CSAR or available at a remote location. The Deployment Artifact references the actual deployment artifact. The Node Template of type *DataResource* represents the data, which are already available in the target runtime environment, abstractly. The Node Template is substituted in the target runtime environment by a concrete Node Template at the latest during provisioning time.

*Assumptions and restrictions*

The Relationship Template can only connect two Node Templates, i.e., only one data resource can be assigned with one Relationship Template to the processing logic. How multiple data resources can be aggregated to a data collection can be seen in Concept 7 and Concept 8. The following assumptions are made to realize data shipping or function shipping:

- The modeler requires additional knowledge about the data, in case of function shipping, and about the processing logic, in case of data shipping. Furthermore, the appropriate transformation operation has to be assigned to the *DataConnector*.
- The assigned Artifact Template representing the transformation operation provides only one operation. For each operation a separate Artifact Template is required.
- The data, which should be processed, or the processing logic, which should process the data, are already available in the target runtime environment.

- A mechanism for the substitution either of the Node Template representing the processing logic or the Node Template representing the data resource with the appropriate Node Template available in the target runtime environment is required.

*Extensions*

The above-mentioned option can be realized with the TOSCA Application Interface extension. Furthermore, the domain specific elements have to be correctly processed. The *DataConnector* connects a specific input parameter of the processing logic to the data and for each processing logic invocation the transformation operation has to be executed. Furthermore, the substitution of either the Node Template representing the processing logic, or the Node Template representing the data, is required and executed by the target runtime environment.

Further properties assigned to the Node Type *DataResource* enable to provide more details about the represented data, so the properties shown in Fig. 5.19 are just examples.

### 5.6.2 Option 6.2: TOSCA Realization with Transformation Assignment via Transformation Interface

This TOSCA realization option is similar to Option 6.1. Fig. 5.20 illustrates the Service Template. Two Node Types are defined representing the processing logic and the data. The Relationship Type *DataConnector* defines the connection between them. In contrast to Option 5.1, an Application Interface instead of a property represents the transformation operation. Thus, the actual transformation operation is assigned to the Relationship Type and effects all Relationship Templates of this type.

*Description*

The Node Type *ProcessingLogic* shown in Listing 5.17, provides an Application Interface *CalculationInterface* with one operation *calculate* and two input parameters. Three properties are attached to the other Node Type *DataResource* as can be seen in Listing 5.23. The properties *ResourceType* and *ResourceFormat* indicate what kind of data resource is represented by this Node Type and in which format the data are stored. The last property *Location* references the actual data. The relationship between the two Node Types is defined by the Relationship Type *DataConnector*, depicted in Listing 5.25 with the attached properties definition in Listing 5.21. Three properties are added to the Relationship Type to identify the precise input parameter the data should be used for. Instead of a fourth parameter that references the transformation operation, an Application Interface *TransformationInterface*, which exposes the operation *transform* is attached to the Relationship Type. The ID of the Node Template representing the data serve as input

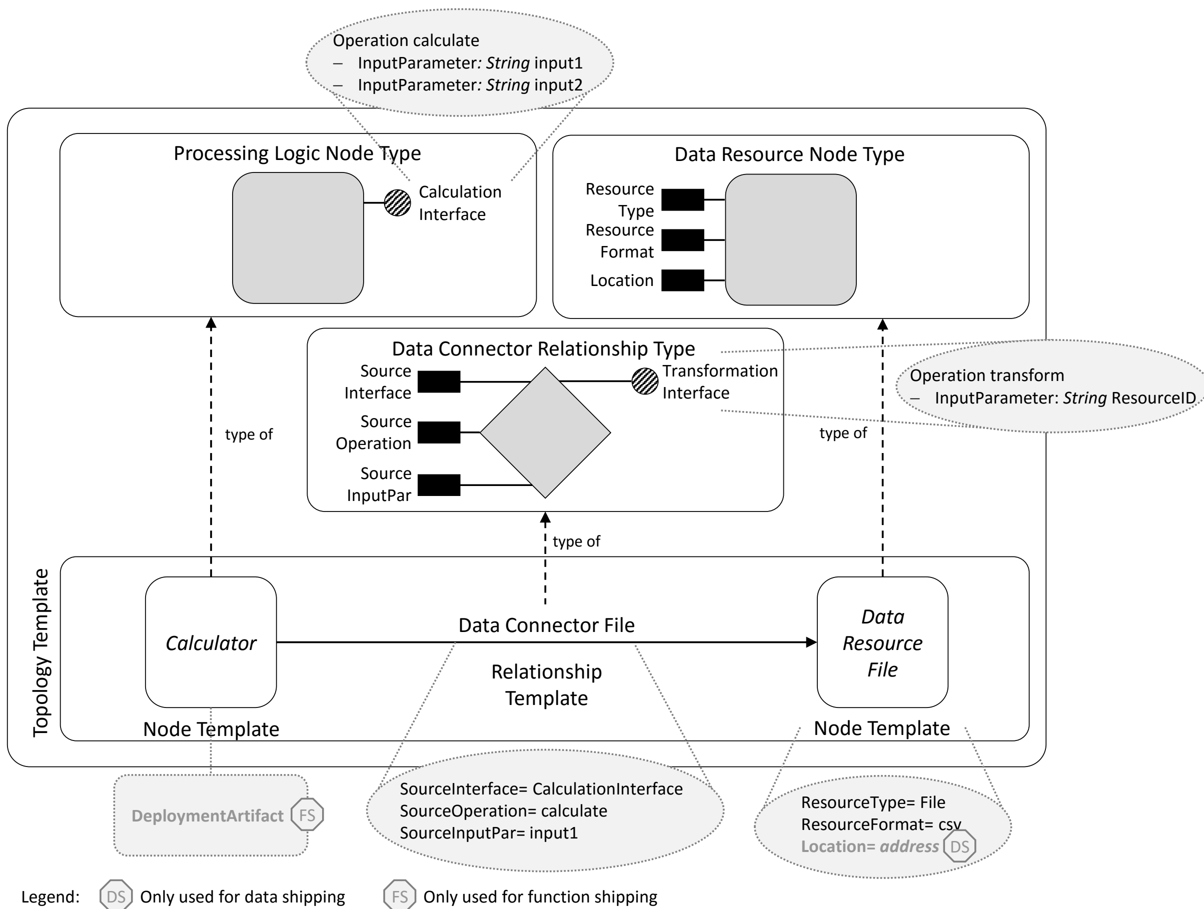



**Figure 5.20:** Option 6.2: TOSCA realization with transformation assignment via transformation interface

parameter. Contrary to an Implementation Interface defined for a Relationship Type, the Application Interface provides an operation, which is invoked for each processing logic execution. The actual implementation of the Application Interface is referenced by a Deployment Artifact which is part of the Relationship Type Implementation. It follows the same principles as described for Application Interfaces exposed by Node Types [Zim16]. However, the implementation are not further discussed in this thesis. In this example, a Topology Template containing a Node Template *Calculator*, specified in Listing 5.11, defined by the Node Type *ProcessingLogic* and a Node Template *DataResourceFile* of type *DataResource*, is defined. The Node Template *DataResourceFile* represents a file with the file format CSV. The two Node Templates are connected by the Relationship Template *DataConnectorFile* of type *DataConnector*. More precisely, it connects the input parameter *input1* of the processing logic to the data. The usage of the elements, shown grayed-out and marked with an icon, depends on the use case the option is used for. In case of function shipping, a Deployment Artifact is required to

**Listing 5.25** Option 6.2: Relationship Type definition *DataConnector*

```
1 <RelationshipType name="DataConnector">
2   ...
3   <PropertiesDefinition element="DataConnectorProperties"/>
4   <opentosca:ApplicationInterfaces
        xmlns:opentosca="http//www.uni-stuttgart.de/opentosca">
5     <Interface name="TransformationInterface">
6       <Operation name="transform">
7         <InputParameters>
8           <InputParameter name="ResourceID" type="xs:string"/>
9         </InputParameters>
10      </Operation>
11    </Interface>
12  </opentosca:ApplicationInterfaces>
13 </NodeType>
```

reference the actual deployment artifact of the processing logic. In case of data shipping, the property *Location* is used instead of the Deployment Artifact to refer to the actual data. Thus, this option can be applied for all data and function shipping use cases.

*Result*

This TOSCA realization option can be used for data and function shipping in the same way as Option 6.1. It can be used flexibly for all data and function shipping use cases illustrated in Fig. 4.2 and Fig. 4.3.

*Assumptions and restrictions*

Due to the definition of the transformation operation as Application Interface for the Relationship Type, the operation is the same for all Relationship Templates of this type. In case different transformation operations are required for different connections multiple Relationship Types have to be defined. Option 6.1 enables assigning the transformation operation on the level of Relationship Templates instead of Relationship Types. Furthermore, the connection between processing logic and data resource is restricted to a 1:1 relationship due to the TOSCA specification. In Concept 7 and Concept 8 is shown, how aggregated data collection and multiple Relationship Templates can be used. The following assumptions are made in order to apply this option for data and function shipping:

- Knowledge about the processing logic as well as the data is required. In addition, transformation requirements have to be known.
- The processing logic in case of data shipping, or the data in case of function shipping, are already available in the target runtime environment.

- The target runtime environment provides substitution mapping capabilities for the mapping of the abstract Node Template with the respective concrete Node Template already available in the target runtime environment.

*Extensions*

The above-mentioned option can be realized with the TOSCA Application Interface extension. Besides the Node Type representing the processing logic the Relationship Type is extended by an Application Interface. Originally, the Application Interface extension is made for Node Types to model the provided operations in TOSCA. In this case the Relationship Type *DataConnector* requires operations, which are executed in case of an processing logic invocation. Management Interfaces are not suitable for this because they are used to manage the relationship in terms of for example establishing the relationship, whereas transformation operations are functionality executed for each invocation. Therefore, an Application Interface is attached to a Relationship Type. The data have to be linked to the specific input parameter specified by the Relationship Template and for each processing logic invocation the transformation operation is executed. Furthermore, the runtime environment has to perform the substitution mapping of the Node Templates.

### 5.6.3 Option 6.3: TOSCA Realization with Explicitly Modeled Data Query

In contrast to Option 6.1 and Concept 6.2, this option focuses solely on data queries. The Service Template is illustrated in Fig. 5.21. The Node Types *ProcessingLogic* and *DataResource* are defined as described for Option 6.1. The Relationship Type *DataConnector* gets a new property *DataQuery* to integrate a data query in the model.

*Description*

Two Node Types are defined: *ProcessingLogic* and *DataResource*. The Node Type *ProcessingLogic*, defined in Listing 5.17, provides an operation *calculate*, which exposes two input parameters via the Application Interface *CalculationInterface*. For the Node Type *DataResource* three properties are defined in Listing 5.23. The properties *ResourceType* and *ResourceFormat* identify what kind of data are represented by the corresponding Node Template and in which format the resource is available. The actual data are referenced by the property *Location*. The Relationship Type *DataConnector* defines four properties (Listing 5.21). *SourceInterface*, *SourceOperation*, and *SourceInputPar* specify the concrete input parameter of the processing logic the data are used for. The values of these properties have to correspond with the name of the interface, operation

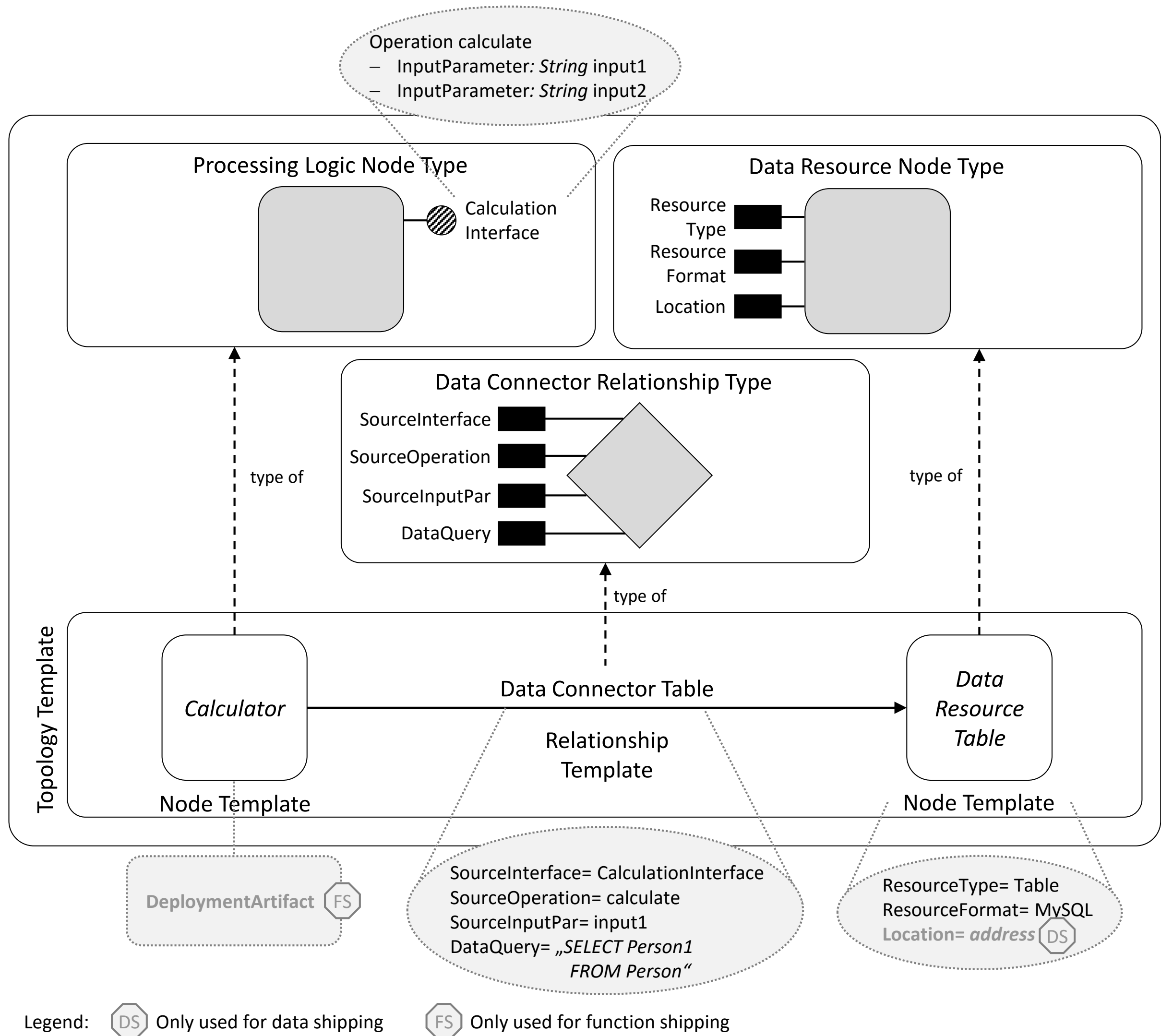



**Figure 5.21:** Option 6.3: TOSCA realization with explicitly modeled data query

and input parameter defined for the Node Type Processing Logic by an Application Interface element. In this modeling scenario the data serve as input for the input parameter *input1*, which is exposed by the operation *calculate*. An additional property *DataQuery* of type "string" is added to these properties. The property enables the modeler to specify a query directly in the Topology Template. In this example a database table is represented by the Node Template *DataResourceTable* and the property *DataQuery* contains a SQL statement, which should be executed on the data resource. A SQL statement constitutes just one example to express a query. Several other data query languages,

such as XQuery[1] for XML documents, Contextual Query Language[2] for search engines and bibliographic catalogs, or NoSQL database specific methods, are conceivable. The elements *DeploymentArtifact* and *Location*, which are grayed-out and marked with an icon, are used either in case of data shipping or in case of function shipping. A Deployment Artifact is only contained in case of function shipping and not in case of data shipping. For the property *Location* it is exactly the other way around, it is only used in case of data shipping.

*Result*

Data shipping as well as function shipping can be realized by this option. The differences, which have to be taken into account are described in detail for Concept 6.1 and apply also to this option.

In case of data shipping use case (b) in which a reference to a remote data location is shipped, the query language which can be used depends on the data resource capability the query should be applied to. For instance, SQL can just be used in case the remote data resource can process SQL statements. Thus, the query is shipped to the data resource location and the result data are transferred to the processing logic. Data shipping in terms of shipping the model with the reference to the data resource location and the later retrieval of the data are realized. However, selection logic and business logic are separated and the selection logic are shipped to the data resource. This corresponds with the query shipping approach explained in Section 2.2. In case of data shipping use case (a) in which the data are stored and shipped in the archive, it depends on the target runtime environment and the data management system which may contained in the archive if queries can be executed on the data resource.

*Assumptions and restrictions*

As seen before, a Relationship Template can just connect two Node Templates to each other. How multiple data resources can be connected to a processing logic and visa versa is shown in Concept 7 and Concept 8. The usage of this option is strongly influenced by the possibility to use data queries for a specific data resource. An explicit SQL statement, as presented in Fig. 5.21, is limited to database systems, which can process SQL statements. Other query languages are already mentioned above. To enable data shipping and function shipping the following general assumptions are made:

- A modeler with knowledge about the processing logic as well as the data is required. Additionally, the modeler has to know how to write data queries and which data query language can be used in each situation.

[1]https://www.w3.org/TR/xquery/
[2]http://www.loc.gov/standards/sru/cql/spec.html

- The processing logic in case of data shipping, or the data in case of function shipping, are already available in the target runtime environment.
- The target runtime environment provides substitution mapping capabilities for the mapping of the abstract Node Template representing the processing logic or data resource with the respective concrete Node Template already available in the target runtime environment.
- Either an external system or the target runtime environment has to be able to process the data query.

*Extensions*

The TOSCA metamodel has to be extended by the Application Interface extension introduced by Zimmermann [Zim16] to explicitly model the operation of the Node Type *ProcessingLogic*. Furthermore, it is important to note the domain specific usage of the existing elements. The Node Type *DataResource* represents data and is connected by the Relationship Type *DataConnector* to a determined input parameter of the Node Type *ProcessingLogic*. In addition, a data query is written and attached to the *DataConnector*. The data query has to be executed for every processing logic invocation with these data. Thereby, a caching mechanism is possible to cache frequently requested data close to the processing logic location. Caching mechanism are not further discussed in this thesis. The target runtime environment has to be able to process the Service Template and to execute the substitution mapping between an abstract Node Template representing the processing logic, or the data, respectively, and a concrete Node Template.

## 5.7 Concept 7: Data Connector between Processing Logic and Data Collection

This concept is an advancement of Concept 5. The main subject of this concept is the integration of a data collection consisting of several data resources in the model. Instead of connecting single data resources with the processing logic, a set of data resources can be linked.

*Context*

Either the data owner wants to ship his or her data or the processing logic owner his or her function. The data resources are part of the same data collection. The data collection itself is just a structuring element, which is used to organize and structure multiple data resources. A data collection can be for example a database with several tables or a directory with several files. For the processing logic several data resources

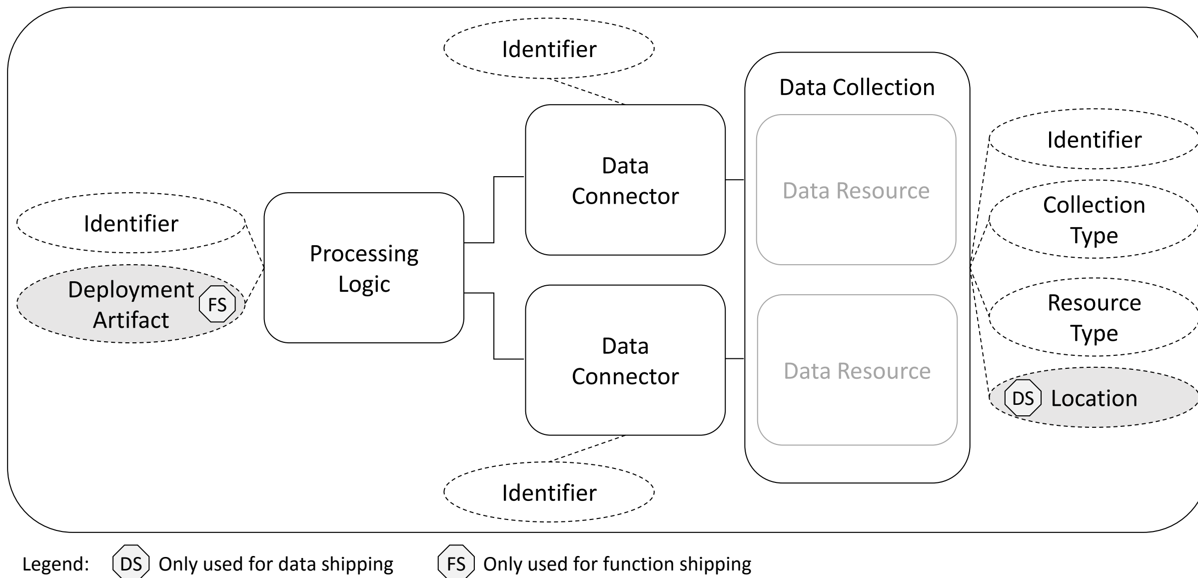



**Figure 5.22:** Concept 7: Data connector between processing logic and data collection

are required as input for the data processing. These data resources are part of the same data collection, i.e., different tables of the same database or different files in a folder. With Concept 5 and Concept 6 only separate data resources can be modeled and transformation operations as well as queries address just one data resource, e.g., one table or one file. To reduce the complexity of the model and to reduce the number of modeled data resources, the required data resources should be modeled as an aggregated data collection.

*Problem*

How can the modeler specify the relationship between processing logic and data resources, which are part of the same data collection?

*Solution*

The modeling concept is outlined in Fig. 5.22. On the left side, the *Processing Logic* entity has an attribute *Identifier* and an attribute *Deployment Artifact*, which references the actual processing logic. On the right side, the *Data Collection* representing a data collection of several data resources is specified by four attributes. The *Identifier* uniquely identifies the Data Collection. *Collection Type* and *Resource Type* add further information about the Data Collection. The *Collection Type* can be for example a database and the related *Resource Type* a table. These attributes can be used by the runtime environment for automated transformation operations or to identify information required to facilitate the communication with the data collection or data resource, respectively. The last attribute *Location* references the actual data collection containing several data resources.

The icons *FS* and *DS* mark the attributes, which are only used in case of function shipping or data shipping. Concept 5 and Concept 6 demonstrate how a *Data Connector* connects the *Processing Logic* to a single *Data Resource*. In this concept, the relationship exists between a single processing logic and a data collection. Thus, several data resources can be linked to a processing logic without modeling every single data resource. However, for each *Data Resource* linked to the *Processing Logic* a separate *Data Connector* is required. An alternative is to model each data resource and to link them to the processing logic separately. This is not illustrated in Fig. 5.22, but is discussed in detail in Option 7.1. Concept 7 can be used for data shipping and function shipping, in each case for both use cases: shipping of the actual data or processing logic and shipping of a reference to a remote location in an archive.

*Implications*

The *Data Collection* represents an addressable set of data resources, which are part of a common data collection structure. This can be, for instance, a database or file folder. The *Data Collection* as shown in Fig. 5.22 replaces several *Data Resource* elements each connected to the *Processing Logic*. If additional transformation operations are required, they can be added to the Data Connector as shown in Concept 6. In case the required data resources are contained in different data collection, multiple data collections can be modeled. How to combine or to apply one operation to multiple data resources, e.g., a SQL join operation, is shown in Concept 8.

### 5.7.1 Option 7.1: TOSCA Realization with Separated Data Resources

This TOSCA realization option with separated data resources, each connected to the processing logic, is depicted in Fig. 5.23. It is based on the elements of Concept 5 and demonstrates how multiple individual data resources can be connected to different input parameters of the same processing logic. This is an alternative to the modeling approach with a data collection. It is an important alternative in case the data resources are not part of the same data collection.

*Description*

Two Node Types *ProcessingLogic* and *DataResource* and a Relationship Type *DataConnector* are defined. An Application Interface *CalculationInterface* is provided by the Node Type *ProcessingLogic*, as defined in Listing 5.17. An operation *calculate*, which exposes two input parameters can be used. The properties of the Node Type *DataResource*, illustrated in Listing 5.3, are defined in Listing 5.23. The two properties *ResourceType* and *ResourceFormat* identify what kind of data are represented. The actual data are referenced by means of the property *Location*. The properties of the Relationship Type *DataConnector* defined in Listing 5.20 and Listing 5.21 are required to specify the exact

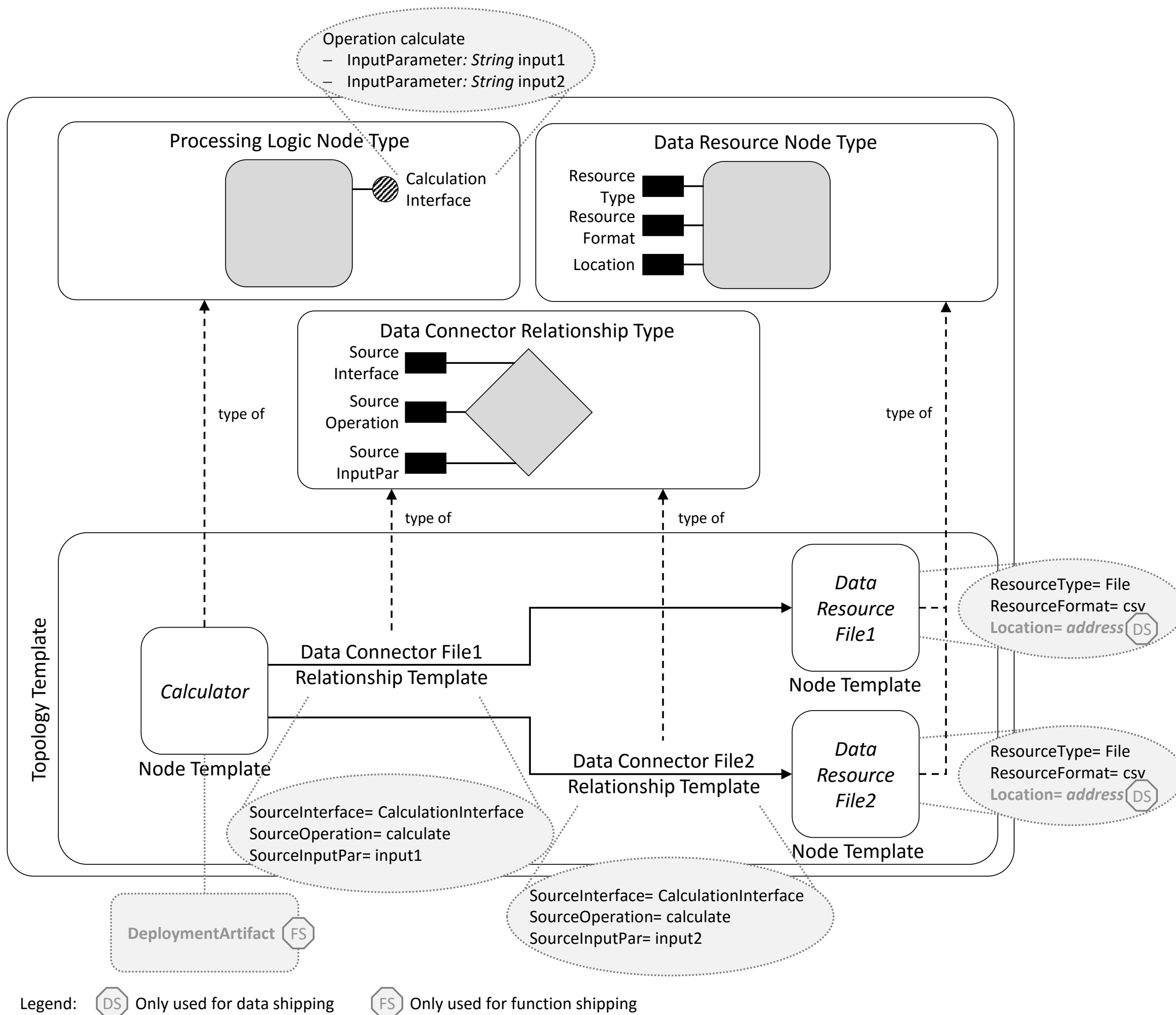



**Figure 5.23:** Option 7.1: TOSCA realization with separated data resources

input parameter the data should be used for. The Topology Template contains a Node Template *Calculator* of type *ProcessingLogic* and two Node Templates defined by the Node Type *DataResource*, each representing a different file. For each connection between the Node Template *Calculator* and the Node Templates representing the data a separate Relationship Template of type *DataConnector* is required. In this example, the Node Template *DataResourceFile1* is linked to the input parameter *input1* of the operation *calculate* and the other Node Template is linked to the input parameter *input2*. The usage of the elements marked with the icons *FS* and *DS* depends on the use case the option is used for. The Deployment Artifact is required in case of function shipping. It references the actual deployment artifact of the processing logic. In case of data shipping the property *Location* instead of the Deployment Artifact is required to reference the actual data.

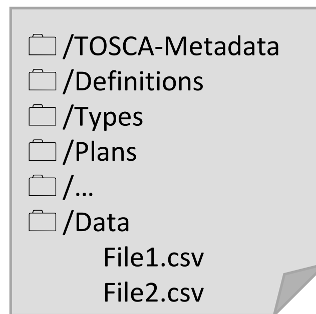



**Figure 5.24:** Exemplary structure of an CSAR containing two different files

*Result*

As mentioned above, this option can be used for data shipping and function shipping, in each case for the shipping of the actual data or function as well as the shipping of the reference to a remote location. For the realization the following has to be considered:

1. Data shipping: the property *Location* of each Node Template of type *DataResource* points to the actual data. For the shipping of the actual data an exemplary CSAR is shown in Fig. 5.24. The data are contained in the CSAR. The corresponding values for the property *Location* are "Data/File1.csv" for Node Template *DataResourceFile1* and "Data/File2.csv" for Node Template *DataResourceFile2*. If the data are available at a remote location, the properties reference these locations. Since the processing logic is already available in the target runtime environment in the case of data shipping, the modeled Node Template *Calculator* is only an abstraction and has to be mapped to the already existing Node Template in the runtime environment. The Node Template *Calculator* has to be declared as abstract and thus no instance of it can be created. The Node Template has to be substituted in the target runtime environment by one having a specialized, derived Node Type.
2. Function shipping: this case is exactly the other way around. Instead of the data, the processing logic is shipped with the CSAR or the remote location, where the processing logic is stored, is referenced in the Deployment Artifact. An exemplary CSAR for the shipping of the actual function in an archive is shown in Fig. 5.11. In this case, the Node Templates representing the data are an abstraction and have to be substituted in the runtime environment where the actual data are already available. The substitution method is the same as for data shipping.

*Assumptions and restrictions*

Each data resource has to be modeled separately. If there is a large number of data resources required, the model becomes very complex. The aggregation of several data resources in one data collection is shown in the Optione 7.2. But separately modeled

data resources can solely be replaced by a *DataCollection* if the resources belong to the same data collection. In order to apply this TOSCA realization option the following assumptions are made:

- The modeler requires additional knowledge about the data in case of function shipping and about the processing logic in case of data shipping.
- The data or processing logic, depending on the shipping use case, is already available in the target runtime environement.
- The substitution mapping of the abstract and the concrete Node Templates can be performed.
- Further transformation operations have to be executed by the target runtime environment or the processing logic itself.

*Extensions*

For modeling the Application Interface of the Node Type *Processing Logic*, the TOSCA extension introduced by Zimmermann [Zim16] is required. Also the domain specific elements have to be correctly interpreted. The referenced data by the Node Templates of type *DataResource* serve as input for the input parameters specified by Relationship Templates defined by the Relationship Type *DataConnector*. For each invocation of the processing logic's operation the linked data are used as input for the execution.

### 5.7.2 Option 7.2: TOSCA Realization with Single Data Collection

In this TOSCA realization option, the data resources which belong to the same data collection are aggregated. In contrast to Option 7.1, the two Node Templates, each representing one data resource, are replaced by one Node Template *DataCollectionFileSystem*, which represents the entity that contains the data resources. The corresponding Service Template is shown in Fig. 5.25.

*Description*

The Node Type *ProcessingLogic* is defined in Listing 5.17. It provides an Application Interface *CalculationInterface* that exposes the operation *calculate* and two input parameters. Instead of a Node Type *DataResource*, the Node Type *DataCollection* is used, which is defined in Listing 5.26 with the attached properties in Listing 5.27. The properties *CollectionType* and *ResourceType* are defined in lines 4 and 5 (Listing 5.27). These properties characterize the data collection. For a Node Template of type *DataCollection* the *CollectionType* can be for example specified as "directory", while the *ResourceType* is determined as "file". This is shown in the example in Listing 5.28. The third property

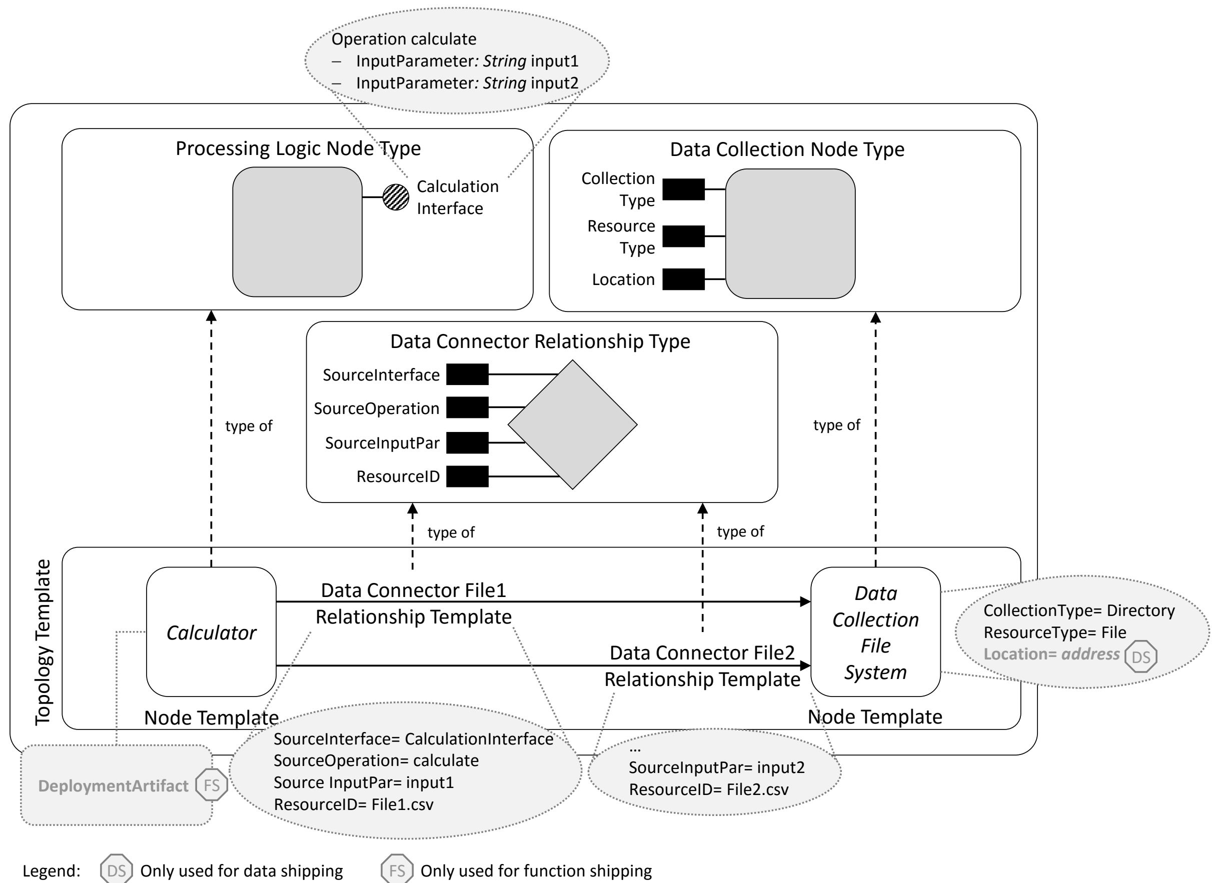



**Figure 5.25:** Option 7.2: TOSCA realization with single data collection

*Location* references the actual data collection, which can be either in the CSAR that contains the Service Template or it is available at a remote location. The Relationship Type *DataConnector* shown in Listing 5.20 represents the connection between the processing logic and the data. With the first three properties *SourceInterface*, *SourceOperation* and *SourceInputPar*, defined in Listing 5.21, the exact input parameter of the processing logic can be specified. For this option an additional property *ResourceID* of type "string" is required, which determines a concrete data resource inside the data collection. The value of this property should map with the name used for this data resource in the data collection, by convention. In this example two files "File1.csv" and "File2.csv" are contained in the data collection and referenced by the property *ResourceID* of the Relationship Templates. In Fig. 5.25 the Topology Template contains a Node Template *Calculator* of Node Type *ProcessingLogic* and a Node Template *DataCollectionFileSytem* defined by the Node Type *DataCollection*. The *DataCollectionFileSystem*, shown in Listing 5.28, represents a directory containing several files. For each relationship between an input parameter and a file one Relationship Template of type *DataConnector* is used. It specifies which specific data resource should be used for a certain input parameter. As

**Listing 5.26** Node Type definition *DataCollection*

```
1 <NodeType name="DataCollection">
2   ...
3   <PropertiesDefinition element="DataCollectionProperties"/>
4 </NodeType>
```

**Listing 5.27** Properties definition for Node Type *DataCollection*

```
1 <xs:element name="DataCollectionProperties">
2   <xs:complexType>
3     <xs:sequence>
4       <xs:element name="CollectionType" type="xs:string"/>
5       <xs:element name="ResourceType" type="xs:string"/>
6       <xs:element name="Location">
7         <xs:complexType>
8           <xs:attribute name="ref" type="xs:string"/>
9         </xs:complexType>
10      </xs:element>
11    </xs:sequence>
12  </xs:complexeType>
13 </xs:element>
```

mentioned before, the usage of the grayed-out elements marked with the icons depends on the shipping use case the option is used for.

*Result*

Both shipping use cases, data shipping and function shipping, can be realized with this option. The following must be considered:

1. Data shipping: the processing logic is already available in the target runtime environment. The Node Template *Calculator* in the Topology Template is only an abstraction and is substituted in the target runtime environment. The substitution

**Listing 5.28** Node Template *DataCollectionFileSystem*

```
1 <NodeTemplate id="DataCollectionFileSystem" name="Data Collection File System"
      type="DataCollection">
2   ...
3   <Properties>
4     <DataCollectionProperties>
5       <CollectionType>Directory</CollectionType>
6       <ResourceType>File</ResourceFile>
7       <Location ref="address"/>
8     </DataCollectionProperties>
9   </Properties>
10 </NodeTemplate>
```

mapping is described in detail in Option 7.1. For a shipping of the actual data, an exemplary CSAR is shown in Fig. 5.24. For this example the property *Location* has the value "Data", which is the subdirectory in the CSAR containing all data files. The property *ResourceID* of the Relationship Templates *DataConnectorFile1* and *DataConnectorFile2* indicates the specific data resource contained in the directory. In this case "File1.csv" and "File2.csv".

2. Function shipping: for function shipping the Node Template representing the data collection is declared as abstract and has to be substituted in the target runtime environment where the data collection is already available. A Deployment Artifact for the Node Template *Calculator* is defined, which references either the actual artifact contained in the CSAR or the artifact available at a remote location.

*Assumptions and restrictions*

The aggregation of several data resources to one data collection is restricted to data resources, which are part of a common addressable collection. Data resources, which are hierarchically subordinated to the data collection can be linked to the processing logic. In this option the connection between processing logic and data resource is still limited to a 1:1 relationship. Concept 8 describes the join of multiple resources for a single input parameter. This option bases on the same assumption as Option 7.1. The processing logic as well as the data resources have to be known. The substitution mapping and further adaptations are performed by the target runtime environment or processing logic, respectively.

*Extensions*

The TOSCA extension introduced by Zimmermann [Zim16] is required for the Application Interface of the processing logic. The Node Template *Data Collection* represents the data collection, which contains the related data resources. The individual data resources are addressed by the property *ResourceID* of the Relationship Template. For each operation invocation the linked data resources have to be used as input for the execution. Furthermore the substitution mapping has to be done by the runtime environment to map the abstract Node Templates to the appropriate Node Template available in the runtime environment.

## 5.8 Concept 8: Data Connector with Operations Applied to Multiple Data Resources

This concept combines the elements of Concept 6 and Concept 7. In contrast to Concept 6, operations are applied not only to one data resource, but to multiple data resources,

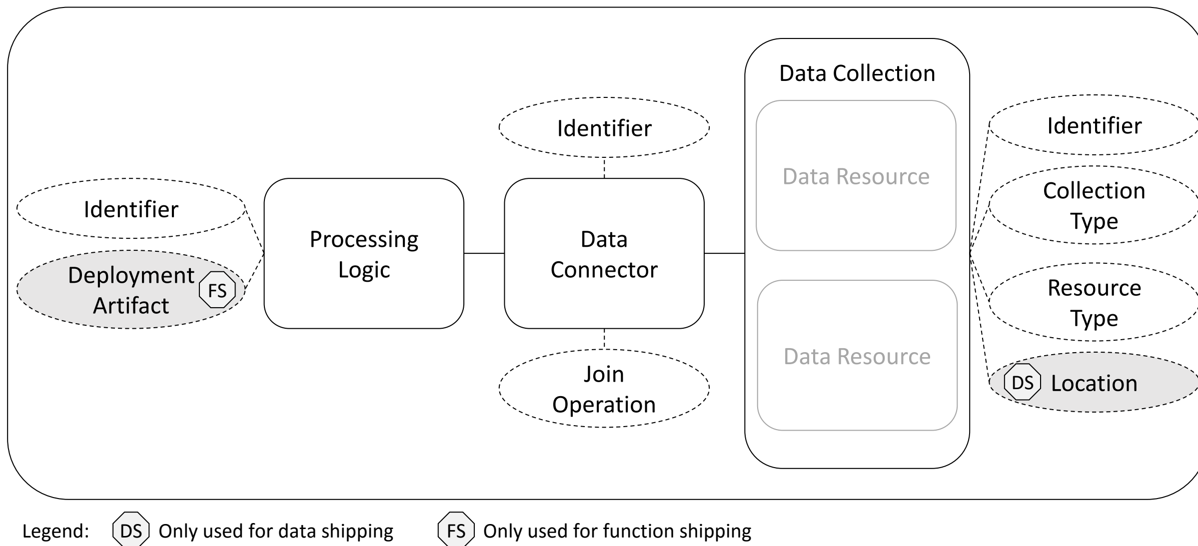



**Figure 5.26:** Concept 8: Data connector with operations applied to multiple data resources

which can be modeled as data collection in case they belong to the same collection. The main purpose is to model operations which apply to multiple data resources and to connect a set of data resources to a processing logic.

*Context*

The data required for one execution of a processing logic are spread over several data resources contained in a common logical entity. Before they can serve as one single input for the processing logic the data have to be joined. This can be for example a join operation such as a SQL statement or the union of several flat files to one single file. Maybe the data resources not only have to be joined, but additional transformation operations are required. Either the logical entity containing the data resources or the processing logic should be shipped.

*Problem*

How can the modeler specify the join of several data resources of the same data collection?

*Solution*

Fig. 5.26 illustrates the modeling concept. The *Processing Logic* on the left side is uniquely addressable by an *Identifier*. The *Deployment Artifact* references the actual processing logic. The *Data Collection* on the right side comprise several *Data Resource* elements. It is addressable by an unique *Identifier* and characterized by the attributes *Collection Type* and *Resource Type*. The *Collection Type* specifies what kind of collection it represents.

This could be for example a database or directory. The *Resource Type* identifies the type of data resources contained in the data collection. The *Location* attribute references the actual data collection. *Processing Logic* and *Data Collection* are linked by the *Data Connector*. The *Data Connector* has an *Identifier* and a *Join Operation* assigned. The *Join Operation* attribute references a specific operation, which can join different data resources together to provide a single input for the processing logic. This could be for example a join of multiple database tables or a join of multiple flat files. The difference to Concept 6 is that one operation executes several data resources. The join operation is assigned to the *Data Connector* and not to the *Data Collection* on purpose. For instance, several processing logics require joined data consisting of data resources contained in the same data collection. However, each processing logic based on an other subset of data contained in the data collection. Thus, the data collection has to be modeled only once and different operations to join the data can be assigned to the *Data Connector*. The usage of the grey-shaded attributes *Deployment Artifact* and *Location* depends on the shipping use case, which should be realized by the modeling concept. The concept can be used for function shipping as well as data shipping. In case of function shipping, the *Deployment Artifact* is required to reference the actual processing logic. In case of data shipping, the *Location* attribute identifies the actual data.

*Implications*

Multiple data resources contained in a data collection can be joined together by means of the join operation of the data connector and can serve as input for the processing logic. The modeling concept as presented in Fig. 5.26 can solely be used in case all data resources, which should be joined relate to the same logical entity. An alternative approach to implicitly join data resources not contained in the same data collection is presented in Option 8.2. This option is not shown in this concept description.

### 5.8.1 Option 8.1: TOSCA Realization with Join Operation Assignment via Join Interface

Fig. 5.27 illustrates this TOSCA realization option. A Node Type representing a processing logic and another Node Type representing a data collection are defined. The join operation is defined as an Application Interface of the Relationship Type *DataConnector*. A Relationship Template of type *DataConnector* can be specified to connect a processing logic to a data collection and to join data contained in the data collection.

*Description*

Two Node Types are defined: *ProcessingLogic* and *DataCollection*. The Node Type *ProcessingLogic* depicted in Listing 5.17 provides an Application Interface

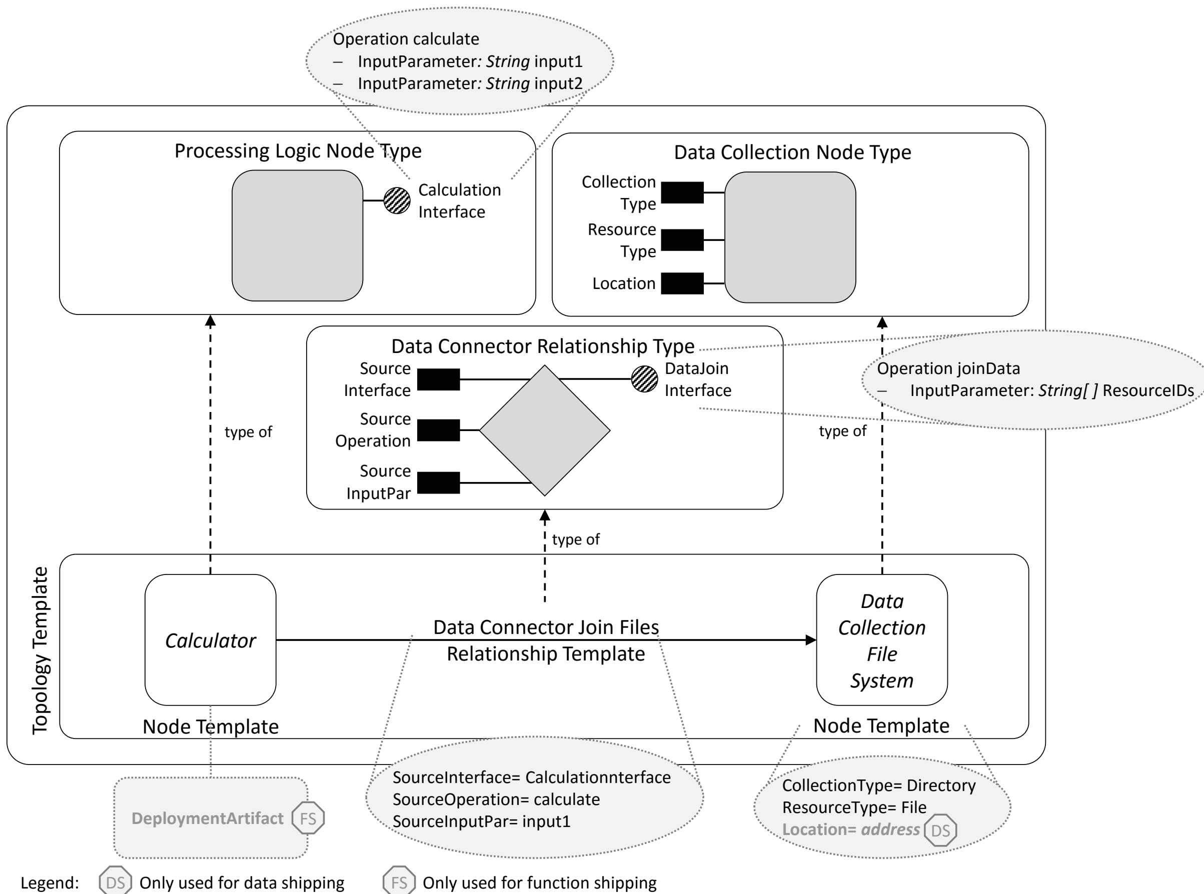



**Figure 5.27:** Option 8.1: TOSCA realization with join operation assignment via join interface

*CalculationInterface*. The operation *calculate* requires two input parameters. The Node Type *DataCollection*, shown in Listing 5.26, is characterized by three properties defined in Listing 5.27. *CollectionType* and *ResourceType* are used to identify what kind of data collection is represented. The last property *Location* references the actual data collection. The Relationship Type *DataConnector* defines the connection between a Node Template of type *ProcessingLogic* and a Node Template defined by the Node Type *DataCollection*. The three properties attached to the *DataConnector*, presented in Listing 5.21, are used to specify the input parameter of the processing logic, the data are used for. The Application Interface *DataJoinInterface* assigned to the Relationship Type provides an operation *joinData* with an Array of ResourceIDs as input. In this example, this operation is able to join an arbitrary number of flat files to one single file and to convert the file formats, if necessary. The Resource IDs identify the data resources comprised in the data collection, which should be joined to serve as input for the processing logic. In this example based on the modeling scenario presented in Chapter 4, the Topology Template contains a Node

Template *Calculator* and a Node Template *DataCollectionFileSystem*. The Relationship Template *DataConnectorJoinFiles* connects the processing logic and the data collection and joins "File1.csv" and "File2.csv" together. The joined file serves as input for the input parameter *input1*. The usage of the grayed-out attribute *Location* and the Deployment Artifact depends on the shipping use case the option is used for. The attribute *Location* is used in case of data shipping and the Deployment Artifact is required for function shipping.

*Result*

This TOSCA realization option enables data shipping as well as function shipping. Some features have to be taken into account in order to use this option:

1. Data shipping: the processing logic, which should process the data is already available in the target runtime environment. The data can be contained in the CSAR or has to be available at a remote location. An exemplary CSAR for the shipping of the actual data in the archive is shown in Fig. 5.24. The data collection represents the subdirectory, which contains the data. Therefore, the placeholder *address* used for the *Location* attribute is replaced by "Data". The two files "File1.csv" and "File2.csv" are joined and serve as input for the input parameter *input1*. The Node Template *Calculator* contained in the Topology Template is declared as abstract. It has to be substituted by an Node Template having a specialized, derived Node Type in the target runtime environment.
2. Function shipping: in this case the data, which should be processed have to be already available in the target runtime environment. The Deployment Artifact is required to ship the actual artifact, which is contained in the CSAR or located at a remote location. The described substitution has to be executed for the Node Template *DataCollectionFileSystem*.

*Assumptions and restrictions*

The required join operation depends on the type and the structure of the data resources, which should be joined. For instance, for the join of CSV files another operation is required than for the join of JSON files or database tables. The use of queries to join tables in a database are discussed later. The join capability is restricted to data resources contained in the same logical entity due to the fact that a Relationship Template can only realize an 1:1 relationship. The Relationship Template can just connect one data collection to one processing logic. To enable the realization the following assumptions are made:

- The modeler requires knowledge about the data as well as the processing logic. Furthermore, the specific data resources and how they can be joined together are known.

- Either the data collection or the processing logic is already available in the target runtime environment depending on the shipping use case.
- The target runtime environment is able to perform the substitution mapping.

*Extensions*

The TOSCA extension for Application Interfaces is required to realize this option. Originally, the extension is introduced for Node Types to define the operation provided by an application in TOSCA. In this case, this approach is also used to define a join operation for the Relationship Type. In contrast to the management interfaces, which are already specified for Relationship Types in TOSCA, the provided join operation is executed for each processing logic invocation the assigned data are used for. The runtime environment has to execute the join operation and link the data to a particular input parameter specified by the Relationship Template. Often requested data can be cached close to the processing logic and do not have to be retrieved from the original data location each time. Furthermore, the runtime environment performs the substitution mapping of the Node Templates.

Option 6.2 follows the same approach as Option 8.1 to assign an operation to the Relationship Type. An alternative approach to assign an operation is shown in Option 6.1. A property is specified for the Relationship Type to refer to the actual artifact of the operation. In the same way the join operation can be assigned to a Relationship Template. The advantage of the alternative approach is a higher flexibility in terms of the assignment of different join operations. If the join operation is assigned via a property, different operations can be attached on the level of Relationship Templates. If the join operation is defined as an Application Interface for a Relationship Type, the same operation applies for all Relationship Templates of this type.

Also the realization of a data join by an explicit modeled data query as described in Option 6.3 is possible. For instance, an SQL statement to join data from multiple tables could be explicit modeled. Especially for the data shipping use case (b) in which a reference to a remote data location is shipped this variant is suitable. The data owner ships the reference together with a data query, which can be processed by the remote data collection to the processing logic owner. If the processing logic is executed, the query is sent to the data collection and the joined data are returned to the processing logic.

### 5.8.2 Option 8.2: TOSCA Realization with Implicit Join Operation

This option does not comply with Concept 8. It is an alternative approach to realize an implicit join operation, illustrated in Fig. 5.28. The join operation is not part of the

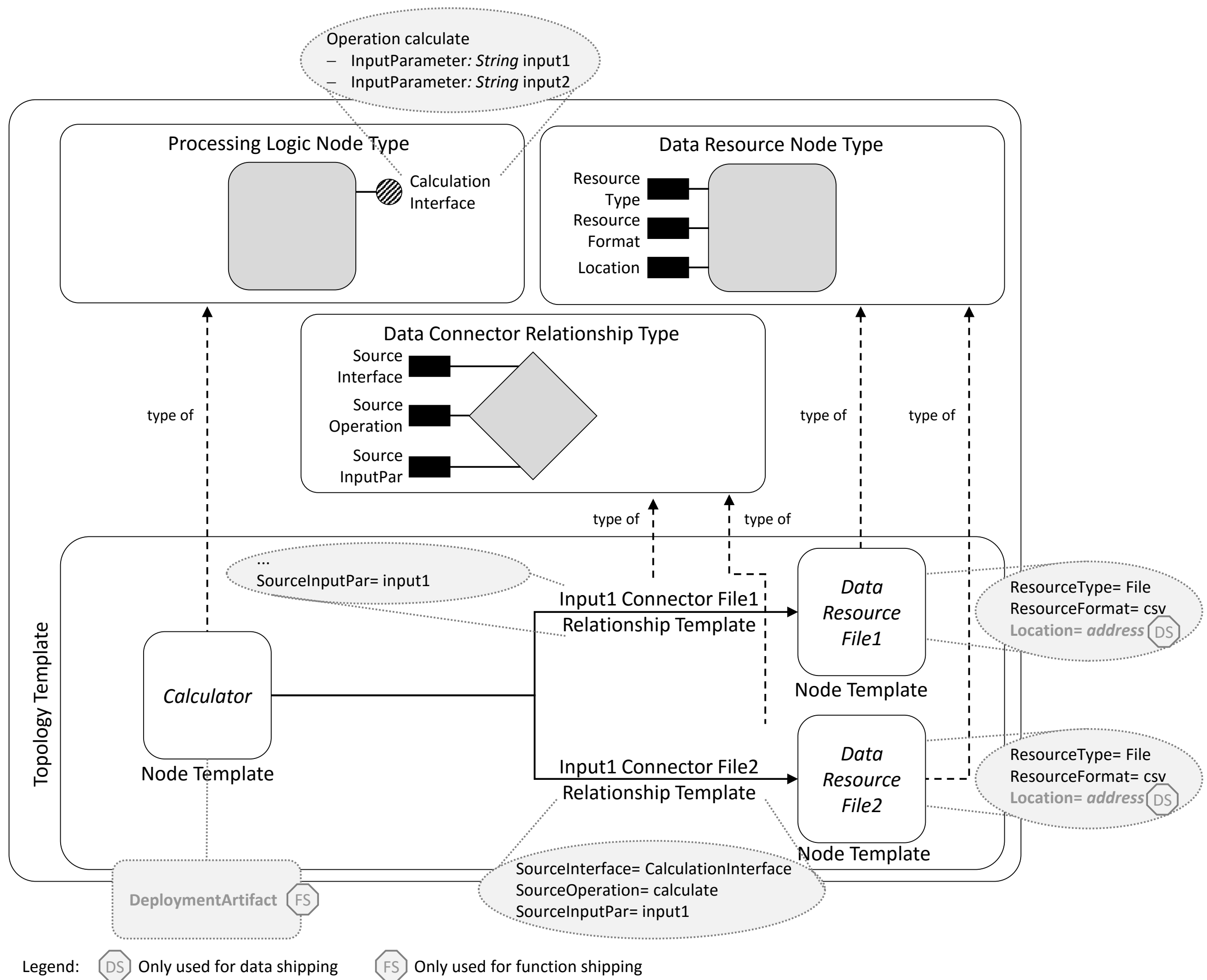



**Figure 5.28:** Option 8.2: TOSCA realization with implicit join operation

model but provided by the runtime environment. The data resources are not comprised in the same logical entity and modeled as individual *Data Resources*.

*Description*

A Node Type *ProcessingLogic* with an Application Interface providing the operation *calculate* is defined in Listing 5.17. In contrast to Option 8.1, a Node Type *DataResource* is defined presented in Listing 5.3 with the attached properties declared in Listing 5.23. The property *ResourceType* specifies what kind of resource is represented by a particular Node Template of this Node Type. This could be for example a file or a table. The property *ResourceFormat* indicates the storage format and the *Location* references the actual data. The Relationship Type *DataConnector* indicates for which input parameter a data resource should be used for. The definition is shown in Listing 5.20 and Listing 5.21. In this example, three Node Templates are specified: One representing a processing logic

called *Calculator* and two Node Templates representing different data resources. Both Node Templates representing the data are linked to the same input parameter *input1* provided by the processing logic. Assigning multiple data resources to the same input parameter implicitly indicates a join in this scenario. A join can be for example the union of several flat files or a join of tables in a database. The required join operation is provided by the runtime environment. The usage of the grayed-out Deployment Artifact and the property *Location* assigned to each Node Template of type *DataResource* depends on the shipping use case. The Deployment Artifact is only used in case of function shipping and the properties in case of data shipping.

*Result*

As mentioned above, both shipping use cases, data shipping and function shipping, can be realized with this option. The following features of the different use cases have to be considered:

1. Data shipping: the processing logic is already available in the target runtime environment. The data can be shipped within the CSAR or has to be available at a remote location and just a reference is shipped to the target runtime environment. Fig. 5.24 shows an exemplary CSAR with two files in the subdirectory "Data". For this case, the values of the corresponding propertyies are "Data/File1.csv" for the Node Template *DataResourceFile1* and "Data/File2.csv" for the other one. Otherwise, it can reference any remote address. The same kind of substitution mapping as described for Option 8.1 has to be done for the Node Template *Calculator*.

2. Function shipping: for function shipping it is just the other way around than described for data shipping. The data are available in the target runtime environment and the Node Templates representing the data resources in the Topology Template have to be substituted. The Deployment Artifact is assigned to the Node Template representing the processing logic and references the actual artifact. It can either be contained in the CSAR or stored at a remote location.

*Assumptions and restrictions*

In contrast to Option 8.1, this option is not limited to data resources in the same logical entity. It is just restricted by the joining capabilities of the runtime environment. To enable the join in each case for all kinds of data resources, the runtime environment has to hold for example SQL joins as well as programms to merge CSV, JSON, or XML files. Additionally, the runtime environment requires a mechanism to determine which join operation has to be applied in a specific case. Since the join operation is not explicitly modeled, a common understanding of multiple links with one single input parameter is important. Furthermore a few assumptions are made to enable this option:

- Knowledge about the processing logic and the data is required for both shipping use cases.
- The data resources or the processing logic are already available in the target runtime environment depending on the shipping use case.
- The target runtime environment can execute the substitution mapping as well as the join operation. The join operation is a capability of the runtime environment and not part of the Service Template.

*Extensions*

Besides the already known Application Interface extension of TOSCA, no further extensions are required to realize this option. A processing logic and data resources are linked via a data connector, which means the data are used as input for the operation. In this option, multiple links to a single input parameter have to be interpreted by the runtime environment as an indicator that a join operation has to be applied.

However, multiple data connectors linking several data resources with a single input parameter can have more than one meaning. In the description above, multiple links are interpreted as an implicit join of the data. An alternative interpretation could be, that the data resources are required for different processing logic invocations. That means for example, for one invocation "File1.csv" and for another invocation "File2.csv" is used. This is a different approach and implies that decision variables exist to specify which data resource should be used in a specific case. This underlines the need of a common understanding of the semantics of multiple links to a single input parameter.

## 5.9 Modeling Concepts Summary

In total eight concepts and 16 options are presented, which can be used for data shipping, function shipping, or both. The main purpose of these concepts is to link processing logic and data together. For each concept the assignment of processing logic and data is realized in different ways. Therefore, depending on the situation's context, the most appropriate concept can be chosen.

However, not every concept can be used for data shipping and function shipping. Fig. 5.29 gives an overview of all concepts and shows which shipping use case can be realized by which specific concept. The first two concepts focus on the modeling of the data resources and are applicable for both data shipping use cases: the actual data are packaged together with the model and transferred to the target runtime environment, or the data remain at a remote location and a reference is shipped. In contrast, the main subject of Concept 3 and Concept 4 is the processing logic. They are exclusively

usable for the function shipping use cases. The remaining four concepts are suitable for data shipping as well as function shipping. These concepts share their emphasis on the connection between the processing logic and data. The processing logic as well as the data are modeled as separate components of the overall service connected by a data connector. Therefore, these concepts are flexible and adoptable to the requirements of different modeling scenarios, but presume modeler's knowledge about the data as well as the processing logic.

To reduce the complexity of the concepts and options, each concept introduces only one aspect, which is important in terms of linking processing logic and data together. The concepts and options can be combined to a certain extent. An example is the combination of Concept 6 and Concept 7. Concept 6 presents the transformation operation attached to the Data Connector. This transformation operation can of course also be used in Concept 7, which presents the modeling of a data collection instead of separate data resources. The same applies to various TOSCA realization options. In Option 1.1 the data are referenced by a Deployment Artifact attached to the Node Template representing the data resource. This approach can also be chosen for the other options. Due to the large numbers of options, only the approach introduced by Option 1.2 is adopted in the other options. This means, a property is used to reference the actual data. However, the other approach is also a valid solution.

Moreover, non-functional requirements or quality-of-service aspects can be attached to the *Processing Logic*, *Data Resource*, *Data Collection*, or *Data Connector*. The TOSCA spezification provides a Policy Type to specify such requirements, which have to be considered by the runtime environement [OAS13b]. Availability, network bandwidth, or data quality are examples for non-functional requirements. In terms of the data processing an appropriate data quality is important. Pipino et al. [Pip+02] and Batini et al. [Bat+06] discuss different dimensions of quality like accuracy, completeness, timeliness, volatility, and consistency. A closer consideration of non-functional requirements is beyond the scope of this thesis. Also technology specific aspects are not considered. Further attributes can be added to adapt the concepts to specific modeling scenarios.

The presented concepts outline possibilities to model data and processing logics including the relationship between them specifically. An adaptation of these concepts is not only conceivable for TOSCA but also for other situations, such as workflows in which data deployment and assignment are important.

| | Concepts \ Scenarios | **Data shipping** Data | **Data shipping** Reference to remote data location | **Function shipping** Function | **Function shipping** Reference to remote function location |
|---|---|---|---|---|---|
| **1** | Uniquely addressable data resource | x | x | | |
| **2** | Uniquely addressable data resource with assigned processing logic identifier | x | x | | |
| **3** | Uniquely addressable processing logic | | | x | x |
| **4** | Uniquely addressable processing logic with assigned data identifier | | | x | x |
| **5** | Data connector between processing logic and data resource | x | x | x | x |
| **6** | Data connector with transformation capability | x | x | x | x |
| **7** | Data Connector between processing logic and data collection | x | x | x | x |
| **8** | Data connector with operations applied to multiple data resources | x | x | x | x |

**Figure 5.29:** Overview of all concepts and the related realizable use cases

# 6 Analysis of Extension Options for TOSCA

One of the goals of this thesis is to enable data shipping as well as function shipping with TOSCA. Currently, data are not considered in the specification of TOSCA and cannot be explicitly modeled. Thus, for the integration of data resources in the Service Template the TOSCA language has to be extended. For the development of the TOSCA realization options presented in Chapter 5 three alternative methods are analyzed. These are extension options for TOSCA to enable data shipping and function shipping. In the following these three methods are explained and discussed based on the developed TOSCA realization options.

The following three extension methods for TOSCA are considered:

1. TOSCA metamodel extension by means of new language elements
2. Extension of existing TOSCA metamodel elements
3. Use of existing metamodel elements

The first method bases on an extension on the type level of TOSCA. A new type, e.g., *Data Type* is added to the two existing types, Node Type and Relationship Type. Additionally, a corresponding Template and the relation between the existing Templates and the new one have to be defined. The advantages are that data specific properties can be defined for the new type and on the metamodel level it can already be distinguished between data and application components. The disadvantages are a complex extension and redundancy between the Node Type definition and the Data Type definition, since the not optional element of a Node Type *name* to identify the Node Type are likewise needed to specify Data Types appropriately.

The second method based on an extension of the existing types to integrate data specific elements to the type and template definitions. In contrast to the first method just data specific elements are added to the metamodel of the existing types. But in this case, all added elements required for data are available for every specified type or template. Additional extension elements, which are not declared as optional have to be set also for types that do not represent data. The Application Interface extension introduced by

Zimmermann [Zim16] is an example for this second method. The Node Type element is extended by means of new elements defining application interfaces, which differ from the already existing management interfaces as described in Section 2.1.3.

With the last method the existing metamodel elements and definitions are used. Especially the *PropertiesDefinition* element of Node Types and Relationship Types can be used to define node specific properties, i.e., data specific characteristics. On the one hand the advantages are that the metamodel does not have to be extended and data specific properties still can be added, on the other hand the disadvantage is that the data can not be distinguished from the application components on the metamodel level.

The main characteristics to describe data appropriately regardless to the semantics of the data include the data location and storage format. These are properties not covered by the existing elements defined for Node Types. However, the not optional element of a Node Type is the attribute *name* to identify the Node Type which is likewise needed to uniquely identify a data resource. Furthermore data specific properties can be defined by means of the *PropertiesDefinition* element. For the options presented in Chapter 5 properties are used, such as *Location* to reference the physical data location, *ResourceFormat* to indicate the format in which the data are stored, or *ProcLogic* to reference the required processing logic for a given data resource (Option 2.2). The same applies for the Relationship Type, which specifies the relation between two Nodes. It can be used to specify the link between processing logic and data with properties like *SourceInterface*, *SourceOperation*, and *SourceInputPar*. That shows how versatile the *PropertiesDefinition* element is.

Considering the data as a component of a composite application it is not contradictory to the actual semantic of Node Types if data are specified as Node Types. Due to the pros and cons of the different methods mentioned above the third method is used in the TOSCA realization options, described in Chapter 5. Only for Option 4.2 the second method is used and a new attribute is defined for the Application Interface element. For Concept 4 to Concept 8 the already existing Application Interface extension, introduced by Zimmermann [Zim16] is used.

# 7 Conclusion and Future Work

The goal of this thesis is to develop and evaluate modeling concepts and the corresponding realizations in TOSCA to enable data shipping and function shipping. Thereby, the modeling of data resources and processing logic, extraction and transformation mechanisms as well as the assignment between data and processing logic are the main aspects. In total, eight modeling concepts and 16 TOSCA realization options, which are applicable for data shipping, function shipping, or both are presented. They include two different shipping use cases for data shipping and function shipping: shipping of the actual data or function and shipping of a reference to a remote location. In the first case, the data or the processing logic are packaged together with the model and transferred to the target environment. In the second case, the data or processing logic remain at their location and a reference to the remote location, where they can be retrieved at a later time, is transmitted.

The eight concepts illustrate an abstract modeling approach and the TOSCA realization options present the implementations of the abstract models in TOSCA. Each concept introduces an assignment method and/or a transformation and extraction mechanism. The first two concepts focus on the data. In Concept 1 the assignment between data and processing logic is not explicit modeled, whereas Concept 2 facilitate a mapping between data resource and the required processing logic. The main object of Concept 3 and Concept 4 is the processing logic, which show different possibilities to model the assignment of data resources to a given processing logic. The last four concepts introduce a modeling element for the explicit modeling of the relationship between processing logic and data. That gives a high flexibility in terms of the use for data shipping and function shipping. Additionally, extraction and transformation operations to retrieve just a subset of data or to execute syntactic or semantic transformations are shown in Concept 6 and Concept 8. For each concept it is shown in which context the concept is applicable and which impact the usage has.

For the TOSCA realization options different extension options for TOSCA were analyzed. Apart from the TOSCA Application Interface extension no further extensions of the TOSCA metamodel are required. in almost all options only existing elements, which are already part of the specification are used. The data resource specific attributes and assignment characteristics are realized by the definition of Node Type specific properties.

Thus, with the developed modeling concepts and TOSCA realization options different possibilities for the modeling of the assignment between data and processing logic as well as the integration of extraction and transformation operations in the model are presented. The presented modeling concepts are evaluated in the current research project SePiA.Pro (01MD16013F). In case the modeling concepts turn out to be proven solution, they can serve as basis for modeling patterns for data shipping and function shipping. Deciding on which modeling concept fits best mainly depends on the modeling scenario, the use case, the modeler's knowledge about the processing logic and data, and the capabilities of the used runtime environment.

Further work is required in terms of the development of a runtime environment, which is able to process the presented concepts. With OpenTOSCA a ecosystem is available for modeling application topology models and for the automated provisioning and management of these applications [Bin+13; Bre+16]. It can process CSARs and can deploy and instantiate the contained applications. This runtime environment could be used and extended to deploy and assign data to the respective processing logic and to enable the invocation of application operations with the data. Hahn et al. [Hah+16] introduce an approach to support the data-related aspects in service compositions and choreographies and to decouple the data flow from the control flow based on a new Transparent Data Exchange (TraDE) middleware layer. This could supplement the OpenTOSCA environment to cover the data-related aspects.

Another field of work is the extension of the presented modeling concepts and options by non-functional requirements, such as data quality, availability, or security. TOSCA provides Policy elements to define and attach non-functional requirements to the application components. Which requirements apply especially in terms of the data, how Policy elements can be used, and if further extensions are required, should be examined. Further research in these areas will help facilitating data shipping and function shipping.

# Bibliography


[Ana16a] Logi Analytics. *Introduce Datasource Connections*. http://devnet.logianalytics.com/rdPage.aspx?rdReport=Article&dnDocID=2101. 2016. (Visited on 06/24/2016).

[Ana16b] Logi Analytics. *Introducing Datalayers*. http://devnet.logianalytics.com/rdPage.aspx?rdReport=Article&dnDocID=2040. 2016. (Visited on 06/24/2016).

[Arm+10] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, et al. "A View of Cloud Computing." In: *Communications of the ACM* 53.4 (2010), pp. 50–58.

[Atz+10] L. Atzori, A. Iera, G. Morabito. "The Internet of Things: A Survey." In: *Computer Networks* 54.15 (2010), pp. 2787 –2805.

[AWS15] AWS. *AWS Serverless Multi-Tier Architectures: Using Amazon API Gateway and AWS Lambda*. https://d0.awsstatic.com/whitepapers/AWS_Serverless_Multi-Tier_Architectures.pdf. 2015. (Visited on 07/06/2016).

[AWS16a] AWS. *Amazon Virtual Private Cloud*. https://aws.amazon.com/vpc/?nc1=h_ls. 2016. (Visited on 07/07/2016).

[AWS16b] AWS. *AWS Elastic Beanstalk*. https://aws.amazon.com/elasticbeanstalk/?nc1=h_ls. 2016. (Visited on 07/07/2016).

[AWS16c] AWS. *AWS Import/Export Snowball*. https://aws.amazon.com/importexport/?nc1=h_ls. 2016. (Visited on 06/22/2016).

[Azu16] Microsoft Azure. *Azure Functions*. https://azure.microsoft.com/en-us/services/functions/. 2016. (Visited on 07/06/2016).

[Bal+14] R. Balasubramonian, J. Chang, T. Manning, J.H. Moreno, R. Murrphy, R. Nair, S. Swanson. "Near-Data Processing: Insights From a Micro-46 Workshop." In: *IEEE Micro* 34.4 (2014), pp. 36–42.

[Bar+12] P. Barnaghi, W. Wang, C. Henson, K. Taylor. "Semantics for the Internet of Things: early progress and back to the future." In: *International Journal on Semantic Web and Information Systems (IJSWIS)* 8.1 (2012), pp. 1–21.

[Bat+06] C. Batini, M. Scannapieco. *Data Quality: Concepts, Methodologies and Techniques*. Springer, 2006.

[Bin+13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner. "OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications." In: *11th International Conference on Service-Oriented Computing*. Springer, 2013, pp. 692–695.

[Bin+14] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann. "TOSCA: Portable Automated Deployment and Management of Cloud Applications." In: *Advanced Web Services*. Springer, 2014, pp. 527–549.

[Bir16] Birst. *Birst: User Data Tier*. https://www.birst.com/product/analytics-technology/. 2016. (Visited on 06/22/2016).

[Bol+00] W.J. Bolosky, J.R. Douceur, D. Ely, M. Theimer. "Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs." In: *ACM SIGMETRICS Performance Evaluation Review* 28.1 (2000), pp. 34–43.

[Bre+16] U. Breitenbücher, C. Endres, K. K´epes, O. Kopp, F. Leymann, S. Wagner, J. Wettinger, M. Zimmermann. *The OpenTOSCA Ecosystem – Concepts & Tools*. University of Stuttgart, 2016.

[Bud+15] A. Buda, K. Främling, J. Borgman, M. Madhikermi, S. Mirzaeifar, S. Kubler. "Data Supply Chain in Industrial Internet." In: *2015 IEEE World Conference on Factory Communication Systems (WFCS)*. IEEE, 2015, pp. 1–7.

[Bus+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, 1996.

[Con16] The Industrial Internet Consortium. *What is the Industrial Internet?* http://www.iiconsortium.org/about-industrial-internet.htm. 2016. (Visited on 06/25/2016).

[Cor+86] D.W. Cornell, D.M. Dias, S.Y. Philip. "On Multisystem Coupling Through Function Request Shipping." In: *IEEE Transactions on Software Engineering* SE-12.10 (1986), pp. 1006–1017.

[Ell+99] D.G. Elliott, M. Stumm, W.M. Snelgrove, C. Cojocaru, R. Mckenzie. "Computational RAM: Implementing Processors in Memory." In: *IEEE Design and Test of Computers* 16.1 (1999), pp. 32–41.

[Feh+14] C. Fehling, F. Leymann, R. Retter, W. Schupeck, P. Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[For+13] Forschungsunion, acatech. *Umsetzungsempfehlungen für das Zukunftsprojekt Industrie 4.0: Abschlussbericht des Arbeitskreis Industrie 4.0 [Recommendations for the Realization of Industrie 4.0: Final Report of the Working Group Industrie 4.0]*. https://www.bmbf.de/files/Umsetzungsempfehlungen_Industrie4_0.pdf. 2013. (Visited on 06/24/2016).

[Fow03] M. Fowler. *Patterns für Enterprise Application-Architekturen [Patterns of Enterprise Application Architecture]*. MITP, 2003.

[Fra+96] M.J. Franklin, B.T. Jónsson, D. Kossmann. “Performance Tradeoffs for Client-Server Query Processing.” In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD 96)*. ACM, 1996, pp. 149–160.

[Hah+16] M. Hahn, D. Karastoyanova, F. Leymann. “Data-Aware Service Choreographies through Transparent Data Exchange.” In: *Web Engineering: 16th International Conference, ICWE 2016*. Springer, 2016, pp. 357–364.

[Har+06] G. Harrison, S. Feuerstein. *MySQL Stored Procedure Programming*. O’Reilly, 2006.

[IBM16] IBM. *IBM Bluemix OpenWhisk*. http://www.ibm.com/cloud-computing/bluemix/openwhisk/. 2016. (Visited on 07/06/2016).

[Kem+04] H.-G. Kemper, W. Mehanna, C. Unger. *Business Intelligence: Grundlagen und praktische Anwendungen [Business Intelligence: Fundamentals and practical applications]*. Springer, 2004.

[Lé+08] F. Lécue, S. Salibi, P. Bron, A. Moreau. “Semantic and Syntactic Data Flow in Web Service Composition.” In: *ICWS: 2008 IEEE International Conference on Web Services*. IEEE, 2008, pp. 211–218.

[Lee+14] J. Lee, H.-A. Kao, S. Yang. “Service Innovation and Smart Analytics for Industry 4.0 and Big Data Environment.” In: *Procedia CIRP* 16 (2014), pp. 3–8.

[Lee+02] J.Y.B. Lee, R.W.T. Leung. “Study of a Server-less Architecture for Video-on-Demand Applications.” In: *Proceedings: 2002 IEEE International Conference on Multimedia and Expo (ICME 02)*. IEEE, 2002, pp. 233–236.

[Mel+11] P. Mell, T. Grance. *The NIST Definition of Cloud Computing*. National Institute of Standards and Technology, 2011.

[OAS13a] OASIS. *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0 - Committee Note Draft 01*. http://docs.oasis-open.org/tosca/tosca-primer/v1.0/cnd01/tosca-primer-v1.0-cnd01.html. 2013. (Visited on 06/14/2016).

[OAS13b] OASIS. *Topology and Orchestration Specification for Cloud Applications Version 1.0*. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html. 2013. (Visited on 06/09/2016).

[Pat+97] D. Patterson, T. Anderson, N. Cardwell, R. Fromm, K. Keeton, C. Kozyrakis, R. Thomas, K. Yelick. "A Case for Intelligent Ram." In: *IEEE Micro* 17.2 (1997), pp. 34–43.

[Pet+14] D. Petcu, A.V. Vasilakos. "Portability in Clouds: Approaches and Research Opportunities." In: *Scalable Computing: Practice and Experience* 15.3 (2014), pp. 251–271.

[Pip+02] L.L. Pipino, Y.W. Lee, R.Y. Wang. "Data Quality Assessment." In: *Communications of the ACM* 45.4 (2002), pp. 211–218.

[Pla16] Google Cloud Platform. *Cloud Functions*. https://cloud.google.com/functions/. 2016. (Visited on 07/06/2016).

[Rei+11] P. Reimann, M. Reiter, H. Schwarz, D. Karastoyanova, F. Leymann. "SIMPL – A Framework for Accessing External Data in Simulation Workflows." In: *Datenbanksysteme für Business, Technologie und Web (BTW 2011): 14. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS)*. Gesellschaft für Informatik (GI), 2011, pp. 534–553.

[Rei+14] P. Reimann, T. Waizenegger, M. Wieland, H. Schwarz. "Datenmanagement in der Cloud für den Bereich Simulationen und Wissenschaftliches Rechnen [Data Management in the Cloud for the Domain Simulations and Scientific Computing]." In: *Proceedings des 2. Workshop Data Management in the Cloud auf der 44. Jahrestagung der Gesellschaft für Informatik*. Gesellschaft für Informatik (GI), 2014, pp. 735–746.

[Rie+98] E. Riedel, G. Gibson, C. Faloutsos. "Active Storage for Large-Scale Data Mining and Multimedia Applications." In: *Proceedings of the 24th Conference on Very Large Databases*. Citeseer, 1998, pp. 62–73.

[Rob16] M. Roberts. *Serverless Architectures*. http://martinfowler.com/articles/serverless.html. 2016. (Visited on 07/06/2016).

[RM+00] M. Rodríguez-Martínez, N. Roussopoulos. "MOCHA: a Self-Extensible Database Middleware System for Distributed Data Sources." In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD 00)*. ACM, 2000, pp. 213–224.

[Sad+04] S. Sadiq, M. Orlowska, W. Sadiq, C. Foulger. "Data Flow and Validation in Workflow Modelling." In: *Proceedings of the 15th Australasian Database Conference-Volume 27*. Australian Computer Society, Inc., 2004, pp. 207–214.

[Sel+98] J. Sellentin, B. Mitschang. “Data-Intensive Intra- and Internet Applications-Experiences Using Java and CORBA in the World Wide Web.” In: *Proceedings of the 14th International Conference on Data Engineering*. IEEE, 1998, pp. 302–311.

[Tiw+13] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, Y. Solihin. “Active Flash: Towards Energy-Efficient, In-Situ Data Analytics on Extreme-Scale Machines.” In: *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, 2013, pp. 119–132.

[Vor+04] K. Voruganti, M.T. Özsu, R.C. Unrau. “An Adaptive Data-Shipping Architecture for Client Caching Data Management Systems.” In: *Distributed and Parallel Databases* 15.2 (2004), pp. 137–177.

[W3C12] W3C. *W3C XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes*. https://www.w3.org/TR/xmlschema11-2/. 2012. (Visited on 06/20/2016).

[Zim16] M. Zimmermann. “Konzept und Implementierung einer Komponente zur Kommunikation TOSCA-basierter Anwendungen [Concept and Impplementation of a Component to Enable Communication Between TOSCA-based Applications].” MA thesis. University of Stuttgart, 2016.

**Erklärung**

Hiermit erkläre ich, dass ich die Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen der Arbeit, die wörtlich oder sinngemäß aus Veröffentlichungen oder aus anderen fremden Äußerungen entnommen wurden, sind als solche einzeln kenntlich gemacht.
Die Masterarbeit habe ich noch nicht in einem anderen Studiengang als Prüfungsleistung verwendet.
Des Weiteren erkläre ich, dass mir weder an den Universitäten Hohenheim und Stuttgart noch an einer anderen wissenschaftlichen Hochschule bereits ein Thema zur Bearbeitung als Masterarbeit oder als vergleichbare Arbeit in einem gleichwertigen Studiengang vergeben worden ist.

Stuttgart-Hohenheim, den 29. Juli 2016

---

Unterschrift