Institute of Architecture of Application Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master's Thesis Nr. 3341

Merging of TOSCA Cloud Topology Templates

Andreas Weiß

Course of studies:	Wirtschaftsinformatik
First Examiner:	Prof. Dr. Frank Leymann
Second Examiner:	Jun.-Prof. Dr.-Ing. Dimka Karastoyanova
Supervisor:	Dipl.-Inf. Tobias Binz
Commenced:	April 17, 2012
Completed:	October 17, 2012
CR-Classification:	C.2.4, D.2.11, H.3.4, G.2.2

Abstract

The paradigm of Cloud Computing requires standardization to avoid vendor lock-in for its users. The Topology and Orchestration Specification for Cloud Applications (TOSCA) provides a standardization approach enabling portability of cloud services between different Cloud Computing providers. The main goal of the TOSCA specification is to enable a cloud provider and environment independent description of these services concerning structure and management aspects during their life cycle. TOSCA specifies so-called Service Templates whose structure is described by a Topology Template.

The TOSCA specification provides means to model enterprise applications in a standardized way. In view of mergers and acquisitions, data center consolidation, for breaking up silo structures in IT departments and for support of modeling experts it is necessary to find concepts for analyzing already modeled TOSCA Topology Templates for similar elements and for unifying these elements and, thus, two Topology Templates.

This master's thesis develops a matching concept for finding similar elements inside and between two Topology Templates by systematically exploring all different constellations TOSCA elements can take. Similar elements are indicated by the notion of a Correspondence. The matching concept is automated by developing algorithms that determine Correspondences and incorporate domain-specific knowledge via type-specific plugins. The plugins handle the matching of properties that cannot be conducted generically. Furthermore, a merging concept and appropriate algorithms are developed that utilize the identified Correspondences for unifying similar elements. All algorithms are designed with the goal of practical computational complexity.

A further part of this work is the design and prototypical implementation of the extendable TOSCAMerge framework that allows for a convenient integration of type-specific plugins. The framework facilitates the assessment and manipulation of the determined Correspondences by domain experts prior to merging. A set of example TOSCA Service Templates for testing the different matching and merging cases complements the implementation.

An extensive evaluation of the concepts and algorithms reveals the algorithms' greedy properties including local optimality but quadratic computational complexity in most cases.

Contents

1 Introduction ... 1
1.1 Problem Statement ... 1
1.2 Motivating Scenario ... 2
1.3 Research Design ... 3
1.4 Outline ... 3
2 Fundamentals ... 5
2.1 Graph Theory ... 5
2.2 Cloud Computing ... 6
2.3 Topology and Orchestration Specification for Cloud Applications ... 8
2.3.1 TOSCA Syntax ... 8
2.3.2 Use Cases of TOSCA ... 15
2.4 Frameworks ... 16
2.4.1 Definitions ... 16
2.4.2 Template Method ... 17
2.4.3 Factory Method ... 18
3 Related Work ... 22
4 Assumptions and Requirements for Matching and Merging ... 26
4.1 Assumptions ... 26
4.2 Requirements ... 30
5 Concept for Matching of Topology Templates ... 32
5.1 Matching of Node Templates ... 32
5.1.1 Analysis of the Basic Case and its Derivations ... 33
5.1.2 Matching of Node Template Properties ... 42
5.2 Matching of Relationship Templates ... 49
5.2.1 Analysis of the Different Relationship Template Matching cases ... 50
5.2.2 Matching of Relationship Template Properties ... 57
5.3 Matching in the Context of Group Templates ... 59
5.3.1 Extension of the Basic Matching Algorithm for Node Templates ... 61
5.3.2 Different Cases of Recursive Access to Group Templates ... 64
5.3.3 Matching of Node Templates on Different Nesting Levels ... 76
5.3.4 Matching of Group Template Policies ... 82
5.3.5 Relationship Templates in the Context of Group Templates ... 85
6 Concept for Merging of Topology Templates ... 88
6.1 Merging of Node Templates ... 88
6.1.1 Analysis of the Basic Case and its Derivations ... 88
6.1.2 Merging of Properties on the Node Template Level ... 99

6.2 Merging of Relationship Templates 103
6.2.1 Basic Merging Algorithm for Relationship Templates 104
6.2.2 Merging of Relationship Template Properties 106
6.3 Merging in the Context of Group Templates 107
6.3.1 Merging of Node Templates 107
6.3.2 Merging of Relationship Templates 111
6.3.3 Merging of Group Templates 112
7 Architecture & Design of an Extendable Framework 116
7.1 High-level Architecture 116
7.2 Refined Design 119
7.3 Extensibility of the TOSCAMerge Framework 126
8 Implementation of the Framework 131
8.1 General Remarks 131
8.2 Implementation of the TOSCA Data Structures 131
8.3 Implementation of the Extensibility Concept 135
8.3.1 Extensibility Concept 135
8.3.2 Adding New Plugins to the TOSCAMerge Framework 138
9 Evaluation of the Algorithms and the Implemented Concepts 139
9.1 Evaluation of the Findings Regarding the General Requirements 139
9.2 Discussion of the Proposed Algorithms 140
9.2.1 General Properties of the Proposed Algorithms 140
9.2.2 Complexity Considerations of the Matching Algorithm 141
9.2.3 Complexity Considerations of the Merging Algorithm 143
9.2.4 Overall Complexity 145
9.3 Creation of Sample TOSCA Service Templates 146
9.4 Discussion of Merging of More Than two Topology Templates 147
10 Conclusion and Future Work 148
Bibliography 150

List of Figures

Fig. 1.1: Example of two Enterprise Topologies to be merged....2
Fig. 2.1: Graphical instance example of a TOSCA Service Template....14
Fig. 2.2: Template Method design pattern....17
Fig. 2.3: Factory Method design pattern....19
Fig. 2.4: Combined design patterns....21
Fig. 4.1: Valid example of 1:1 correspondences between Node Templates....27
Fig. 4.2: Invalid example of a n:1 correspondence between Node Templates....27
Fig. 4.3: Relevant Node Type tree....28
Fig. 4.4: Relevant Relationship Type tree....29
Fig. 5.1: Example of a general matching case....33
Fig. 5.2: Motivation for matching inside a Topology Template....34
Fig. 5.3: Example of an Application Node Template....35
Fig. 5.4: Correspondences to more than one Node Template....36
Fig. 5.5: Node Type parent relationships....37
Fig. 5.6: Node Type sibling relationship....38
Fig. 5.7: Possible, but invalid correspondence leads to undesired semantics....51
Fig. 5.8: Four matching cases of Relationship Templates with Communication semantics....52
Fig. 5.9: Four matching cases of Relationship Templates with Dependency semantics....53
Fig. 5.10: Example of invalid merged Topology Template....54
Fig. 5.11: Matching in the context of Group Templates....60
Fig. 5.12: Example of matching inside a Group Template....64
Fig. 5.13: Example of Group Template Correspondence on Nesting Level 0....65
Fig. 5.14: Example of correspondences on a deeper Nesting Level....66
Fig. 5.15: Example of Group Template access case 3a....67
Fig. 5.16: Example of Group Template Access case 3b....68
Fig. 5.17: Example of Group Template access case 4....69
Fig. 5.18: Example of Group Template access case 5....70
Fig. 5.19: Different Nesting Level case 1....77
Fig. 5.20: Different Nesting Level case 2....78
Fig. 5.21: Different Nesting Level case 3....79
Fig. 5.22: Different Nesting Level case 4....80
Fig. 6.1: Basic merging case....88
Fig. 6.2: Full mapping between all Node Templates....94
Fig. 6.3: Partial mapping between Node Templates....94
Fig. 6.4: Full mapping but inverted directions....95
Fig. 6.5: Example of "pulling" a Node Template into a Group Template....110
Fig. 6.6: Function relocateEdges....112
Fig. 7.1: Overall architecture with matching part....117
Fig. 7.2: Overall architecture with merging part....118
Fig. 7.3: Class diagram of the TOSCAMerge framework service access....119
Fig. 7.4: Detailed design of the matching part....121
Fig. 7.5: Detailed design of the merging part....122
Fig. 7.6: Class diagram of the Correspondences....124
Fig. 7.7: Class diagram of the Group Template Hierarchy concept....125
Fig. 7.8: Class diagrams of the combined design patterns....127
Fig. 8.1: Section of the Java class of a Node Template....134
Fig. 8.2: Class BaseClass....135
Fig. 8.3: Method createNodeTemplateMatcher of class TOSCAMatchingFactory....136

Fig. 8.4: Plugin-class for matching Node Templates with identical Node Types.....................137
Fig. 9.1: The two merged Topology Template of the motivating scenario................................146

List of Listings

Listing 2.1: High level syntax of TOSCA Service Template....9
Listing 2.2: Syntax of XML complexType TExtensibleElements....10
Listing 2.3: Code example of a framework class using the Template Method pattern....18
Listing 2.4: Code example of a specific class using the Template Method pattern....18
Listing 2.5: Code example of abstract factory class of the Factory Method design pattern....19
Listing 2.6: Concrete factory class of the Factory Method design pattern.....20
Listing 2.7: Initialization of a factory....20
Listing 2.8: Usage of the factory....20
Listing 5.1: Function findMapping part 1....40
Listing 5.2: Function findMapping part 2....41
Listing 5.3: Function isDerivedFrom....42
Listing 5.4: PropertyDefaults example of a generic application server....43
Listing 5.5: PropertyConstraints example of a generic application server....43
Listing 5.6: Function match for Node Templates....46
Listing 5.7: Function matchPolicies....47
Listing 5.8: Function determineDerivedPolicies....49
Listing 5.9: Function handleRelationshipTemplates for HostedOn semantics....56
Listing 5.10: Function match for Relationship Templates....58
Listing 5.11: Function matchRelationshipConstraints....59
Listing 5.12: Data structure GroupTemplateHierarchy as double-linked list....61
Listing 5.13: Extended function findMapping part 1....62
Listing 5.14: Extended function findMapping part 2....63
Listing 5.15: Function isGroupTemplateAccessPossible part 1....71
Listing 5.16: Function isGroupTemplateAccessPossible part 2....72
Listing 5.17: Function checkGroupTemplateCorrespondence....73
Listing 5.18: Function getHierarchyElement....75
Listing 5.19: Function checkCorrespondencesOfLevel....76
Listing 5.20: Function isMatchingPossible....81
Listing 5.21: Function matchPolicies for Group and Node Templates....83
Listing 5.22: Function determineHierarchyPolicies for Group Templates....84
Listing 5.23: Function getAllPolicies....85
Listing 5.24: Function buildRelationshipTemplateSet....86
Listing 6.1: Function performNodeTemplateMerge....90
Listing 6.2: Function reconnectEdges....92
Listing 6.3: Function reconnectCorrespondences for Node Templates....93
Listing 6.4: Function hasCorrectNumberOfCorrespondences part 1....97
Listing 6.5: Function hasCorrectNumberOfCorrespondences part 2....99
Listing 6.6: Function merge for Node Templates....100
Listing 6.7: Function decideNodeType....101
Listing 6.8: Example for type-specific PropertyDefaults merging....102
Listing 6.9: Function mergePropertyConstraints....103
Listing 6.10: Function handleRelationshipTemplates....105
Listing 6.11: Function reconnectCorrespondences for Relationship Templates....106
Listing 6.12: Function merge for Relationship Templates....107
Listing 6.13: Extension of the function performNodeTemplateMerge....109
Listing 6.14: Function performGroupTemplateMerge part 1....113
Listing 6.15: Function performGroupTemplateMerge part 2....114
Listing 6.16: Function merge for Group Templates....115
Listing 7.1: Schema of Node-Type-specific configuration part 1....128

Listing 7.2: Schema of Node-Type-specific configuration part 2......129
Listing 7.3: Example of a Node-Type-specific configuration file......130
Listing 8.1: Example of an anonymous type in the TOSCA XML-schema......132
Listing 8.2: Section of the used JAXB Customization Bindings......133

List of Tables

Table 8.1: Necessary elements to create a new plugin..138

List of Abbreviations

AGG	Attributed Graph Grammar System
API	Application Programming Interface
BPEL	Business Process Execution Language
BPMN	Business Process Model and Notation
CAPEX	Capital expenditure
CRM	Customer Relationship Management
DOM	Document Object Model
EAR	Enterprise Archive
EJB	Enterprise Java Bean
EPC	Event-driven Process Chain
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IaaS	Infrastructure as a service
I/O	input/output
IT	Information Technology
Java EE	Java Platform, Enterprise Edition
Java SE	Java Platform, Standard Edition
JAXB	Java Architecture for XML Binding
JVM	Java Virtual Machine
NIST	National Institute of Standards and Technology
OPEX	Operational expenditure
OS	Operating System
PaaS	Platform as a service
REST	Representational state transfer
ROI	Return on Investment

SaaS	Software as a service
SAX	Simple API for XML
SESE	Single-Entry-Single-Exit
SME	Small and medium enterprise
TOSCA	Topology and Orchestration Specification for Cloud Applications
URI	Universal Resource Identifier
WAS	Websphere Application Server
WSDL	Web Services Description Language
XML	Extensible Markup Language
XSD	XML Schema Document

1 Introduction

Cloud Computing is a new paradigm discussed in research, the IT industry and beyond [5], [20]. It brings the no longer recent goal [36] of computing resources being available as utilities comparable to gas, water or electricity closer to reality. It has the potential of "creative destruction" that destroys an old economic structure and creates a new one [44]. The potential lies in the use of computing resources as pay-per-use services that scale on demand and enable organizations to invest more into their core competences than into building and maintaining IT systems.

However, without the standardization of Cloud Computing there is the danger of vendor lock-in for its users. Once a particular Cloud Computing provider relying on proprietary approaches is chosen it may be difficult to obtain computing services from a different provider [5], [30]. The Topology and Orchestration Specification for Cloud Applications (TOSCA) provides a standardization approach enabling portability of cloud services between different Cloud Computing providers [32]. TOSCA is an XML-based language and metamodel whose grammar provides the possibility to describe IT services. The main goal of the TOSCA specification is to enable a cloud provider and environment independent description of these services concerning structure and management aspects during their life cycle. TOSCA specifies so-called Service Templates whose structure is described by a Topology Template. Plans located in a Service Template provide the possibility to manage the service instances during run-time. They contribute to the (semi-) automatic creation and management of IT services as suggested by the cloud computing paradigm [6]. Ultimately, the definition of the topology and orchestration plans as interoperable artifacts are supposed to make IT services exchangeable between different cloud providers.

1.1 Problem Statement

The TOSCA specification provides means to model enterprise applications in a standardized way. In view of mergers and acquisitions, data center consolidation, for breaking up silo structures in IT departments and for support of modeling experts it is necessary to find concepts for analyzing already modeled TOSCA Topology Templates for similar elements and for unifying these elements and, thus, two Topology Templates.

The goals of this master's thesis are the following: (1) the development of a matching concept for finding corresponding elements inside and between two Topology Templates by systematically exploring all different constellations TOSCA elements can take. Additionally appropriate algorithms implementing the concept have to be formulated. (2) the matching concept and algorithms are the prerequisite for developing a concept and corresponding algorithms that merge the TOSCA elements that have been identified as compatible. (3) All the findings of goal (1) and (2) are incorporated into an extendable framework named *TOSCAMerge* framework that is implemented prototypically using the programming language Java. (4) the last goal is the creation of a set of example TOSCA Service Templates to evaluate the prototype framework and the researched concepts.

Not in the scope of this master's thesis is the research how the management plans of two TOSCA Service Templates have to be adjusted after the TOSCA elements of two Topology Templates have been merged.

1.2 Motivating Scenario

To illustrate the motivation for the merging of two Topology Templates the example of Fig. 1.1 is given. Two enterprises of equal size are merged in order to achieve synergies in their business operations. Their enterprise applications are modeled with simplified Topology Templates; the quadrangles with rounded corners represent the IT components, the arrows the relationships and connections between them. Several overlapping elements such as the *Tomcat Application Servers* [4] and *the MySQL Databases* [35] can be seen in the example. Furthermore, the Operating Systems in both Topology Templates are similar but not identical. Nevertheless, they are also redundant. In order to contribute to the synergetic effects the IT organizations of both enterprises must merge their enterprise application topologies.

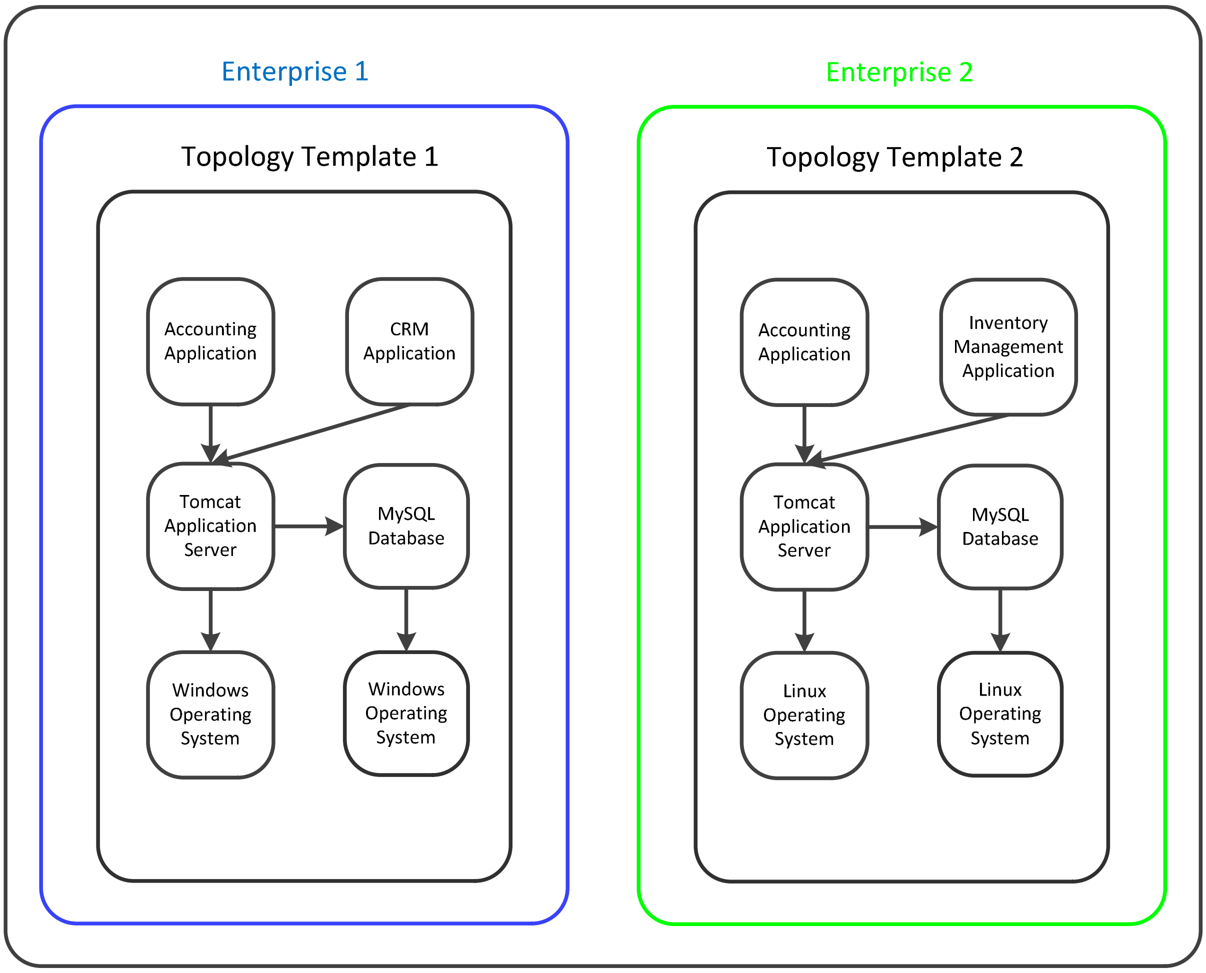


Fig. 1.1: Example of two Enterprise Topologies to be merged

However, the manual identification of similar elements in both Topology Templates as well as the manual unification is a very exhausting task especially when the number of components is much larger than in this motivating scenario. Therefore, the development of concepts of to find the overlapping elements inside and between two Topology Templates and merge them in a subsequent step aims to provide a tool that can automate this task. The example in Fig. 1.1 is picked up again in Chapter 9 when evaluating the results of this master's thesis.

1.3 Research Design

The master's thesis at hand has the goal to develop concepts and algorithms for finding compatible elements inside and between two Topology Templates and merging them in a subsequent step. Therefore, the state of the art of the work in related areas such as graph, process and schema matching and merging is reviewed and analyzed. Appropriate approaches will be adapted for the matching and merging of Topology Templates. Subsequently, a set of requirements for the concepts, algorithms and the TOSCAMerge framework are derived that have to be followed in the next research step of systematically exploring the different matching and merging cases and developing concepts and algorithms to cope with these cases. The researched concepts and algorithms are implemented in a framework that allows the adding of domain-specific knowledge if certain steps cannot be conducted generically. A set of newly created TOSCA Service Templates will be used to evaluate the concepts, algorithms and the framework.

1.4 Outline

This master's thesis is structured in the following way:

Chapter 1 - Introduction gives an introduction to the topic of merging of Topology Templates by stating the research problem, delineating the scope of this work, giving a motivating scenario and stating the research design.

Chapter 2 - Fundamentals explains all the necessary fundamentals this work is based on. This includes Graph Theory, an introduction to Cloud Computing, the TOSCA specification and syntax as well as how to design extendable frameworks.

Chapter 3 - Related Work discusses the work in related research fields such as graph, process and schema matching and merging and evaluates the usefulness and adaptability for this thesis.

Chapter 4 - Assumptions and Requirements for Matching and Merging states the assumptions made by the author regarding the matching and merging of Topology Templates and identifies requirements on the concepts and algorithms for matching and merging.

Chapter 5 - Concept for Matching of Topology Templates covers the finding of similar elements in two Topology Templates and identifies and discusses all different matching constellations that can be found inside and between the elements of Topology Templates. Furthermore, algorithms implementing the concept are designed.

Chapter 6 - Concept for Merging of Topology Templates is based on the concept of chapter 5 and proposes and explores how to unify the elements of two Topology Templates while adhering to the requirements identified in chapter 4. The findings are then incorporated in appropriate merging algorithms.

Chapter 7 - Architecture & Design of an Extendable Framework presents the high-level architecture of the TOSCAMerge framework and some selected detailed components. The chapter is completed by discussing the extendibility approach of the framework.

Chapter 8 - Implementation names the libraries used to implement the TOSCAMerge framework and discusses the challenges the author faced during implementation. Furthermore, the extensibility concept is shown in detail using code examples. Furthermore, it is explained how to add new plugins.

Chapter 9 - Evaluation of the Algorithms and the Implemented Concepts evaluates the proposed matching and merging concepts as well as the introduced algorithms with regard to the identified goals and requirements of this thesis.

Chapter 10 - Conclusion and Future Work summarizes the findings of this master's thesis and suggests related topics for future research.

2 Fundamentals

This chapter provides the fundamentals necessary for this thesis. In Section 2.1 the definitions from Graph Theory used throughout this work are provided. Section 2.2 gives a brief introduction to Cloud Computing and Section 2.3 presents the syntax of the TOSCA specification in detail. Finally, Section 2.4 defines the fundamentals and the approach for designing extendable frameworks.

2.1 Graph Theory

One of the underlying theoretical principles of this thesis is graph theory. This section introduces the terms and definitions used throughout the rest of the thesis.

Informally a graph or general graph is a set of nodes, also called vertices, and a set of edges. There exist many slightly different formal definitions and notations of graphs. The formal definitions in this master thesis are based on [47], [8] and [48].

Definition 2.1 (Undirected graph): *An undirected graph* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *consists of a set* V *of vertices and a set* E *of edges and every* $\mathrm{e} \in \mathrm{E}$ *of* G *connects two, not necessarily distinct, vertices of* V*. For every graph G, the following holds true:* $\mathrm{V} \cap \mathrm{E} = \emptyset$*. A graph* G *is called simple if it has no edge ending at the same vertex (loop) or parallel edges.*

Note: It is assumed that the sets V and E of a graph G are finite.

Definition 2.2 (Incidence and adjacency): *Given a graph* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *and an edge* $\mathrm{e} \in \mathrm{E}$*, one can write* $\mathrm{e} = \{\mathrm{u}, \mathrm{v}\}$ *whereas* $\mathrm{u}, \mathrm{v} \in \mathrm{V}$ *are vertices that are called the ends of e. If* v *is an end of* e*, we say* v *is incident to* e*. Two vertices that are the ends of an edge* e*, are called adjacent to each other. The same holds true for two edges that are incident with a common vertex.*

Definition 2.3 (Subgraph): *Given a graph* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$*, a graph* $\mathrm{H} = (\mathrm{W}, \mathrm{F})$ *is a subgraph of* G*, if* $\mathrm{W} \subseteq \mathrm{V}$ *and* $\mathrm{F} \subseteq \mathrm{E}$*. We say* G *contains* H *or* H *is contained in* G *and write* $\mathrm{G} \supseteq \mathrm{H}$ *or* $\mathrm{H} \subseteq \mathrm{G}$ *respectively.*

Definition 2.4 (Path and cycle): *Given a graph* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *a path is a linear sequence of vertices that are adjacent if they are consecutive in the sequence and nonadjacent otherwise. No vertex is repeated in the sequence. A cycle is a closed sequence where* $\mathrm{u}_1 = \mathrm{u}_\mathrm{n}$ *and* $\mathrm{u}_1, \mathrm{u}_\mathrm{n} \in \mathrm{V}$*, i.e. the starting and ending vertex are identical.*

Definition 2.5 (Connected graph and components): *A graph* $\mathrm{G} = (\mathrm{V}, \mathrm{E})$ *is connected if there exists a path from every* $\mathrm{u} \in \mathrm{V}$ *to every* $\mathrm{v} \in \mathrm{V}$*. Otherwise* G *is called disconnected. A maximum connected subgraph is called a component.*

Definition 2.6 (Directed graph): *A directed graph, or short digraph,* $G = (V, E)$ *consists of a set* V *of vertices and a set* E *of directed edges (arcs) such that to every* $e \in E$ *a unique ordered pair* $(u, v) \in V$ *has been assigned. The vertex u is called tail and the vertex v is called head of the arc* $e = (u, v)$*. A loop is an arc* $e = (v, v)$ *where head and tail are the same vertices. Two* *arcs $e = (u, v)$ and $f = (u, v)$ are parallel if their head and tail are the identical in each case. A simple directed graph is a graph that has neither loops nor parallel arcs. For every simple directed graph $G = (V, E)$ the following always holds true: $E \subseteq V \times V$. The definition 2.4 of subgraphs can also be applied to directed graphs.*

Note: A graph can also have assigned types and label to its vertices and edges [25].

Definition 2.7 (Trees): *A connected graph $G = (V, E)$ is called a tree if it does not contain any circles. In a tree there exists only one path between two vertices $u, v \in V$. One vertex is called root. If the graph G is directed, there exists only one path from the root to every vertex in the tree. Vertices with only one incident edge are called leafs.*

2.2 Cloud Computing

According to Mell and Grance from the National Institute of Standards and Technology (NIST) [27] Cloud Computing is "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction." The authors define five essential characteristics, three service models and four deployment models. These properties also coincide with the definitions and explanations of [43] that are also used in this section.

Characteristics of Cloud Computing

The five characteristics are on-demand-self-service, broad-network-access, resource-pooling, rapid-elasticity and measured service. *On-demand-self-service* connotes that a consumer in need of a cloud service, e.g. processing power or storage, does not require any human interaction by the service provider to provision it. Service capabilities are accessed over network using client agnostic standard mechanisms (*broad-network-access*). *Resource-pooling* refers to cloud providers pooling their resources to serve multiple customers at the same time. This is also called multi-tenancy. Resources, physical and virtual, are assigned respectively removed dynamically to that pool to satisfy the respective demand. Consumers usually do not know the exact location of the resource they obtain, but some service providers offer the possibility to specify some high level location parameters, e.g. Amazon with its concept of availability zones that are distributed around the world [1]. The ability to provision and release resources automatically and fast to scale according to demand is called *rapid elasticity*. This creates the illusion of unlimited resources that can be acquired at any time for the consumer. The last of the five characteristics is the *measured service*. Automatic control and metering helps optimizing the resource allocation. The resources are monitored and reported to enable a pay-per-use billing model.

Service Models of Cloud Computing

Typically there exist three types of Cloud Computing offerings at different layers: Infrastructure, platform and applications. *Infrastructure as a Service* (IaaS), also called Resource Cloud, provides enhanced virtualization capabilities. The consumer has access to computation power, networks and storage and can install and use arbitrary software raging from operating systems to business applications. The underlying cloud infrastructure, however, is not controlled or managed by the consumer.

Platform as a Service (PaaS) resides conceptually one layer above IaaS often using the underlying capabilities of the latter. The PaaS cloud provider offers a platform including programming languages, Application Programming Interfaces (API), libraries, services and tools to the consumer to run self-created or purchased software. The consumer does not manage the underlying infrastructure such as operating systems and servers but can use the APIs to specify the behavior of the platform. The programmatic use of the provider's APIs often binds the created software to the specific cloud provider and impedes migration to other providers.

Software as a Service (SaaS) provides consumers with complete applications accessible through a web browser or through a client program interface. The applications are running on a cloud infrastructure that is beyond the consumer's control. Only some limited application parameters may be configured.

Combinations of the three service models are also common, e.g. the Google App Engine [17] as a PaaS offering in combination with a SaaS application such as Google Docs [18]. Furthermore Salesforce.com has augmented its SaaS offering with the PaaS platform force.com where consumers can write extensions to the existing application [41].

Deployment Models

Four Deployment Models of Cloud Computing are commonly differentiated in literature: Private cloud, Community cloud, Public cloud and Hybrid cloud. A *Private cloud* is exclusively available for a single organization. The location of the cloud can be on-premise in a datacenter of the organization or off-premise hosted by a third party. The same holds true for ownership, management and operation. Multiple consumers, e.g. several small and medium enterprises (SMEs), with shared interests, such as their mission and security requirements use *Community clouds*. Hosting, operation and management is done by one of the community organizations, a third party or a combination of both. A *Public cloud* is hosted on the premises of a cloud provider and offered to the general public. A *Hybrid cloud* is the combination of two or more of the aforementioned cloud deployment models. The distinct cloud infrastructures are combined to enable particular benefits, e.g. cloud bursting to transfer additional computation load that cannot be handled on-premise from a Private cloud to a Public cloud.

Benefits and Risks

To finish the Cloud Computing section some of the benefits and risks will be illustrated. According to Schubert [43] some of the benefits of Cloud Computing are cost reduction as infrastructure purchase and operational costs can possibly be reduced. Moving to the cloud, private or public, is an investment that has to be calculated beforehand and a positive Return on Investment (ROI) is not always certain. Modifications on applications or the own data center may be necessary. However, in extreme cases capital expenditures (CAPEX) can be completely turned into operational expenditures (OPEX) if a pay-per-use model in combination with a public cloud is used. Another important benefit is improved time to market especially for SMEs without the delay for building up an on-premise infrastructure.

There also exist some risks when using cloud computing, especially in the Public cloud deployment model. Armbrust et al. [5] name privacy concerns regarding sensitive data, vendor-lock-in that prevents migration to other providers, poor predictability of performance, reputation and liability issues.

2.3 Topology and Orchestration Specification for Cloud Applications

The Topology and Orchestration Specification for Cloud Applications (TOSCA) is an XML-based language and metamodel whose grammar provides the possibility to describe IT services. The main goal of the TOSCA specification is to facilitate a description of these services concerning structure and management aspects during their life cycle which is independent from cloud providers or a certain environment. The structure of a so-called Service Template is described by a Topology Template, whereas means to manage the service instances during run-time are provided by Plans. Plans contribute to the (semi-) automatic creation and management of IT services as suggested by the cloud computing paradigm [6]. A Topology is defined as the "the individual components of a service and their relations" [32]. Ultimately, the definition of the topology and orchestration plans as interoperable artifacts are supposed to make IT services exchangeable between different cloud providers.

The following section describes the most important elements of the TOSCA specification [32], [7] using *Version 1.0 Working Draft 05*.

Listing 2.1 shows the overall high-level syntax of a Service Template consisting of a Topology Template or a TopologyTemplateReference, Node Types, Relationship Types and Plans. The *?* denotes an optional element or attribute, the | an exclusive decision (xor), the * zero or many and the + one or many elements or attributes. Words written in italics denote XML elements and attributes.

2.3.1 TOSCA Syntax

Service Template

The *ServiceTemplate* element is the root element of a TOSCA XML document. It has a set of properties; the most important ones will be discussed next. Every Service Template has a unique *id* regarding its namespace and a descriptive, human readable *name*. The *targetNamespace* attribute declares the namespace of the Service Template, an important feature as a Service Template might be referenced in another Service Template.

The optional *Extensions* element offers an extension mechanism to define additional vendor-specific and domain-specific information. The attribute *namespace* specifies the namespaces of extension attributes and elements that are used within the Service Template under definition, the attribute *mustUnderstand* indicates if a TOSCA compliant implementation must adhere to the extension and reject it in case of not understanding. Additionally every XML element specified by the TOSCA specification extends the XML complexType denoted by tExtensibleElements. The complexType is depicted in Listing 2.2. It contains a xs:any element that permits the adding of further XML elements that are not defined by the TOSCA schema. The attribute *processContents*="lax" indicates that a XML processor can validate that element if it is able to obtain any schema information but will not generate error messages if not [50], [49].

```
High level syntax of TOSCA Service Template
```

```
<ServiceTemplate id="ID" name="string"? targetNamespace="anyURI">

  <Extensions>?
    <Extension namespace="anyURI" mustUnderstand="yes|no"?/>+
  </Extensions>

  <Import namespace="anyURI"? location="anyURI"?
    importType="anyURI"/>*

  <Types>?
    <xs:schema .../>*
  </Types>

  (
  <TopologyTemplate>
      ...
  </TopologyTemplate>
  |
  <TopologyTemplateReference reference="xs:QName">
  )?

  <NodeTypes>?
    ...
  </NodeTypes>

  <RelationshipTypes>?
    ...
  </RelationshipTypes>

  <Plans>?
    ...
  </Plans>
</ServiceTemplate>
```

Listing 2.1: High level syntax of TOSCA Service Template

Syntax of the XML complexType tExtensibleElements

```
<xs:complexType name="tExtensibleElements">
  <xs:sequence>
    <xs:element ref="documentation" minOccurs="0"
      maxOccurs="unbounded"/>
    <xs:any namespace="##other" processContents="lax" minOccurs="0"
      maxOccurs="unbounded"/>
  </xs:sequence>
  <xs:anyAttribute namespace="##other" processContents="lax"/>
</xs:complexType>
```

Listing 2.2: Syntax of the XML complexType TExtensibleElements

The optional *Import* element provides means to use external Service Templates, XML Schema or WSDL definitions. A Service Template must name all external references that it uses via *Import* elements.

With the optional *Types* element additional XML definitions can be specified and used throughout the Service Template document, e.g. as attribute in other elements, without the need to define them in separate documents and import them via *Import* elements. The types are XML Schema elements by default but could also be of any type system.

A *TopologyTemplate* defines the topological structure of an IT service as a directed graph. The vertices are represented by a set of *NodeTemplate* elements and the directed edges by a set of *RelationshipTemplate* elements. The edges express the semantics of the relationships between the vertices. The TOSCA specification either demands one *TopologyTemplate* or a *TopologyTemplateReference* which references a *TopologyTemplate* imported via an *Import* element. A Service Template may only have zero or one *TopologyTemplate* or *TopologyTemplateReference*. The explanation of the different properties of a *TopologyTemplate* is continued below.

The optional element *NodeTypes* contains one or many *NodeType* elements which describe the type of *NodeTemplates*, i.e. their properties and behavior. In contrast, the optional element *RelationshipTypes* contains the types of *RelationshipTemplates* and their properties.

The element *Plans* contains *Plan* elements specifying how to manage the IT Service under definition, e.g. how to instantiate und terminate the service.

The TOSCA specification allows that a Service Template serves as a document that only describes an IT Service and is composed into another Service Template that can be instantiated into a service instance. A Service Template document that is to be instantiated must contain either a *TopologyTemplate* or *TopologyTemplateReference* whereas a document only intended for description purposes must contain at least one element of the elements *NodeTypes*, *RelationshipTypes*, or *Plans*. These syntactical specifications ensure that a Service Template document can be designed modularly.

Node Type

As already mentioned above Node Types are an important part of a Service Template. They describe the properties of one or more Node Templates. The following paragraph discusses the properties of a *NodeType* element which are important in the scope of this thesis.

NodeTypeProperties are the observable properties such as configuration and state of a *NodeType* element. They are defined in the *Types* element of a Service Template or in an external XML Schema file.

TOSCA offers an inheritance/derivation mechanism via the *DerivedFrom* element. It contains a reference to another Node Type acting as a basis for derivation. The properties and operations of the base Node Type either form a union with the newly defined properties and operations or are overridden by the new Node Type in case of conflict.

The optional *InstanceStates* element represents the states a *NodeType* element can occupy once it has been instantiated. Possible states are e.g. started or stopped.

The *Interface* element contains the description of the functions that belong to a certain Node Type. These functions are invoked by operations which are in turn implemented by so called *ImplementationArtifacts*. The Operation element can either define a Web Service call and its WSDL port type and operation or a REST call with its HTTP methods and headers as well as optional parameters. The third kind of operation that can be defined is the script via the *ScriptOperation* element and its input and output parameters. The optional *ImplementationArtifact* element names the concrete artifacts that are needed to implement the abstract operations of the *Interface* element. The artifacts could be Python scripts or WSDL files that can be provided in place or referenced from an external location. The *RequiredContainerCapability* element and its attribute *capability* indicate if there exist particular dependencies to the execution environment of the operation implementation, e.g. the need for environment provided interfaces to manipulate images or EJBs.

The optional *Policies* element is a container for *Policy* elements describing the kind of policies a Node Type instance supports. These policies apply to management aspects such as billing or monitoring.

The last important element of a Node Type is the optional *DeploymentArtifacts* element. It specifies all the concrete deployment artifacts that are required to instantiate a particular Node Type. These could be EAR files for a Java EE application or a virtual image for installing a Java EE Application Server.

Relationship Type

Relationship Types specify the type of one or more Relationship Templates which serve as edges between the Node Templates in a Topology Template graph. Similar to the Node Types the Relationship Types define properties and potential states during runtime but no operations are specified. An important attribute of a *RelationshipType* element is *semantics* which denotes the expected behavior of the *RelationshipType* under definition.

Additional *RelationshipTypeProperties* can be referenced analogous to the *NodeTypeProperties* of the *NodeType* element.

Topology Template

The following paragraph describes the *TopologyTemplate* element of a Service Template and the nested *NodeTemplate*, *RelationshipTemplate* and *GroupTemplate* elements. From the perspective of graph theory a Topology Template forms a directed graph with Node Templates as vertices and Relationship Templates as edges between these Node Templates specifying the relationships. Group Templates form subgraphs of the Topology Template graph, containing Node Templates, Relationship Templates and possibly other Group Templates.

The *NodeTemplate* element symbolizes any kind of component that is part of the IT service under definition. It has a *nodeType* attribute referencing a beforehand defined Node Type. The attributes *minInstances* and *maxInstances* specify the minimum respective maximum number of instances of the Node Template that must or can be created in parallel. Because of the notion of Node Types and Node Templates the term node rather than vertex will be used in the rest of the thesis to avoid confusion.

The optional *PropertyDefaults* element provides initial values for the Node Type properties of the Node Type. The *PropertyDefaults* contain an XML instance of the Node Type Properties' schema definition regarding configuration and state. The inheritance hierarchy of a Node Type via *DerivedFrom* property is also included, i.e. the XML instance document considers the union or overriding of the properties. The *PropertyConstraints* element names constraints regarding the Node Type Properties of the Node Type of the Node Template under definition. The *property* attribute of this element contains an XPath expression pointing to particular XML fragment in the *NodeTypeProperties* element that is to be constrained. The *constraintType* attribute contains an URI explaining the semantics of the constraint.

The *Policies* element has the identical meaning as the one nested inside the *NodeType* element. It specifies management policies such as billing or monitoring but in this case regarding the Node Template. Policies in the corresponding Node Type are overridden if they possess an identical *name* and *type*, the union of the Policies is formed otherwise.

The *EnvironmentConstraints* element contains definitions for constraining the runtime environment of the Node Template standing for a concrete IT component. This could be network security settings or the prerequisite of certain resources.

Similar to the *Policies* element, the *DeploymentArtifacts* element on the *NodeTemplate* level can also override Deployment Artifacts from the *NodeType* level in case of identical names or otherwise complement them. The same overriding and complementation semantics hold true for the element *ImplementationArtifact* (see above).

A Relationship Template specifies the relationship between two Node Templates. The attribute *relationshipType* indicates which of the beforehand defined *RelationshipType* elements provide detailed properties to the Relationship Template under definition. Forming a directed edge, the elements *SourceElement* and *TargetElement* specify the navigation path between two Node Templates. Both elements reference either a Node Template or a Group

Template. The referenced elements must be defined inside the Service Template document under definition. For references to outside Node Templates or Group Templates a *TargetElementReference* element can be specified, but there is no syntax element for referencing an outside source of the edge. *TargetElement* and *TargetElementReference* must not be specified both at the same time inside a *RelationshipTemplate* element.

The *PropertyDefaults* element of a Relationship Template serves the same purpose as its equivalent in a Node Template. Initial values for the corresponding Relationship Type properties are given via a XML document instance. *PropertyConstraints* also refer to the properties of the used Relationship Type properties and specify constrains such as uniqueness of a given property value. The optional *RelationshipConstraints* element contains *RelationshipConstraint* elements specifying constraints on the use of the defined relationship.

The last element to be introduced is the *GroupTemplate* element. It forms a subgraph of *NodeTemplate*, *RelationshipTemplate* and other nested *GroupTemplate* elements. The *GroupTemplate* element has a *minInstances* and *maxInstances* attribute defining the minimal and maximal instances when creating the Group Template. A Group Template can also have its own Policies attached.

Plans

Orchestration Plans specify the operational management behavior of a Service Template. The Plans specify discrete steps called tasks or activities and their order. The steps are executed either by the operations exposed by the Node Templates interfaces or by invoking a Service Template API. The TOSCA specification already names two types of Orchestration Plans: Build plans for the creation of a Service Template instance and Termination plans for the removal of an instance. Modification plans for the managing of instances during lifetime have not yet been developed. This will be done by domain experts and the authors of specific Service Templates.

The *Plans* element of a Service Template contains one or more *Plan* elements. Each *Plan* has an attribute *planType* indicating the type of plan, e.g. build plan or termination plan, by means of an URI (http://docs.oasis-open.org/tosca/ns/2011/12/PlanTypes/BuildPlan and http://docs. oasis-open.org/tosca/ns/2011/12/PlanTypes/TerminationPlan).

To specify the modeling language describing the plan under definition the *languageUsed* attribute is used. It contains an URI indicating the language, e.g. http://www.omg.org/spec/BPMN/2.0/ for BPMN 2.0.

The *Precondition* element contains a condition that has to be satisfied prior to execution. The condition is expressed by an expression language indicated by the *expressionLanguage* attribute. The content of the condition usually relates to the instance states of the Node or Relationship Templates.

The *PlanModel* element contains the actual model specified in the beforehand denoted modeling language, the *PlanModelReference* element provides a reference to the actual model. A *Plan* element instance must either specify a *PlanModel* or a *PlanModelReference* but not both at the same time.

However, the TOSCA Plans are only introduced here for the sake of completeness since the merging of plans is not in the scope of this work.

TOSCA Example

To exemplify the description of an IT service using TOSCA consider the following situation as depicted in Fig. 2.1. The simple Topology Template consists of an application running on an application server. The application is represented by the Application Node Template and the application server by the Application Server Node Template. The "HostedOn"-relationship between the two Node Templates is realized by the HostedOn Relationship Template.

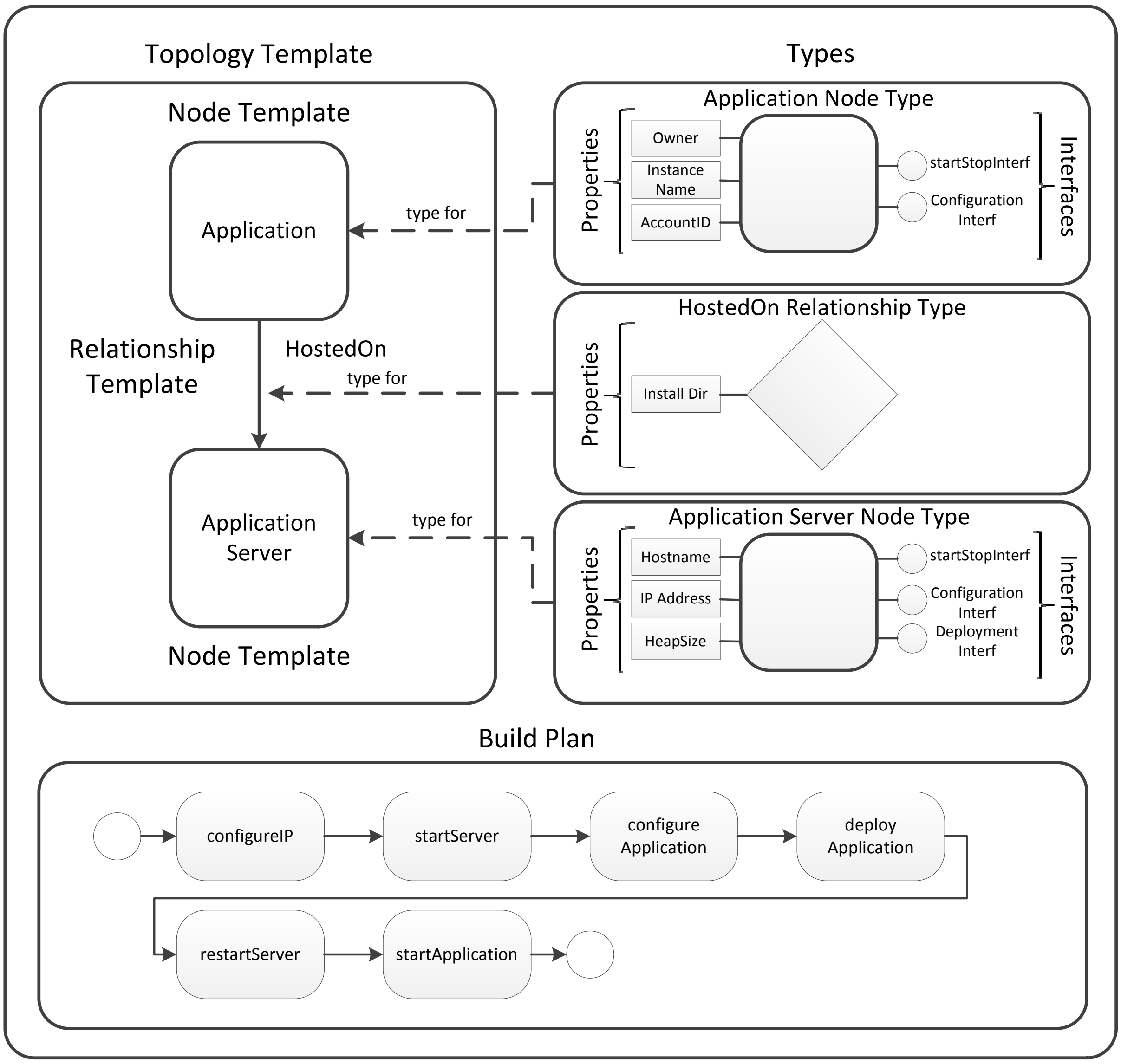


Fig. 2.1: Graphical instance example of a TOSCA Service Template[1]

The two Node Templates and the Relationship Template are typed und therefore enriched with properties by Node Types respectively a Relationship Type. The Node Types additionally provide interfaces that expose operations to interact with the Node Templates. Fig. 2.1

[1] Adopted from [32]

also shows a buildplan illustrating the sequence of activities that must be performed to start und deploy both server and application and bring them up to a working state. The build plan exploits the operations exposed by the displayed configuration and management interfaces.

2.3.2 Use Cases of TOSCA

The authors of the TOSCA specification have proposed several supported use cases [32].

Services as Marketable Entities

The first one is the notion of services as marketable entities. According to the authors, a market for hosted IT services will emerge if Service Templates are standardized. Service developers with profound knowledge about a particular service could create Topology Templates consisting of a set of components and their mutual dependencies und thus define the structure of IT services in an interoperable manner. Service Providers could then select predefined Service Templates out of service catalogs and make them available for potential customers. The service providers would adapt the selected Service Template to their concrete infrastructure, mapping the Topology Templates and management plans to make the Template executable. The necessary management plans, i.e. the plans for managing the whole life cycle of a service from creation to termination, will also be provided by service developers. Having access to those plans there is the potential of significantly reducing service hosting costs for a service provider as they can rely on reusable knowledge and apply management best practices. The domain knowledge of the modeler is encapsulated in the management plans hiding the complexity from its user, who can simply invoke it.

Portability of Service Templates

A second use case is related to the portability of Service Templates. Definitions of IT services become portable when standardizing them with TOSCA. The authors define portability in this context as the ability of one party to understand a Service Template's structure and behavior that has been created by a second party. The creator can be anyone from a cloud provider, to an enterprise IT department or a service developer. The authors point out that the portability they define only refers to the service definitions, i.e. the topology model and the corresponding plans but not to the individual (physical) components. Their portability is not in the scope of the specification.

Service Composition

The third use case the authors cite is service composition. That means that the abstractions provided by a Service Template help to compose components and automation products from different suppliers and hosting providers to one service. This is possible because the Service Template does not imply any particular hosting environment. These properties facilitate strategic decisions such as using datacenters at different geographical locations that together form one IT Service.

2.4 Frameworks

The following section covers the definitions and fundamentals regarding the development of frameworks.

2.4.1 Definitions

Scherp and Boll [42] define a software framework as an only partially finished architectural structure for a complex application area that can be extended for the specific requirements of a particular application. Literature [42], [10], [37] names the following characteristics of frameworks: (1) a framework inverts the control flow of an application from a so called *Call-down-principle*, where the application logic invokes functions of class libraries, to a so called *Call-back-* or *Hollywood-principle* (“Don’t call us, we’ll call you!”), where the framework invokes the application-specific parts of an application. (2) a framework specifies a concrete software architecture that defines the generic functionality and only allows for flexibility at particular points. Thereby, the inversion of control leads to a “software-skeleton” implementing generic algorithms that can be extended to satisfy the specific requirements of a particular application. (3) the framework is extendable through so called *variation points* or *hot spots*. They characterize the points of the architecture where a particular functionality is already typed but the final implementation is done through the concrete application using the framework.

Furthermore, Scherp and Boll [42] distinguish between *component-based* and *object-oriented* frameworks. A component-based framework provides a set of components with predefined behavior and interfaces. The extensibility of the behavior results from composing the components to realize specific functionality or adding new components implementing the interfaces. However, the component-based framework defines no internal architecture for the components. On the other hand, the object-oriented frameworks consist of a set of concrete and abstract classes that provide a generic software system for a particular application area. The variation points are the abstract framework classes that can be extended using inheritance and polymorphism [23], i.e. the application specific extensions are achieved by subclassing the abstract classes and providing appropriate implementations. These subclasses are then invoked by the generic framework using the aforementioned Call-back-principle. Object-oriented frameworks should also follow the important architectural *open/close principle*. This states that software, and an object-oriented framework in particular, should be open to future extensions but closed with regard to the modification of the existing generic code. This implies that no piece of code that has already been implemented should be overridden by application specific classes. However, as Scherp and Boll [42] point out, real life examples such as the framework Java Swing for Graphical User Interfaces (GUI) often provide a default implementation that can also be overridden by users of the framework.

According to Buschmann et. al. [10] an object-oriented framework is not limited to object-oriented techniques such as inheritance and polymorphism but could also apply software design patterns. A software design pattern is a generic and well-proven solution to a software design problem. Two patterns are notably suitable for building frameworks and are used to design the architecture of the TOSCAMerge framework: the Template Method and the Factory method. These two patterns will be described hereafter and are based on [15]. The figures use UML [33] as modeling language.

2.4.2 Template Method

The Template Method design pattern is the pivotal pattern for framework design and is depicted in Fig. 2.2. The abstract, i.e. not instantiable, *FrameworkClass* provides the structure for the framework and delegates the implementation of specific processing steps to its subclasses. The so-called *template method* defines the overall algorithm, using the abstract *hook methods* to call the application specific implementation at the desired time in the control flow.

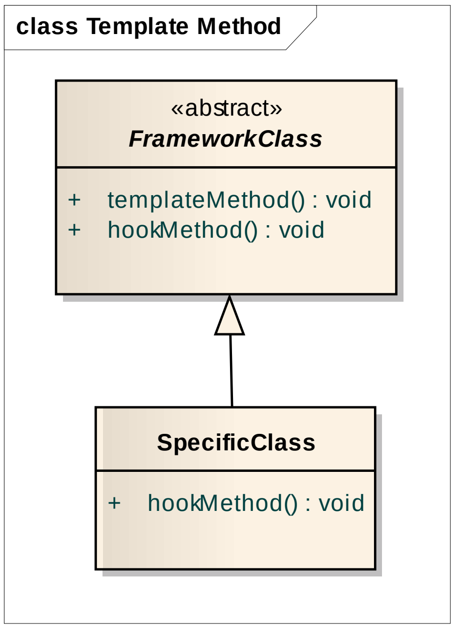


Fig. 2.2: Template Method design pattern[2]

This is illustrated by the corresponding code example in Listing 2.3. The method *templateMethod* invokes the abstract method *hookMethod* after executing some generic steps and continues after the return of the method. The *templateMethod* is marked as *final* to comply with the *open/close-principle* and prohibit the modification of the overall algorithm by overriding the templateMethod. The SpecificClass in Fig. 2.2 extends the *FrameworkClass* and overrides the abstract method *hookMethod* providing its own specific solution to a problem.

[2] Adopted from [15]

```
Framework class using the Template Method pattern
public abstract class FrameworkClass{
  public abstract void hookMethod();
  public final void templateMethod(){
    // generic algorithm steps
    hookMethod();
    // generic algorithm steps
  }
}
```

Listing 2.3: Code example of a framework class using the Template Method pattern[3]

This is clarified further by the code example in Listing 2.4. The non-abstract *SpecificClass* extends the *FrameworkClass* by inheritance and overrides the *hookMethod* to provide specific steps for the overall solution of a particular application area specific problem.

```
Specific class using the Template Method pattern
public class SpecificClass extends FrameworkClass{
  public void hookMethod(){
    // specific algorithm steps
  }
}
```

Listing 2.4: Code example of a specific class using the Template Method pattern [4]

2.4.3 Factory Method

The second design pattern used in the context of frameworks is the Factory Method.[5] The goal of the Factory Method design pattern is to decouple the generic programming logic from the concrete implementation classes. This is especially useful if not all the concrete classes are known during build time. Therefore, the generic (framework) code is implemented against interfaces or abstract classes and is independent of concrete subclasses. The classical Factory Method design pattern is depicted in Fig. 2.3. The *Factory* class provides an abstract *factoryMethod* that declares an interface for creating objects of type *Product*, i.e. extending the abstract *Product* or also possible implementing an Interface of that name. For every *ConcreteProduct* that implements or overrides the *Product*, a corresponding *ConcreteFactory* that knows how to specifically create a ConcreteProduct has to be provided.

[3] Adopted from [15]
[4] Adopted from ibid
[5] and also its variation the Abstract Factory

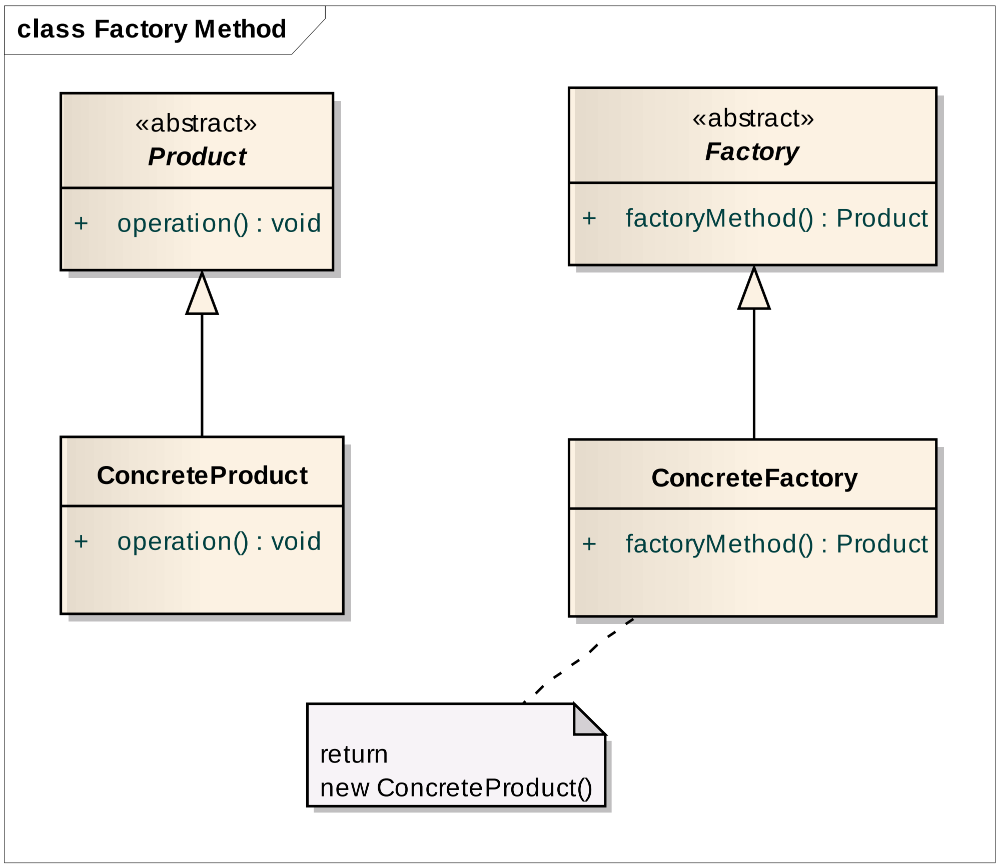


Fig. 2.3: Factory Method design pattern[6]

In the following Listing 2.5, Listing 2.6, Listing 2.7 and Listing 2.8 provide a simple continuous example of the usage of the Factory Method.

```
Abstract Factory using the Factory Method
public abstract class Factory {
    public abstract Product createProduct();
}
```

Listing 2.5: Code example of abstract factory class of the Factory Method design pattern

The abstract class *Factory* in Listing 2.5 shows the aforementioned abstract factory that defines an abstract factoryMethod to create a new instance of a class implementing a particular interface, in this case the abstract class *Product* not depicted here but being the same as in Fig. 2.3. A concrete implementation of the abstract factory class is shown in Listing 2.6. The class overrides the *createProduct* factory method and returns an instance of the *ConcreteProduct* class from Fig. 2.3. The choice of the specific factory must take place only once, e.g. in an initialization method of the framework as depicted in Listing 2.7. In the rest of the code the factory creates instances by utilizing polymorphism without knowing the exact type of *Product*.

[6] Adopted from [15]

```
Code example for a concrete implementation of the Factory Method
```

```java
public class ConcreteFactory extends Factory{
    public Product createProduct(){
      return new ConcreteProduct();
  }
}
```

Listing 2.6: Concrete factory class of the Factory Method design pattern.

```
Code example for an application initialization
```

```java
public void init(){
    Factory myConcreteFactory = new ConcreteFactory()
      ...
  }
}
```

Listing 2.7: Initialization of a factory

```
Code example for a concrete implementation of the Factory Method
```

```java
public doSomething(){
    Product newProduct = myConcreteFactory.createProduct()
  }
}
```

Listing 2.8: Usage of the factory

Fig. 2.4 shows the combined patterns Template Method and Factory Method to design a framework. The abstract *FrameworkClass* defines a *templateMethod* to specify the overall control flow of the application using the framework and a *hookMethod* that can be overridden by instances of *SpecificClass* to provide application area specific implementations to the overall algorithm. The Factory Method part defines an abstract *Factory* class that specifies a factory method for creating Instances of *SpecificClass.* For each different type of *SpecificClass,* a separate *SpecificClassFactory* that has the knowledge of creating an instance of a particular type of *SpecificClass* must be provided.

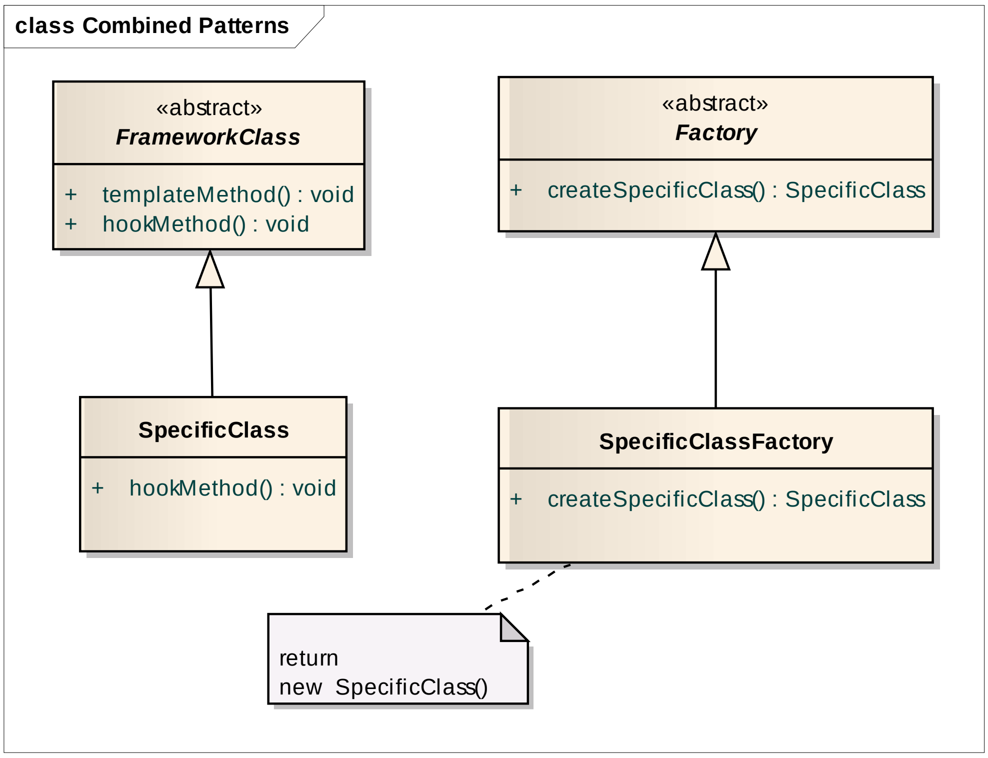


Fig. 2.4: Combined design patterns

3 Related Work

The following chapter discusses related work in the area of merging of graphs, business processes and (database) schemata and other graph like structures. Thereby, it is the aim to evaluate their usefulness and to derive concepts for merging of TOSCA Topology Templates.

Process and Graph Matching and Merging
In the light of mergers, take-overs, and acquisitions of companies, Gottschalk et al. [19] introduce an algorithm for the merging of Event-Driven Process Chains (EPC). The algorithm integrates two process models described by EPCs into one process model. The behavior of the original process models is preserved in the resulting one. The authors divide the process merge in three phases. First the problem of EPC merging is reduced to a graph merging problem by formally describing the process models by means of so-called *Function Graphs*. A Function Graph depicts the active behavior of an EPC, i.e. its sequence of functions. Therefore functions represent the vertices in a Function Graph while directed edges are annotated with types that depict the $\wedge$, XOR, or $\vee$ behavior, i.e. the split and join behavior during process execution. The second phase of the algorithm is the actual merge. The two Function Graphs are united by merging their sets of functions and directed edges and calculating the split and join types of the edges by analyzing the functions in both graphs preceding and succeeding the corresponding arc. The third phase of the algorithm transforms the resulting Function Graph back into an EPC. The proposed approach of Gottschalk et al. is not used for the merging of Topology Templates as it is connected with EPCs too closely. However, the aim of preserving all the behavior of the original process models will be translated into the context of this work.

La Rosa et al. [25] also use business processes modeled by the EPC notation as input for their business process model merging algorithm. The problem is reduced to a graph merging problem introducing the concept of an *Annotated Business Process Graph* formally representing an EPC process model. In contrast to [19] not only functions but also events and connectors are depicted by typed graph nodes. To establish the best possible mapping between the nodes of two input graphs a process matching is carried out. The authors calculate a matching score using edit distance and graph edit distance to find the best mapping between the nodes. Mappings of nodes of different types are given a matching score of zero. The labels of functions and events are scored using edit distance [40], the similarity of connector nodes is calculated by analyzing the so-called presets and postsets of the connector nodes. Presets and postsets are sequences of nodes before and respectively after a particular node. The scoring function furthermore counts and weights the number of node and edge substitutions, insertions and deletions to transform one graph into the other (analogous to graph edit distance [22]). The mapping is used as an input to the actual merge algorithm. The algorithm returns a merged graph which has to be post-processed by a set of reduction rules, e.g. redundant edges have to be removed. The evaluation of the algorithm showed that it is idempotent, commutative and associative. A process model can be merged with itself producing an unchanged model (idempotency). The other two properties state that more than two models can be merged without importance of the merging order. The work in [25] provides valuable insight into the high level approach of merging of graphs: (1) a mapping between the nodes of two graphs has to be found, this is called graph matching. (2) mapped nodes in the graphs are merged. (3) optional post-processing steps have to be conducted. This work will also follow this high level pattern.

Schema Matching

Literature from the research area of information integration also suggests this approach [26]. Information integration deals with the integration and merging of database or XML schemata. Leser and Naumann point out that a mapping is an important concept in this context. A mapping is a set of *correspondences* between the elements of a schema that share the same semantics. The mapping is then e.g. used to generate queries that integrate data from two databases. The correspondences can have a 1:1, 1:n, n:1 and n:m characteristics. Having other matching characteristics than 1:1 correspondences leads to computational complexity issues. Instead of $M \times N$ comparisons to find the complete mapping $2M \times 2N$ comparisons have to be performed as every possible subset from the one schema has to be compared with every possible subset from the second schema leading to exponential complexity.

Constructing a mapping for a large schema by manual identification of correspondences proves to be too cumbersome. It requires automated approaches, however, Leser and Naumann argue that schema management and notably the finding of mappings involves many only implicitly given semantics that require domain experts to confirm the findings of automated algorithms. One of these algorithms that is also graph based is *Similarity Flooding* by Melnik et al. [28]. Two database schemata are transformed into directed graphs containing attributes, data types, and other constructs (e.g. table names) as nodes. Using edit distance an initial mapping is established. Subsequently Similarity Flooding is used on all mapped pairs between the graphs. If two nodes are similar they also contribute to the similarity of their neighbor nodes. The similarities are propagated until a fix point is reached. A threshold value indicates the best generated mappings. The authors see Similarity Flooding as a semi-automated process where domain experts review the generated mappings: The number of adjustments a human expert has to perform on the correspondences is seen as a quality metric for the algorithm. The schema integration approach and its partition in the steps matching and integrating once again gives insight how to structure the approach of this thesis. Furthermore, it shows that the expertise of domain experts is necessary to evaluate the algorithm's result afterwards or that the domain experts' knowledge must be codified in an extension of the algorithm beforehand. The proposed quality metric also implies that the algorithm should be able to do as much as possible in an automatic fashion and the less human correction is required the better. This understanding is also used for the merging of Topology Templates.

Notion of a Correspondence Between Similar Elements

The matching part in the aforementioned high-level approach is further studied in [13]. Dijkman et al. contribute four algorithms that match two process graphs and return a mapping with the highest similarity. As the naïve approach to the mapping problem, i.e. the construction of all possible mappings has factorial time complexity, the four algorithms try to reduce complexity by using heuristic measures. The authors propose a greedy algorithm constructing a mapping that maximizes the similarity of the graphs in each step, an exhaustive algorithm with pruning if its recursion tree reaches a specified size, a process heuristic algorithm that is a deviation of the exhaustive algorithm with consideration of the relative position of nodes and finally a variation of the A-Star heuristic search algorithm. Although the proposed algorithms in [13] cannot be directly used in this thesis, it can be noted that generating a mapping between nodes of two graphs is a problem of high time complexity and either heuristic measures or restricting assumptions have to be applied to decrease problem space.

Pottinger and Bernstein [38] examine the generic merging of two models such as UML, ontology models or database schemata. They also use the notion of *correspondences* between two models. The correspondences are assumed to be given and their generation is not further discussed in their work. The contributed merge operator returns a "duplicate-free union" of the two mapped input models. Arising conflicts consisting of so-called *representation*, *metamodel* and *fundamental* conflicts are identified and resolution strategies are provided. The introduced approach provides *Generic Merge Requirements* such as element preservation, equality preservation and relationship preservation that are adopted in this work.

Küster et al. [24] contribute an approach to merging of business process models in absence of a change log. A copied and altered version of the original process model is to be merged with the original one. Differences between the models have to be detected as there is no change log that describes the carried out changes. The differences are resolved requiring some manual aid by a domain expert. Their work makes also use of the concept of *correspondences* and enriches them with the technique of *Single-Entry-Single-Exit* fragments (SESE fragments). As Topology Templates do not contain block-structured SESE fragments as a business process this approach is not suitable for this work, however, the notion of *correspondences* is picked up.

Merging of Petri Nets

Sun et al. [46] describe an approach for merging process models described by Petri nets. They introduce the concept of *Merge points*, i.e. place nodes in in both Petri nets that are mapped on each other. However, a method to find matching places in a Petri net and thus construct a mapping is not provided. Subsequently so-called merge patterns are applied to merge both process models. Therefore a domain expert must provide input how the places and transitions between two merge points, i.e. a starting and an ending Merge point have two be merged. Possible patterns the expert can choose from are *Sequential*, *Parallel*, *Conditional* and *Iterative* merge. The discussed approach is not suitable for our work as it only works for block-structured workflows and not for general graphs. Moreover too much additional manual input is required.

Graph Transformations

Segura et al. [45] introduce a completely different approach to the merging problem than those discussed before. The application domain is the merging of *Feature Models* in the context of *Software Product Lines*. Feature Models represent the features of the software products in a Software Product Line as a tree-like structure. The authors use *Graph Transformations* to merge two Feature Models. They present a catalogue of 30 merge rules consisting of two input patterns (subgraphs) and a corresponding output pattern. The input patterns are called left-hand side and the output patterns right-hand side. The rules are implemented in the Attributed Graph Grammar System (AGG), a free Java tool for Graph Transformations.

Graph Transformations are also used by Gala et al. [16] to merge the navigation histories of web site users to utilize them for Web Mining. Web sites and the interactions and navigations with the websites are modeled as so-called *semistructured temporal graphs* and merged to a summarizing graph structure using Graph Transformations. A web site's objects are represented by the nodes of the graph, the navigational links by edges. The navigation of a user during a session is also a semistructured temporal graph that forms a sequence of nodes through the web site graph. Each node in a web site and the navigation graph is annotated with a temporal element (hence the label temporal) indicating either the temporal validity of a web site object in case of the web site graph or the visiting and retention time of the navigation graph. The authors define one Graph Transformation rule merging each node with the same object id unifying their temporal elements. The corresponding edges are collapsed to the merged node.

Both Graph Transformation approaches are not directly comparable to our approach, but they are either limited to special graph structures such as trees in case of [45] or narrowly specialized for a particular application domain like [16]. However, the notion of left-hand side and right-hand side can also be used in context of this work to refer to the elements being involved in the matching and merging.

4 Assumptions and Requirements for Matching and Merging

As already pointed out in chapter 2.4, this master's thesis will follow the high-level approach of graph and schema merging. The fundamental steps are graph matching to find a set of correspondences, called a mapping, and then utilize these correspondences to unify the source and the target elements. Section 4.1 states all the assumptions that are made with regard to merging and matching. Section 4.2 discusses the requirements for the concepts of matching and merging and the algorithms implementing them.

4.1 Assumptions

The following section discusses the assumptions made for this thesis. The assumptions delineate the scope of the thesis and exclude elements that would go beyond it.

Assumption 1: Matching and merging of two Topology Templates

This master's thesis only considers the matching and merging of two Topology Templates. The consideration of more than two Topology Templates at once is beyond the scope.

Assumption 2: Equality of TOSCA elements is indicated by their qualified name

Node Types are considered equal if their *id* attributes have identical values and their namespaces are identical.

Assumption 3: Only exact matching of Node Templates and Relationship Templates.

This work only considers exact matches between Node Templates. Therefore, similarity scoring techniques such as edit distance on the Node Type or Node Template ids or names to rate the similarity in an interval between 0 and 1 are not used. The similarity used in this thesis only has the discrete values 1 and 0 indicating full correspondence or none at all. The same holds true for Relationship and Group Templates.

Assumption 4: Only 1:1 correspondences between the Node Templates of a Topology Template

One underlying assumption is that there exist only 1:1 correspondences between the Node Templates of two Topology Templates. Fig. 4.1 shows a valid example of a mapping between a database Node Templates and Linux operating system (OS) using only 1:1 correspondences.

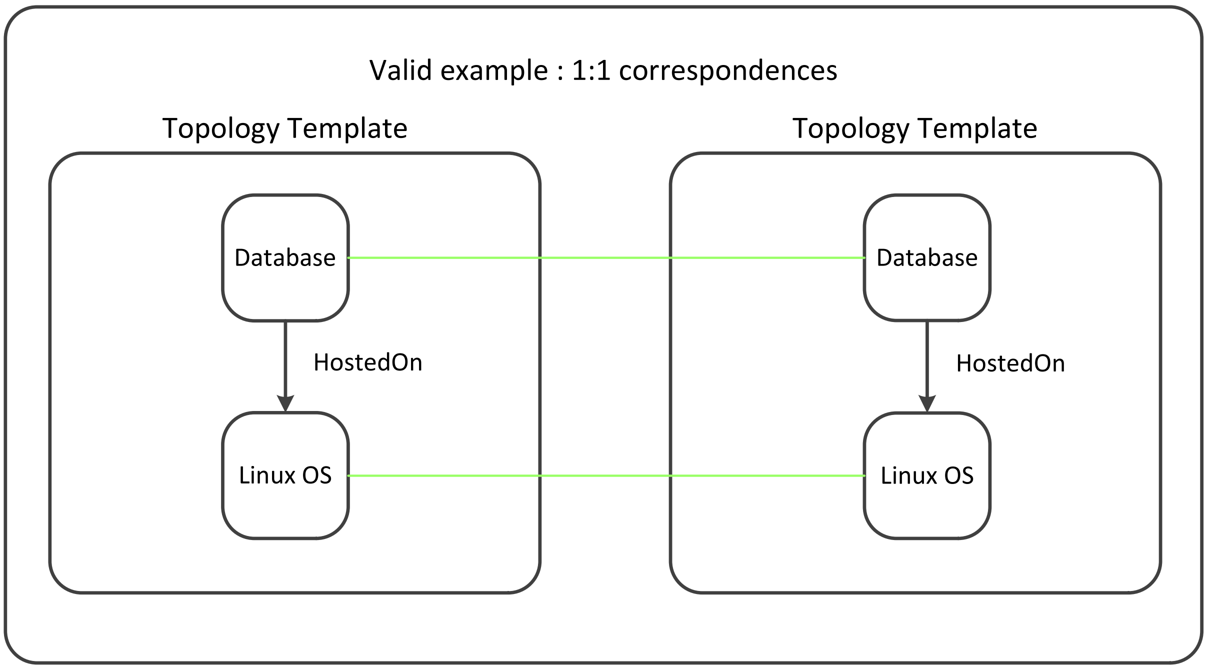


Fig. 4.1: Valid example of 1:1 correspondences between Node Templates

Fig. 4.2 in contrast shows an example of a n:1 correspondence. The TOSCA specification does not imply any kind of modeling style and granularity of Node Templates. In an extreme case one could consider a Topology Template consisting only of one Node Template representing the whole service structure as valid in the sense of the specification. However, the aggregation of Node Templates that can be viewed as separate entities camouflages the structure of a service and impedes the comparison of Node Templates with the same semantics.

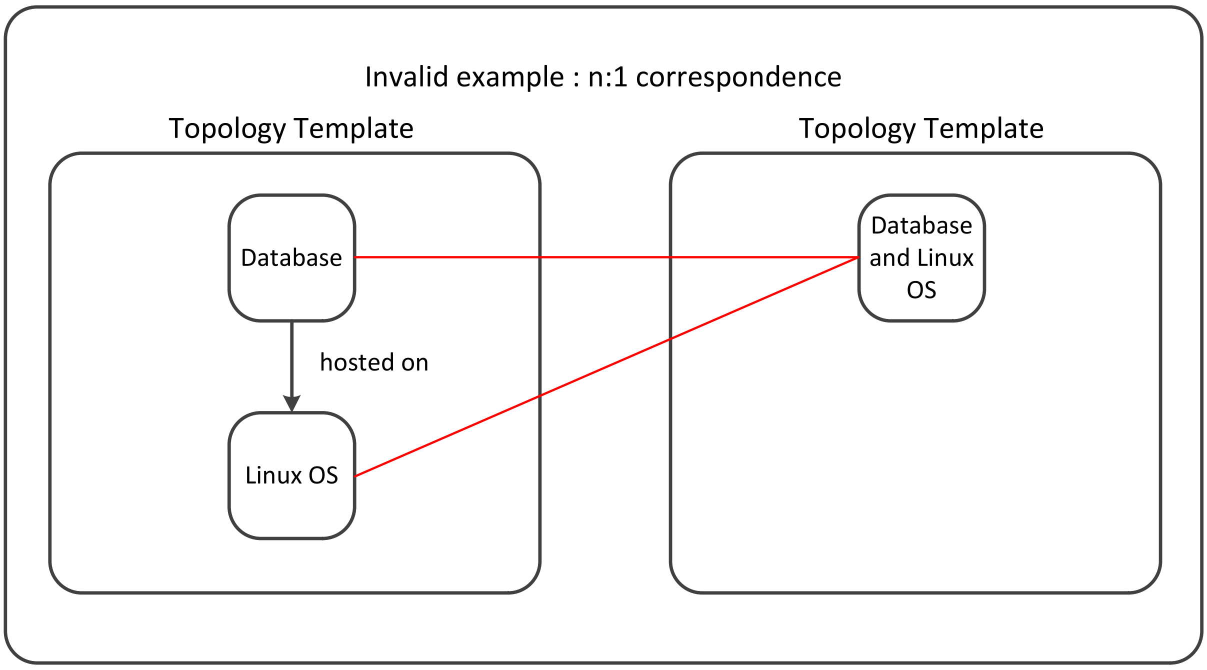


Fig. 4.2: Invalid example of a n:1 correspondence between Node Templates

Assumption 2 and 3 also imply that 1:1 correspondences are considered exclusively. This arises from the fact that (1) the correspondences in Fig. 4.2 cannot be calculated if the Node Types of the Node Templates have different ids and (2) without approaches to syntactic or semantic similarity, which are explicitly excluded from this work, no correspondences between the separate entities of database and OS on the left-hand side and the aggregated entity on the right-hand side of Fig. 4.2 can be established.

Assumption 5: TOSCA elements have the same modeling granularity

In assumption 4 it was stated that only 1:1 correspondences are considered when matching. Additionally this thesis assumes that all Node, Relationship and Group Templates have the same modeling granularity and aggregated elements such as in Fig. 4.2 do not occur in the Topology Templates respectively are not considered.

Assumption 6: A limited set of Node and Relationship Types is sufficient for this thesis

It is assumed that a limited set of Node and Relationship Types is sufficient to develop the matching and merging concepts and that it is easily possible to transfer the findings to additional types. Furthermore, only for the specified set below a prototypical implementation provided for. The types are organized as a tree. Fig. 4.3 shows the Node Types that are relevant for this work, as indicated by the green color.

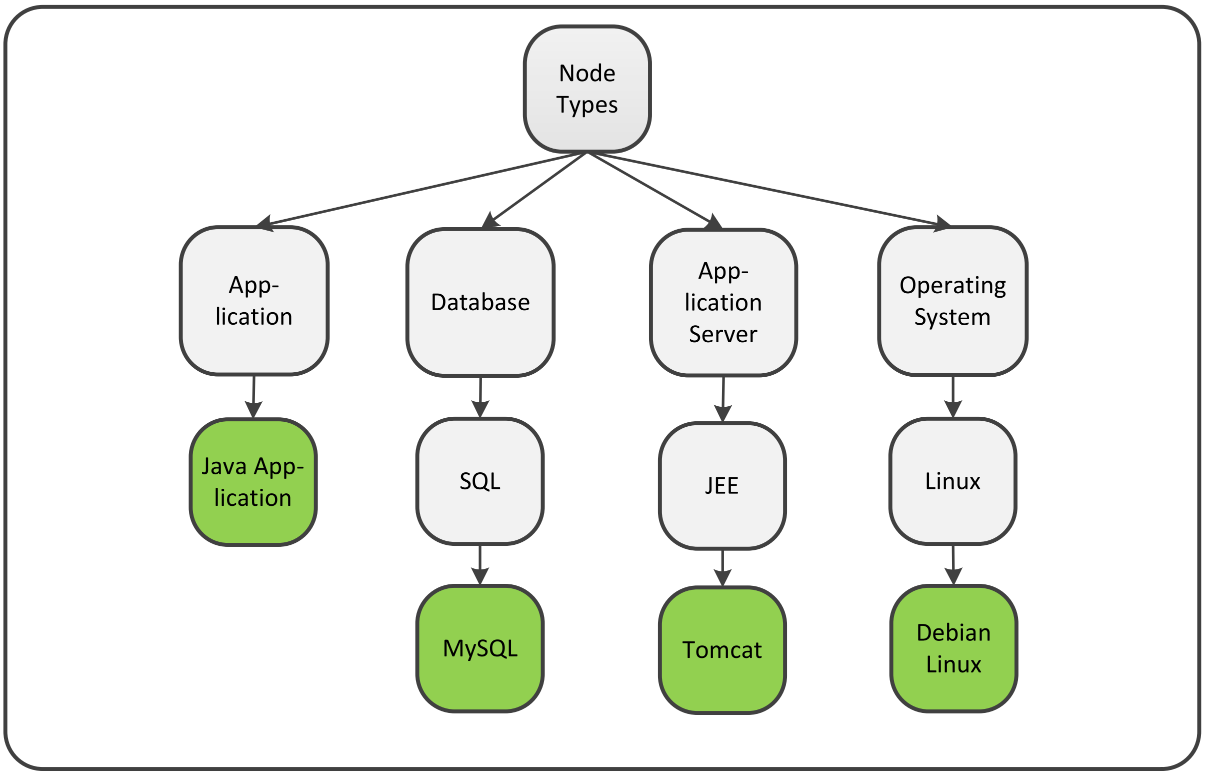


Fig. 4.3: Relevant Node Type tree

Fig. 4.4 illustrates the three Relationship Types that are discussed in this work. The *HostedOn* Relationship Type of a Relationship Template has the semantics of the source Node Template being installed or deployed on the target Node Template.

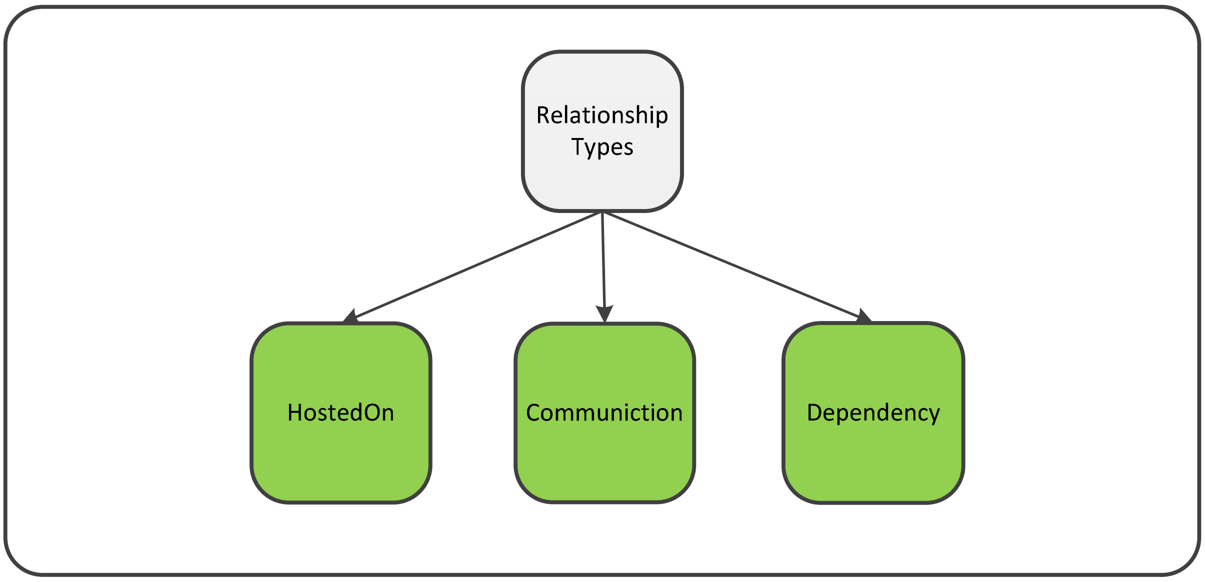


Fig. 4.4: Relevant Relationship Type tree

The assumption is that each Node Template can only be source for one HostedOn Relationship Template whereas each Node Template can be the target of many HostedOn relationships. The *Communication* Relationship Type indicates that a corresponding Relationship Template forms a communication link between two Node Templates. Every Node Template can be source and target of an arbitrary number of Communication-typed Relationship Templates. It is even possible that several communication links exist between two particular Node Templates. The *Dependency* Relationship Type provides a Relationship Template with the semantics of a dependency relation. The source Node Template of such a Relationship Template depends on another Node Template depicted by the target of the Relationship Template.

Assumption 7: Only XML schema types are considered

Although the TOSCA specification allows for any type system in the Type section of a Service Template to declare the properties of the Node and Relationship Types, this thesis only considers XML schema types.

Assumption 8: Relationship Templates target Node Templates inside a Group Template but not the Group Template itself.

Assumption 9: Topology Templates have valid semantics

All Topology Templates that have to be matched and merged have valid semantics, e.g. no IT component instance is hosted on two or more other IT component instances. Valid in this context means that Topology Templates are modeled in a way that reflects reality. Although it is possible to define a Node Template as the source of a Relationship Template with HostedOn semantics to more than one other Node Template, in a real IT environment this makes no sense.

Assumption 10: No discussion of TOSCA import mechanism

The TOSCA import mechanism is not within the scope of this thesis. Without restricting generality it is assumed that the functionality of importing other Service Template documents, WSDL files or XML schema documents and generating a resulting Service Template is done by a generic importer that can be used in a step before the matching of the Topology Templates.

4.2 Requirements

The following section postulates requirements that the matching and merging concepts and the corresponding algorithms must adhere to.

General Requirements

As already mentioned in Chapter 3, Pottinger and Bernstein define a set of so-called Generic Merge Requirements [38]. Some of these requirements are picked up and adapted for the concepts and algorithms of this work.

Element preservation: Each element, in this case Node and Group Templates, of the two input Topology Templates that is not source or target of a correspondence must be element of the resulting merged Topology Template. That means no element must be discarded. The elements that correspond to each other must be part of the merged Topology Template as unified elements replacing their originators.

Relationship preservation: The same requirement is true for Relationship Templates. Each input relationship between the Node Templates that is not source or target of a correspondence, i.e. is not merged, must be element of the resulting Topology Template. The relationships that correspond to each other must be replaced by a merged relationship.

Extraneous item prohibition: No additional elements should be generated that were not part of the input Topology Templates.

Property preservation: The properties of each element must be preserved in the merged result. A merged element must not have unified properties that contradict its original semantics.

Value preference: When merging the properties of two elements and it does not matter which value is used for in the unified model, e.g. the *name* attribute of a Node Template, the value of the source of the correspondence, i.e. left-hand side element is used. Pottinger and Bernstein call the left-hand side of the comparison the *preferred model.*

Semantically correct results: The concepts and the implementing algorithms should not only produce syntactically correct results that adhere to the TOSCA specification but also generate semantically correct results. That means that the merged Topology Templates must not contain elements that are implausible in the context of an IT environment.

Inclusion of domain-specific knowledge: After the study of related work in Chapter 3, it can be presumed that not all decisions of the matching and merging of Node, Relationship, and Group Templates can be made generically. Thus, either the concepts and the implementing algorithms exclude domain-specific knowledge and require manual action to solve corresponding problems or the knowledge can be integrated into the generic concepts and used to solve the arising problems. This master's thesis will follow the latter approach and incorporate the generic concepts into the TOSCAMerge framework which invokes plugins implementing domain-specific knowledge when needed. To alleviate the use of the framework, it must allow for simple adding of plugins. This thesis follows a *closed world assumption* approach that states that everything that is not modeled is assumed to be false [39]. That means for the plugins if no appropriate ones can be found, the framework assumes that matching respectively merging is not possible.

Requirements for the Algorithms and Their Implementation

After having stated the general requirements for the concepts and algorithms this section specifies some additional requirements that apply to the matching and merging algorithms only. The first two are often applied to all kinds of algorithms in general [40]: Termination of the algorithm and a deterministic result.

Termination of the algorithms: The algorithms to find a mapping between the TOSCA elements and merging them subsequently should always terminate after a number of finite steps and present a result.

Deterministic result: If the algorithms are executed again the same result should be generated, provided the input was the same.

Practicable computational complexity: The proposed algorithms for matching and merging should have a combined computational complexity that allows for the solving of the tasks in a reasonable time. According to Saake and Sattler [40], this is achieved by having less or equal to quadratic computational complexity. Therefore, the algorithms in this work should be designed with the aim of terminating within an amount of time that allows practical use even with more than a few hundred elements.

Assessability of the intermediary results: Pottinger and Bernstein proposed the number of necessary human corrections as a quality metric for their algorithm. As mentioned above the algorithms of this work should integrate domain-specific knowledge into the generic concepts in the form of plugins to a generic framework. Thus, it is the aim of this work to minimize the required interaction. However, it may not be possible to codify every bit of domain specific knowledge into the plugins. Therefore, the framework must provide the possibility to review the intermediary results, i.e. the mapping, and possibly correct it by adding or deleting correspondences. Subsequently, it must be possibly to proceed with the merging.

5 Concept for Matching of Topology Templates

The overall approach of this master's thesis consists of two high-level steps namely matching and merging. This chapter deals with the matching of TOSCA elements to find correspondences between them. Section 5.1 discusses the different cases and situations that can be found when matching the Node Templates of two Topology Templates. For each case an algorithm respectively a subroutine of an overall algorithm is presented and explained. Section 5.2 introduces the matching of Relationship Templates while Section 5.3 extends the concepts of matching Node and Relationship Templates by considering their position in Group Templates. Moreover, the finding of correspondences between Group Templates is also part of this section. Altogether Chapter 5 forms the conceptual basis for the matching functionality of the TOSCAMerge framework.

5.1 Matching of Node Templates

The matching of two Topology Templates is needed to identify correspondences between the Node Templates. According to assumption 4 we assume that only 1:1 correspondences are expected in the matching process. This is necessary as it is not in the scope of this work to conduct inexact matchings that only state the possibility of a correspondence. The exclusion of 1:n, n:1 and m:n mapping characteristics also avoids the necessity to compare every possible subset of Node Templates of one Topology Template with every possible subset of Node Templates of another Topology Template. Instead of $2^m \times 2^n$ comparisons and $O(2^m \times 2^n)$ time complexity, at most $m \times n$ comparisons are needed yielding a time complexity of $O(m \times n) \cong O(n^2)$ with m the number of Node Templates in the first Topology Template and n the number of Node Templates in the second Topology Template. That implies that in principle the Topology Templates are matched by comparing every Node Template in the first one (left one or left-hand side) with every Node Template in the second one (right one or right-hand side). This basic pattern will be extended during the following discussion of the different cases.

To clarify the concept of correspondences used in this thesis the following definition is provided:

Definition 5.1 (Correspondence): *In general a correspondence is an overlay edge indicating that one node corresponds to another node und therefore the two nodes can be merged. Overlay means that a correspondence is added on top of the existing graph structure.*

Although a correspondence can be seen as a directed edge between two nodes, as the beforehand comparison starts from one node to all the others and the Correspondence is the result of a successful matching, in the following illustrations the Correspondences are not depicted as arcs. The reason is that for the matching cases it does not matter from which node the matching started, therefore, the arrowheads are omitted for simplification.

5.1.1 Analysis of the Basic Case and its Derivations

Basic Case for Node Templates

The basic case is illustrated in Fig. 5.1. A Correspondence between two Node Templates is established if and only if their Node Types are identical. For the sake of simplicity in the basic case, it is assumed that all Relationship Templates have HostedOn semantics and no PropertyDefaults that could contradict each other. Later this assumption will be dismissed and different Relationship Template semantics will be included in the matching concept. In the example at hand both Topology Templates have identical MySQL *Database* and Debian-Linux *operating system* Node Templates. The Relationship Templates have a *HostedOn* semantics indicating that each database is hosted, i.e. installed on an operating system. Since the Node Types are identical in each case, a so called *Node Template Correspondence* can be established.

Definition 5.2 (Node Template Correspondence): *A Node Template Correspondence is a Correspondence as generically defined in Definition 4.1 with the constraint that it is only established between two matching Node Templates.*

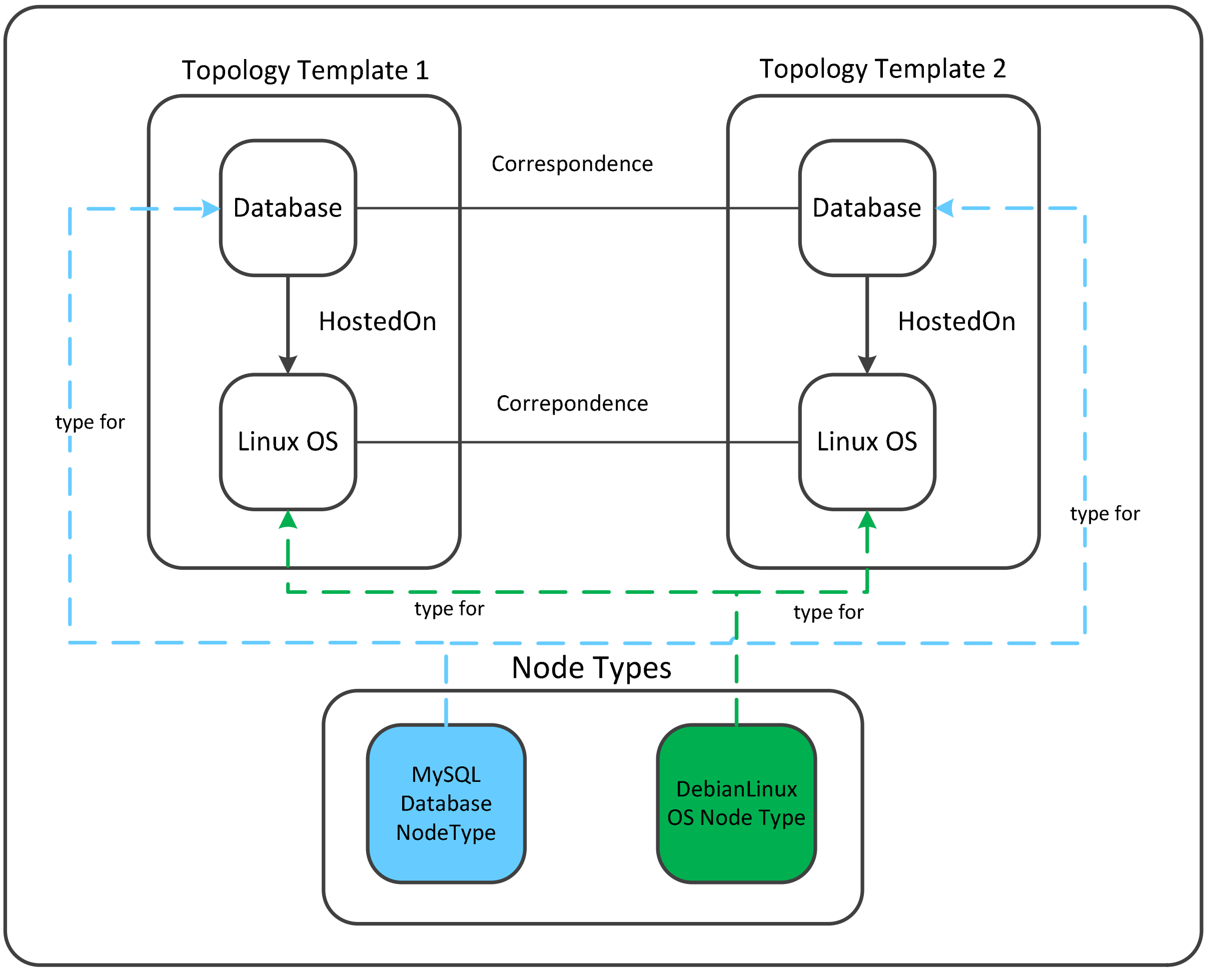


Fig. 5.1: Example of a general matching case

No Corresponding Node Templates in the Second Topology Template

Fig. 5.2 illustrates the case when there exists a possible Correspondence between two Node Templates in one Topology Template but there is no equivalent Node Template with a corresponding Node Type in the second Topology Template. The comparison of all nodes of Topology Template 1 with all nodes of Topology Template 2 does not reveal the correspondence between the Application Sever Node Templates as there is no counterpart in Topology Template 2. Only a comparison inside Topology Template 1 would reveal the Correspondence. A comparison of every element with every other element inside a Topology Template means a time complexity of $O(n^2 - n)$ where n is the number of Node Templates. With regard to both Topology Templates the time complexity is $O((n^2 - n) + (m^2 - m))$. Adding the complexity for the actual comparison between the two Topology Templates one yields a complexity of $O((n^2 - n) + (m^2 - m) + (m \times n)) \cong O(3n^2) \cong O(n^2)$. That means it is still about quadratic time complexity.

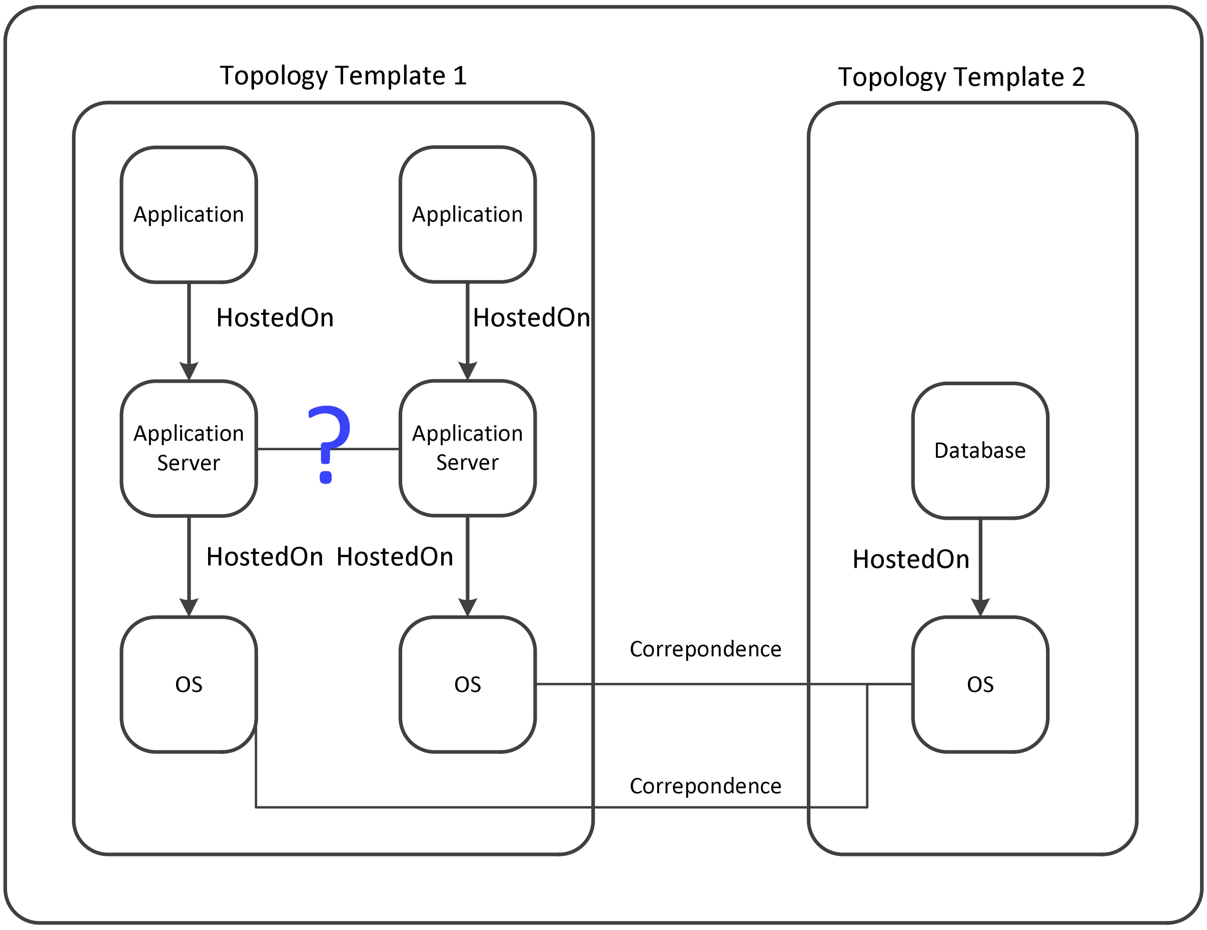


Fig. 5.2: Motivation for matching inside a Topology Template

This work follows the Divide-and-conquer principle that divides a problem into smaller problems and applies an algorithm to the sub-problem in order to solve them more easily [40]. The author proposes to perform the whole matching and merging on each Topology Template separately and merge and match the results in a next overall step. Thus in the following it will be explicitly stated if the matching inside a Topology Template or between two Topology Templates is meant if there is a difference. If nothing is explicitly said, it applies to both cases.

Node Types of Node Templates that cannot be Matched

There exists a number of Node Types that cannot be matched or merged. This applies e.g. to all kinds of business applications. Even if the Node Types have the same the intention, the implemented business logic is different and cannot be combined to a single Node Template. Fig. 5.3 shows this matching situation. A Node Template Correspondence between the two *Application Servers* can be established without difficulty, even if they can be seen as applications as well, i.e. installed on operating system nodes. However, their internal logic is generic and can be unified. In this example the two *Application* Node Types do not match regardless of their identical Node Types.

To determine which Node Types represent non-matchable Node Types, the TOSCAMerge framework must contain an extendable list with the ids and targetNamespaces of the relevant Node Types so that any Node Type for which matching and merging is not desired can be added to this list.

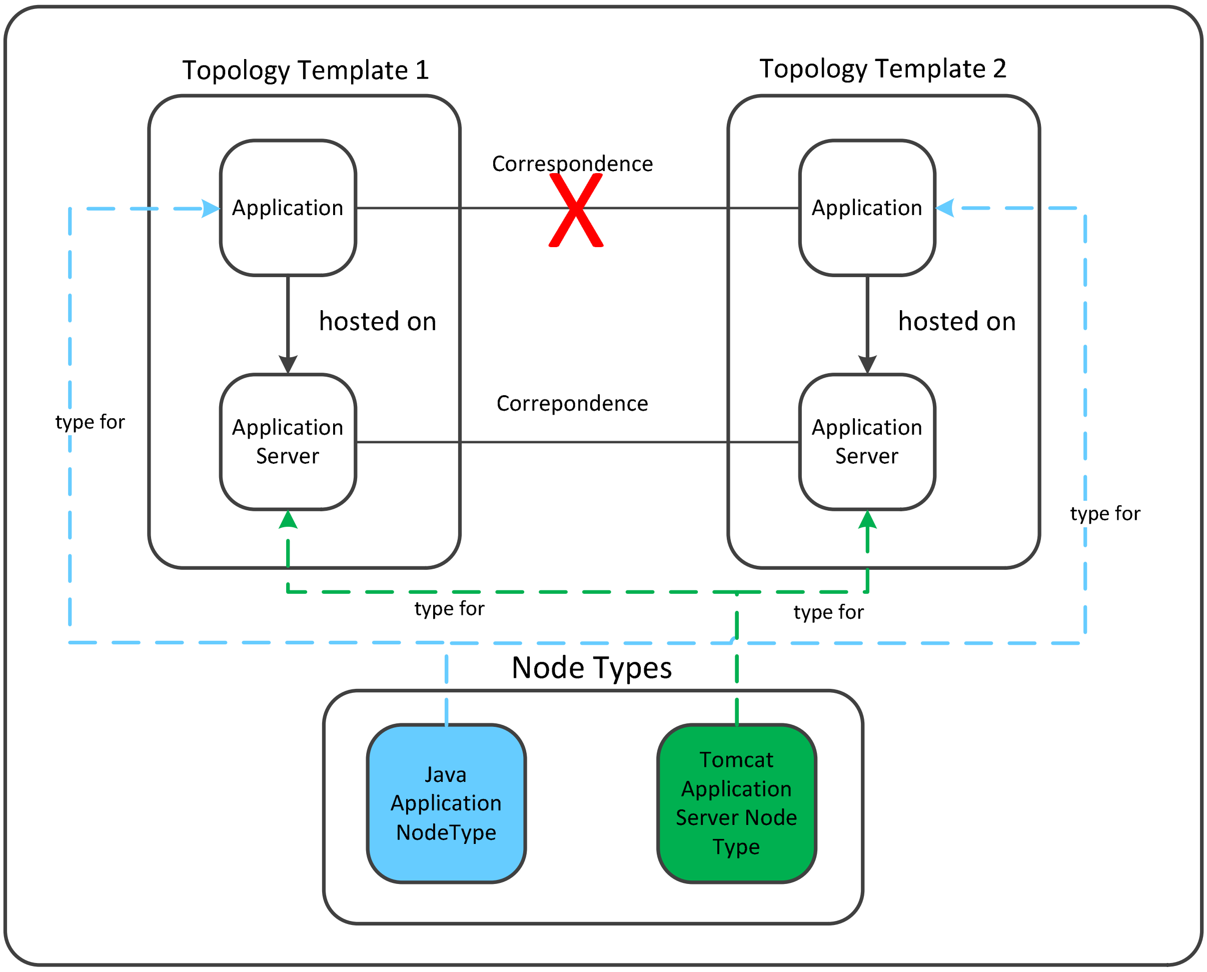


Fig. 5.3: Example of an Application Node Template

Correspondences to More Than one Node Template

Another matching situation to be discussed occurs when a Node Template matches more than one Node Template inside one Topology Template. This must not be confused with 1: n or n:1 mapping characteristics. Those characteristics imply that an entity is mapped to more than one other entity in order to represent the same semantics. Fig. 4.2 in Section 4.1 exemplifies such a situation. In contrast the concept discussed here is the occurrence of several Node Templates the have identical Node Types. Fig. 5.4 shows that the *OS* Node Template the *database* is hosted on matches both *OS* Node Templates of the two *Application Servers*. Therefore, two Correspondences have been established. Each Node Template at which the comparison started, must store the found Node Template Correspondence additionally to the mapping. We will see in Section 6.1.1 that this is a necessary requirement for the merging algorithm in order to generate a correct result under certain, yet to be discussed, circumstances. If each Node Template has a Correspondence to every other Node Template, which comes into consideration because of the appropriate Node Type, the author calls this a *full mapping* between the respective Node Templates. Otherwise it is called a *partial mapping* in this thesis.

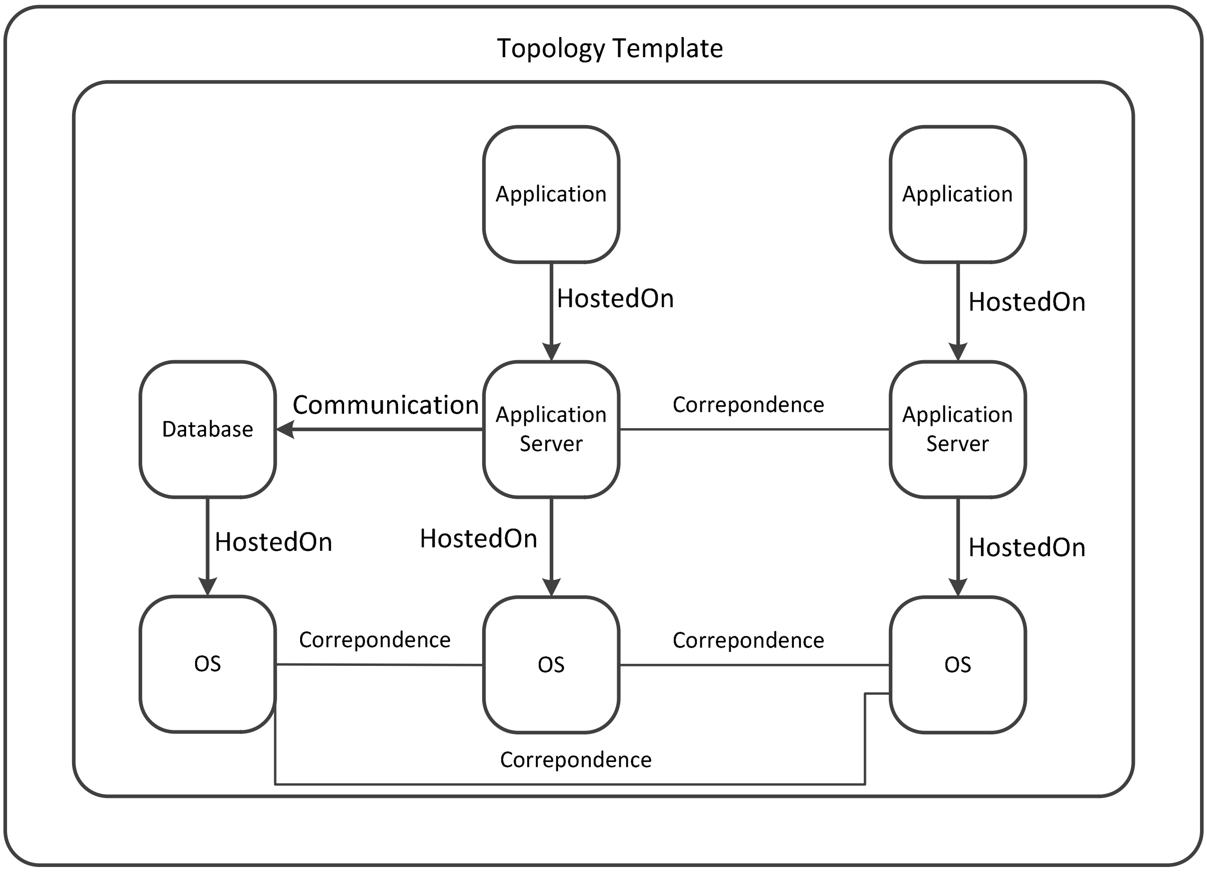


Fig. 5.4: Correspondences to more than one Node Template

Fig. 5.4 also shows Topology Templates containing disconnected graphs, i.e. the graphs with more than one component can be handled by the matching concept.

Consideration of Derived Node Types

The different cases discussed above assume that the Node Types must possess an identical *id* and *targetNamespace* attribute in order to even take into consideration the matching of two Node Templates. However, the concept of matching can be expanded to Node Types that are not identical but have some kind of relation with each other in a Node Type inheritance tree. Fig. 5.5 shows the relationship between different application server Node Types. In this example a *Websphere Application Server (WAS) 6.1* with *Web Service Feature Pack* is derived from *Websphere Application Server 6.1* which itself is derived from a generic Application Server Node Type. The *DerivedFrom* attribute indicates the respective Node Type that is extended. The relationships between the Node Types in Fig. 5.5 form an "inverted" tree with the arcs pointing in the direction of the root Node Type *(Generic) Application Server.*

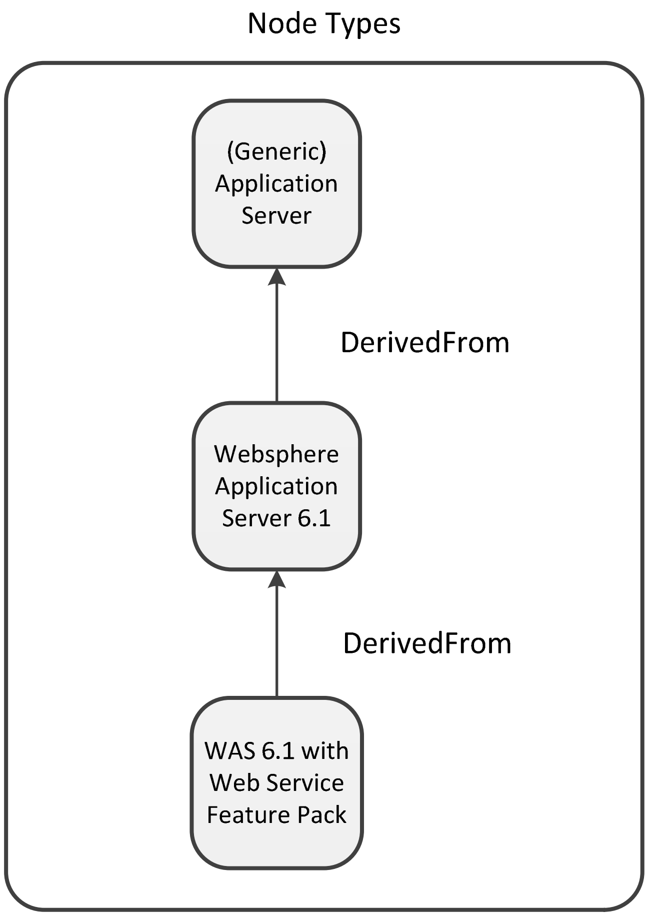

Node Types
(Generic) Application Server
DerivedFrom
Websphere Application Server 6.1
DerivedFrom
WAS 6.1 with Web Service Feature Pack

Fig. 5.5: Node Type parent relationships

A similar case is depicted by Fig. 5.6. Two application server Node Types from different software vendors share the same root Node Type and thus are siblings in the inheritance tree.

To match Node Templates using node Types that are not identical but are related with each other as indicated by the inheritance tree, the author of this work proposes an extension of the TOSCAMerge framework discussed above with its generic part and the type-specific plugins. The generic part must recursively resolve the actual properties of the Node Type under evaluation. That means if a Node Type used by a Node Template has a filled *De-*

rivedFrom attribute, the framework will retrieve the corresponding Node Type and traverse the inheritance tree recursively up to the root Node Type and calculate the effective overall NodeType properties. According to the TOSCA specification the properties and operations of the basis Node Type either form a union or are overridden by the derived Node Type in case they share the same name. Exceptions are the *InstanceState* elements which are combined to one single Instance State if their Instance state names are identical.

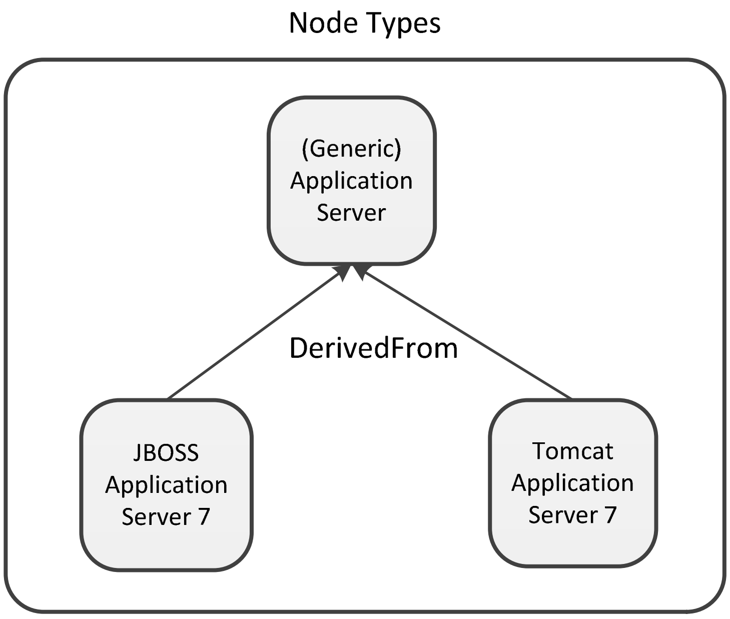

Node Types
(Generic) Application Server
DerivedFrom
JBOSS Application Server 7
Tomcat Application Server 7

Fig. 5.6: Node Type sibling relationship

Type-specific plugins that are invoked subject to the Node Types qualified name are provided with yet another qualified name. Therefore, each type-specific plugin is understood as a matcher between two Node Templates, using Node Types identified by their qualified name (Node Type *id* and *targetNamespace*). If the Node Types are identical, the framework calls the corresponding matcher for the identical qualified names. If the Node Types are not identical, the framework looks up its type-specific configuration and calls the Node Template matcher that has specified both qualified names. If no suitable matcher is found, the framework assumes that the Node Templates under evaluation and their corresponding Node Types are not compatible and do not match. This follows the closed world assumption approach mentioned in Section 4.2. Missing matchers for two qualified Note Type names can easily be added to the TOSCAMerge framework (see Section 8.3.2). To subsume the previous section, the framework must be able to handle Node Templates with two identical Node Types, regardless if they are derived or not, Node Templates with Node Types being at different levels in an inheritance hierarchy and Node Templates with Node Types that are siblings in the inheritance hierarchy.

Basic Algorithm for Finding a Mapping Between Node Templates

After having discussed the basic matching case and different extensions thereof, a basic high level algorithm to find a mapping is proposed in the following section. Listing 5.1 and Listing 5.2 show the function *findMapping* used to find a mapping between two Topology Templates. The function has the following three input parameters: Two Topology Templates, denoted by TT_1 and TT_2, as well as an empty set M. The function's output is the possibly empty set M, holding the determined Correspondences. The algorithm iterates over all Node Templates in TT_1 and conducts a first evaluation if the current Node Template's Node Type is of a prohibited type such as an application. If so, the processing of the current Node Template can be skipped (line 7-9). In the next step, a nested loop visits all Node Templates in Topology Template TT_2. Here as well, the Node Type of the current Node Template n_2 is evaluated against a list of prohibited types and possibly not processed any further (line 12-14). Subsequently, the Node Types $nt_1$and nt_2 of the current Node Templates n_1 respectively n_2 are retrieved for a more convenient use thereafter (line 16 and 17).

To proceed to the actual comparison of the current two Node Templates, the algorithm evaluates if the retrieved Node Types are either identical, i.e. their namespaces and ids are equal, or if nt_1 is derived from nt_2 or vice versa, or if both have a common ancestor Node Type. The derived-from-check is done via subroutine-call of the function *isDerivedFrom* which is shown in Listing 5.3 and explained in detail further below, whereas the evaluation if both Node Types have a common ancestor is conducted in the function *hasCommonAncestor* which is not shown here due to space limitations. Only if one of the four conjunctions evaluates to true, the processing of the two current Node Templates may proceed (line 19). In order to match n_1 and n_2, in line 24 a Node Template Matcher is created, subject to their Node Types nt_1 and nt_2. The actual match of the Node Templates properties is executed by the call of *match* function located in the previously created matcher (line 26). It provides Node-Type-specific functionality that is discussed in detail in the next section. If the *match* function evaluates to true, the Relationship Templates incident to n_1 and n_2 are matched using a construct denoted by *RelationshipTemplateMatchingHandler*. In doing so, for each relevant Relationship Type a different handler is created and executed by invoking a function called *handleRelationshipTemplates*. The function returns a possibly empty set of so-called Relationship Template Correspondences in each case. The relevant pseudo code can be seen in the lines 28-32 and is listed here to complete the basic matching algorithm. However, it is reviewed again in Section 5.2 when a detailed analysis of all matching cases between Relationship Templates follows.

The rest of the *findMapping* function in line 34-39 comprises the following: a Node Template Correspondence, denoted by c is created from n_1 to n_2 and the overall set of Relationship Template Correspondences between incident Relationship Templates of the current Node Templates is added to c. c itself is unified with the set of already existing Node Template Correspondences, i.e. the mapping M. Finally c is also added to n_1 since the comparison started at this Node Template and, as already mentioned, the information is needed for the merging algorithm proposed in Section 6.1.1.

```
Basic algorithm for finding a mapping between Node Templates 1
Input: Topology Template $TT_1$
Input: Topology Template $TT_2$
Input: Mapping M = ∅
Output: Mapping M
findMapping($TT_1$,$TT_2$,M)
  **for each** Node Template $n_1 \in TT_1$ **do**
    **if** Node Type of $n_1 \in$ NotAllowedTypes **then**
      **continue**
    **end if**

    **for each** Node Template $n_2 \in TT_2$ **do**
      **if** Node Type of $n_2 \in$ NotAllowedTypes **then**
        **continue**
      **end if**

      Node Type $nt_1$ = Node Type of $n_1$
      Node Type $nt_2$ = Node Type of $n_2$

      **if** $nt_1 == nt_2$
      $\vee$ isDerivedFrom($nt_1$,$nt_2$)
      $\vee$ isDerivedFrom($nt_2$,$nt_1$)
      $\vee$ hasCommonAncestor($nt_1$,$nt_2$)
      **then**
        Matcher matcher = createNodeTemplateMatcher($nt_1$,$nt_2$)
          ...
```

Listing 5.1: Function findMapping part 1

```
Basic algorithm for finding a mapping between Node Templates 2
          ...
        **if** matcher.match($n_1$,$n_2$) **then**

          Set H = createRelationshipTemplateMatchingHandlers()
          Set of Rel Template Correspondences RC = new Set()
          **for each** Handler h $\in$ H **do**
            RC.add(h.handleRelationshipTemplates($TT_1$,$TT_2$,$n_1$,$n_2$,M))
          **end for each**

          Node Template Correspondence c = new Node Template
          Correspondence($n_1$,$n_2$)

          c.addRelationshipCorrespondences(RC)
          M.add(c)
          $n_1$.addToCorrespondences(c)
        **end if**
      **end if**
    **end for each**
  **end for each**
**end**
```

Listing 5.2: Function findMapping part 2

Listing 5.3 shows the aforementioned algorithm to determine if two Node Types are related to each other. The function *isDerivedFrom* has two input parameters, namely two Node Types denoted by nt_1 and nt_2. A Boolean value is returned to indicate the relation of the two Node Types, i.e. more specifically whether nt_1 is a descendant of nt_2 and, therefore, derived from the latter. The algorithm first evaluates in line 5 if Node Type nt_1 is derived from any other Node Type. If not, the algorithm returns false and terminates, as nt_1cannot be derived from nt_2. If nt_1 has a parent Node Type, it is retrieved as $nt_{derived}$ (line 6) and compared with nt_2. If both are identical, i.e. have identical qualified names, the algorithm returns true and terminates. Otherwise it is invoked recursively with $nt_{derived}$ and nt_2.as input parameter determining if $nt_{derived}$ is a descendant of nt_2.

```
Algorithm to check if two Node Types are related to each other
Input: Node Type $nt_1$
Input: Node Type $nt_2$
Output: Boolean
isDerivedFrom($nt_1$,$nt_2$)
  if $nt_1$.getDerivedFrom() $\neq$ null then
    Node Type $nt_{derived}$ = $nt_1$.getDerivedFrom()
    if $nt_{derived}$ == $nt_2$ then
      return true
    else
      return isDerivedFrom($nt_{derived}$,$nt_2$)
    end if
  end if
  return false
end
```

Listing 5.3: Function isDerivedFrom

5.1.2 Matching of Node Template Properties

Analysis of the Different Node Template Properties

Further above it was stated that Node Templates match if their Node Types are identical. This concept must be expanded by the inclusion of the different properties that are defined by the TOSCA specification. It introduces a set of properties that are placed on the Node Template level and complement or override the properties of the corresponding Node Types. The properties that override or complement are *MinInstances*, *MaxInstances*, *PropertyDefaults*, *PropertyConstraints*, *Policies*, *EnvironmentConstraints*, *DeploymentArtifacts* and *ImplementationArtifacts*. Subsequently, we will discuss the different properties and illustrate how they relate to the matching of Node Templates. In doing so, the *property preservation* requirement of Section 4.2 is followed. Furthermore, it will be investigated whether the aforementioned properties can be matched generically or not. If not, type-specific matching procedures must be added as plugins to the generic framework. Only if all matching procedures being part of the generic framework or being a plugin signal a matching of their properties, a Correspondence between the two Node Templates under investigation can be established.

MinInstances and maxInstances: The MinInstances property specifies the minimum number of instances that have to be started when a particular Node Template is instantiated. The maxInstances property specifies the maximum number thereof. The two properties must be unified when two Node Templates are merged, but for matching they are ignored since they have no influence on the matching decision.

PropertyDefaults: The PropertyDefaults of a Node Template contain the initial values of the Node Type properties of a specific Node Template. When matching two Node Templates it must be ensured that (1) the same XML instance documents defined by schema elements in the *Types* section of the Service Template are used and (2) the initial values of the two Node Templates do not conflict with each other. (1) can be ensured by validating the content of both *PropertyDefaults* elements against the schema in the *Types* section. Although the TOSCA specification does not demand all initial values to be present in a non-instantiated service they should be present in order match them appropriately. (2) cannot be decided generically as it is impossible to understand the exact semantics of each XML element in a generic way without using inexact matching techniques (see assumption 3). To overcome this problem the use of type-specific concepts that verify if the initial values of particular Node Type Properties in a Node Template do not contradict each other is proposed. The type-specific matching concepts will be plugged in the generic framework and executed whenever Node Templates using a corresponding Node Type are to be matched. Listing 5.4 illustrates a *PropertyDefaults* example of a generic application server.

```
PropertyDefaults example
<tosca:PropertyDefaults>
  <AppServerProperties>
    <HostName>cs052276.myhost.com</HostName>
    <IPAddress>10.171.50.26</IPAddress>
    <HeapSize>256</HeapSize>
    <SoapPort>8080</SoapPort>
  </AppServerProperties>
</tosca:PropertyDefaults>
```

Listing 5.4: PropertyDefaults example of a generic application server

The type-specific validation of the initial values is strongly related with the *PropertyConstraints* that will be discussed below. Conflicting initial values will mostly be discovered by means of conflicting constraints set on them.

PropertyConstraints: The *PropertyConstraints* define constraints on the initial values of the *PropertyDefaults* element.

```
PropertyContraints example
<tosca:PropertyConstraints>
  <tosca:PropertyConstraint
  constraintType="http://www.example.com/PropertyConstraints/Unique"
    property="/AppServerProperties/HostName">
    <scope>service</scope>
  </tosca:PropertyConstraint>
</tosca:PropertyConstraints>
```

Listing 5.5: PropertyConstraints example of a generic application server

The constraints are defined by URIs and XPath expressions pointing to a particular element that is constrained. Listing 5.5 shows an example of a PropertyConstraint set on the PropertyDefaults of Listing 5.4. The attribute *constraintType* indicates the semantics and the format of the constraint. The attribute *property* points to the element that is constrained, i.e. the *HostName* element in the *AppServerProperties* element. The constraint in this example demands the uniqueness of the *HostName,* the *scope* element is a self-defined, constraint-specific element that specifies that the scope of the uniqueness has to be service-wide.

Similar to the PropertyDefaults the semantics of the PropertyConstraints cannot be determined generically without violating the assumptions made in this work. Therefore, type-specific evaluations of the XPath expressions in the *property* attribute and the semantics of the *constraintType* URI have to be conducted to make sure that two Node Templates to be matched do not contain any conflicting constraints. Additionally, constraint specific elements nested in every *PropertyConstraint* element, such as the *scope* element in Listing 5.5 are also subject to type-specific evaluations and cannot be matched generically. Note that two Node Templates match if one or both of them do not possess any PropertyConstraints. This will be evaluated generically by the framework.

Policies: Another element of the Node Template that has to be considered is the *Policies* element. Nested inside the element *Policies, Policy* elements carry the information about management practices concerning a particular Node Template. When matching two Node Templates it must be evaluated if their Policies do not contradict each other. Policies can be found on the Node Template level and on the Node Type level. The TOSCA specification defines that Polices on the Node Type level are unified with those on the Node Template level. In case of identical *name* and *type* attribute values, the Policy on the Node Template overrides the one defined inside the Node Type. The proposed framework in this thesis will be able to calculate the "effective" policies similar to the WS-Policy framework [51]. Effective in this context means that, given two Node Templates, for each one a list of Policies consisting of Node Type Polices and Node Template Polices is calculated by examining all Policies and evaluating which ones are not overridden. The lists are then passed to a type-specific implementation that handles the semantic evaluation of the Policies and their potential mutual exclusion. Particularly the Policy-specific content that is not part of the TOSCA specification will be evaluated. Note that two Node Templates match if one or both contain no effective Policies. This will be evaluated by the TOSCAMerge framework.

EnvironmentConstraints: The matching of *EnvironmentConstraint* elements consisting of the *constraintType* attribute and an EnvironmentConstraint-specific content is also an instance for type-specific matching. By using only exact matching techniques in the proposed framework, the semantics of the *constraintType* attribute and the nested specific non-TOSCA-defined content cannot be evaluated. A plugin to the framework must decide whether the constraints that one Node Template defines for its runtime environment, such as particular security settings, are matched by the other Node Template or not. Note that the absence of EnvironmentConstraints in one or both Node Templates indicates that both Node Templates match with regard to their EnvironmentConstraints. This will be evaluated by the TOSCAMerge framework.

DeploymentArtifacts: Deployment Artifacts describe concrete artifacts that are needed to implement a physical component of an IT Service, e.g. a virtual image. The *type* attribute of a *DeploymentArtifact* element could contain the URI http://docs.oasis-open.org/tosca/ns/2011/12/deploymentArtifacts/ovfRef indicating that the DeploymentArtifact describes an OVF [14] package, while the nested type-specific body contains XML fragments referencing that package and mapping Service Template data and elements to the OVF format. Similar to the Policies of a Node Template the effective set of DeploymentArtifacts must be calculated as Deployment Artifacts on the Node Template level override those in the used Node Type provided that the attributes *name* and *type* are identical. The calculation of the effective set of Deployment Artifacts will be done by the framework, the actual matching and the evaluation of the type-specific content is done by a plugin to the framework. The absence of DeploymentArtifacts at one or both Node Templates implies a positive matching of the Node Templates.

ImplementationArtifacts: The last properties of a Node Template that need to be discussed are the *ImplementationArtifacts*. They depict the implementations of the operations defined in the corresponding Node Type. For example a REST operation could be implemented by a Java Servlet. Again, there exist ImplementationArtifacts on Node Type and Node Template level and the calculation of the effective set of artifacts is done by the framework based on the values of the *operationName* and *type* attribute. A type-specific matcher to the framework must be executed to evaluate the semantics of the artifacts and how their embedded *RequiredContainerCapabilities* elements and artifact-specific content fit together. Note that analogous to the aforementioned properties the absence of ImplementationArtifacts in one or both Node Templates under evaluation means that the matching for this property category evaluates to true.

Node Type InstanceStates: The optional *InstanceStates* element is not located on the Node Template level but inside a Node Type. Nevertheless, this property must be analyzed if the Node Types of two particular Node Templates are not identical but related. It contains *InstanceState* elements, indicating runtime states via a *state* attribute holding an URI. The framework must determine all *InstanceState* elements of the derived Node Types and can terminate the comparison if one of the Node Types has no *InstanceState* elements at all. Otherwise it must call a plugin, specific to the two non-identical Node Types, which compares the InstanceStates.

Node Type Interfaces: The element *Interfaces* is also part of a Node Type and not of a Node Template. In case of a matching of two non-identical but related Node Types the nested *Interface* elements and their nested *Operation* elements must be compared. The determination of the effective set of Interfaces with regard to Node Types that are derived from others can be done generically by the framework. However, the actual matching of the Interface elements must be conducted by a type-specific plugin.

Algorithm for Matching Node Template Properties

In the following section a algorithm is proposed for matching the properties of two Node Templates. Furthermore, one of the subroutines will be reviewed exemplarily and the previously introduced concept of calling type-specific plugins will be exemplified.

Listing 5.6 depicts the function *match* that unites several subroutines for the individual Node Template properties explained above. The function requires two Node Templates as input parameters, denoted by n_1 and n_2. A Boolean value is returned indicating the result of the matching of n_1 and n_2. The high-level function returns true if and only if all subroutines evaluate to true.

```
Algorithm for matching the properties of two Node Templates
Input: Node Template $n_1$
Input: Node Template $n_2$
Output: Boolean
match($n_1$,$n_2$)
  if matchPropertyDefaults($n_1$,$n_2$)
    $\wedge$ matchPropertyConstraints($n_1$,$n_2$)
    $\wedge$ matchPolicies($n_1$,$n_2$)
    $\wedge$ matchEnvironmentConstraints($n_1$,$n_2$)
    $\wedge$ matchDeploymentArtifacts($n_1$,$n_2$)
    $\wedge$ matchImplementationArtifacts($n_1$,$n_2$)
    $\wedge$ matchNodeTypeInstanceStates(Node Types of $n_1$,$n_2$)
    $\wedge$ matchNodeTypeInterfaces(Node Types of $n_1$,$n_2$) then
      return true
  else
    return false
  end if
end
```

Listing 5.6: Function match for Node Templates

The review of all the introduced subroutines would go beyond the scope of this document; therefore, one substantial subroutine, namely the *matchPolicies* function is discussed next. It implements the aforementioned concept of calculating the effective set of Policies and, therefore, has the task to generically collect all *Policy* elements attached to the two Node Templates under consideration and pass them to the Node-Type-specific plugin that has the domain-specific knowledge to handle the policies appropriately. In addition the policies attached directly to the Node Templates, all policies of the Node Types and possibly their ancestors in the inheritance hierarchy must be collected respecting the override behavior described by the TOSCA specification. Listing 5.7 shows the mentioned *matchPolicies* function. It requires two Node Templates, denoted by n_1 and n_2, as input and returns a Boolean value.

```
Algorithm to match the Policies of two Node Templates
Input: Node Template $n_1$
Input: Node Template $n_2$
Output: Boolean
matchPolicies($n_1$,$n_2$)
  Node Type $nt_1$ = $n_1$.getNodeType()
  Node Type $nt_2$ = $n_2$.getNodeType()
  Set $NodeTypePolicies_1$ = determineDerivedPolicies($nt_1$)
  Set $NodeTypePolicies_2$ = determineDerivedPolicies($nt_2$)

  Set $NTPolicies_1$, $NTPolicies_2$ = new Sets of Policies()
  if $n_1$.getPolicies() $\neq$ null then
    $NTPolicies_1$ = $n_1$.getPolicies()
  end if
  if $n_2$.getPolicies() $\neq$ null then
    $NTPolicies_2$ = $n_2$.getPolicies()
  end if

  if $NodeTypePolicies_1 == \emptyset \wedge NTPolicies_1 == \emptyset$ then
    return true
  end if
  if $NodeTypePolicies_2 == \emptyset \wedge NTPolicies_2 == \emptyset$ then
    return true
  end if
  if $NodeTypePolicies_1 \neq \emptyset$ then
    for each Policy $p_1 \in NodeTypePolicies_1$ do
      if $p_1 \notin NTPolicies_1$ then
        $NTPolicies_1$.add($p_1$)
      end if
    end for each
  end if
  if $NodeTypePolicies_2 \neq \emptyset$ then
    for each Policy $p_2 \in NodeTypePolicies_2$ do
      if $p_2 \notin NTPolicies_2$ then
        $NTPolicies_2$.add($p_2$)
      end if
    end for each
  end if
  return matchPoliciesTypeSpecificContent($NTPolicies_1$, $NTPolicies_2$)
end
```

Listing 5.7: Function matchPolicies

In the first step in line 5 and 6 the Node Types of n_1 and n_2 are retrieved for convenience and stored in variables denoted by nt_1 and nt_2 respectively. Afterwards, for both of the Node Types the Policies and the Policies the particular Node Type is derived from are determined by invoking the subroutine *determineDerivedPolicies* twice (line 7 and 8). The result is stored in two sets: $NodeTypePolicies_1$ and $NodeTypePolicies_2$. The explanation of the algorithm of this subroutine will follow shortly. After the determination of the Node Type Policies the Node Template Policies are retrieved dependent on their existence and stored in the previously declared sets $NTPolicies_1$ and $NTPolicies_2$ (line 10-16). If both of the two sets belonging to n_1 respectively n_2 are empty, the algorithm returns true and terminates. In this case one of the two Node Templates does not have any Policies that could contradict the other ones Policies (line 18-23). Subsequently, the actual unification of Policies follows in the lines 24-30. First it is evaluated if the Node Type policies of n_1 are not empty. If so, each *Policy* element of $NodeTypePolicies_1$ is checked against the Node Template Policy set $NTPolicies_1$ and added to it if it is not yet element of it. As we have seen earlier, this is necessary since Node Template Policies override Node Type Policies with identical *name* and *type* attributes. The same approach is used in line 31-37 for the calculation of the effective policy of Node Template n_2. After having calculated two sets with the effective policies, they are passed as input parameters to the type-specific plugin with the subroutine call *matchPoliciesTypeSpecificContent.* There, a type-specific implementation or an external Policy engine can compare the passed policies.

As addressed earlier, the *determineDerivedPolicies* function can be found in Listing 5.8. It has only one input parameter: the Node Type, denoted by nt of a particular Node Template. The output is a set of Policies. First, a new set $Policies$ is created that can hold the Policies (line 4). If nt is derived from another Node Type, this Node Type is retrieved and stored in a variable denoted by $nt_{derived}$. Subsequently, the function is invoked recursively with $nt_{derived}$ as input parameter and, thereby, follows the inheritance hierarchy. The returned Policies are added to $Policies$ (line 6-9). If the current Node Type nt has attached policies, they are retrieved and each *Policy* element is compared against the overall $Policies$ set. If a *Policy* element p is already element of $Policies$, i.e. it has the same *name* and *type* attributes and has been added on a hierarchy level closer to the root, the element is removed from $Policies$ and the current p is added to the list. If p is not yet element of $Policies$, it is added immediately (line 11-21).

```
Algorithm to determine all Policies along a Node Type hierarchy
Input: Node Type nt
Output: Set of Policies
determineDerivedPolicies(nt)
  Set Policies = new Set of Policies()

  if nt.getDerivedFrom() ≠ null then
    Node Type $nt_{derived}$ = nt. getDerivedFrom()
    Policies = Policies.addAll(determineDerivedPolicies($nt_{derived}$))
  end if

  if nt.getPolicies() ≠ null then
    List NTPolicies = nt.getPolicies()
    for each Policy p ∈ NTPolicies do
      if p ∈ Policies then
        Policies.remove(p)
        Policies.add(p)
      else
        Policies.add(p)
      end if
    end for each
  end if
  return Policies
end
```

Listing 5.8: Function determineDerivedPolicies

5.2 Matching of Relationship Templates

When discussing the basic Node Template matching case, it was assumed that the Relationship Templates between the Node Templates only represented HostedOn semantics. Now, Relationship Templates with additional properties located in the *PropertyDefaults* element, additional constraints on these properties (*PropertyConstraint* elements) and constraints on the use of a Relationship Template itself, denoted by *RelationshipConstraint* elements, are studied. Furthermore, the relationship semantics of HostedOn and others (see Fig. 4.4 for the Relationship Type tree) have to be considered. For each of the mentioned properties it will be investigated whether they can be matched generically or not. If not, type-specific matching procedures must be added as plugins to the generic TOSCAMerge framework. Only if all generic Relationship Template matching procedures of TOSCAMerge or the responsible plugins signal a matching of the properties, a Correspondence between the two Relationship Templates under investigation can be established. The type-specific plugins are identified by their qualified name. The TOSCAMerge framework prototype will provide the implementation for three typical Relationship Types: *HostedOn*, *Communication* and *Dependency*. Additional Relationship Types can easily be added to as plugins to the framework (see Section 8.3.2).

Definition 5.3 (Relationship Template Correspondence): *A Relationship Template Correspondence is defined as an indicator that two particular Relationship Templates have non-contradictory PropertyDefaults, PropertyConstraints and RelationshipConstraints and can be unified in a subsequent merging step. The Relationship Template Correspondence is attached to a Node Template Correspondence. The Relationship Template that marks the source of the Correspondence also stores all found Correspondences to other Relationship Templates.*

The Relationship Template Correspondences can be seen as overlay edges between Relationship Templates that in this case do not represent edges but nodes in an overlay graph over the TOSCA graph.

The matching cases analyzed below not only serve to decide which incident Relationship Templates to two particular Node Templates can be unified but also to indicate in which cases the matching of Node Templates must be skipped to avoid invalid semantics.

5.2.1 Analysis of the Different Relationship Template Matching Cases

HostedOn Semantics

After all the generic matchers and matcher plugins of the TOSCAMerge framework have decided that two Node Templates match, their relationships to other Node Templates have to be evaluated. As already mentioned, each Node Template must only be source of one *HostedOn* Relationship Template. So if we look at Fig. 5.7, a correspondence between the two *Application Server* Node Templates could be established assuming all the matchers return true. But matching both of the (blue) *HostedOn*-Relationship Templates reveals that the Relationship Templates Properties are incompatible with each other. Merging both *Application Server* Node Templates would lead to the crossed out Topology Template with the undesired semantics of an instance of an application server installed on two different operating systems at the same time. Therefore, the underlying matching algorithm must make sure that no Node Template Correspondence is established between two Node Templates if both are the source, or speaking in terms of graph theory the tail, of HostedOn-Relationship Templates, which Properties are marked incompatible by the Relationship Template generic matchers and plugins of the TOSCAMerge framework. This is stipulated by the *semantically correct result* requirement of Section 4.2.

If the two HostedOn Relationship Templates have non-contradictory properties and constraints, a Relationship Template Correspondence can be created and attached to the Node Template Correspondence.

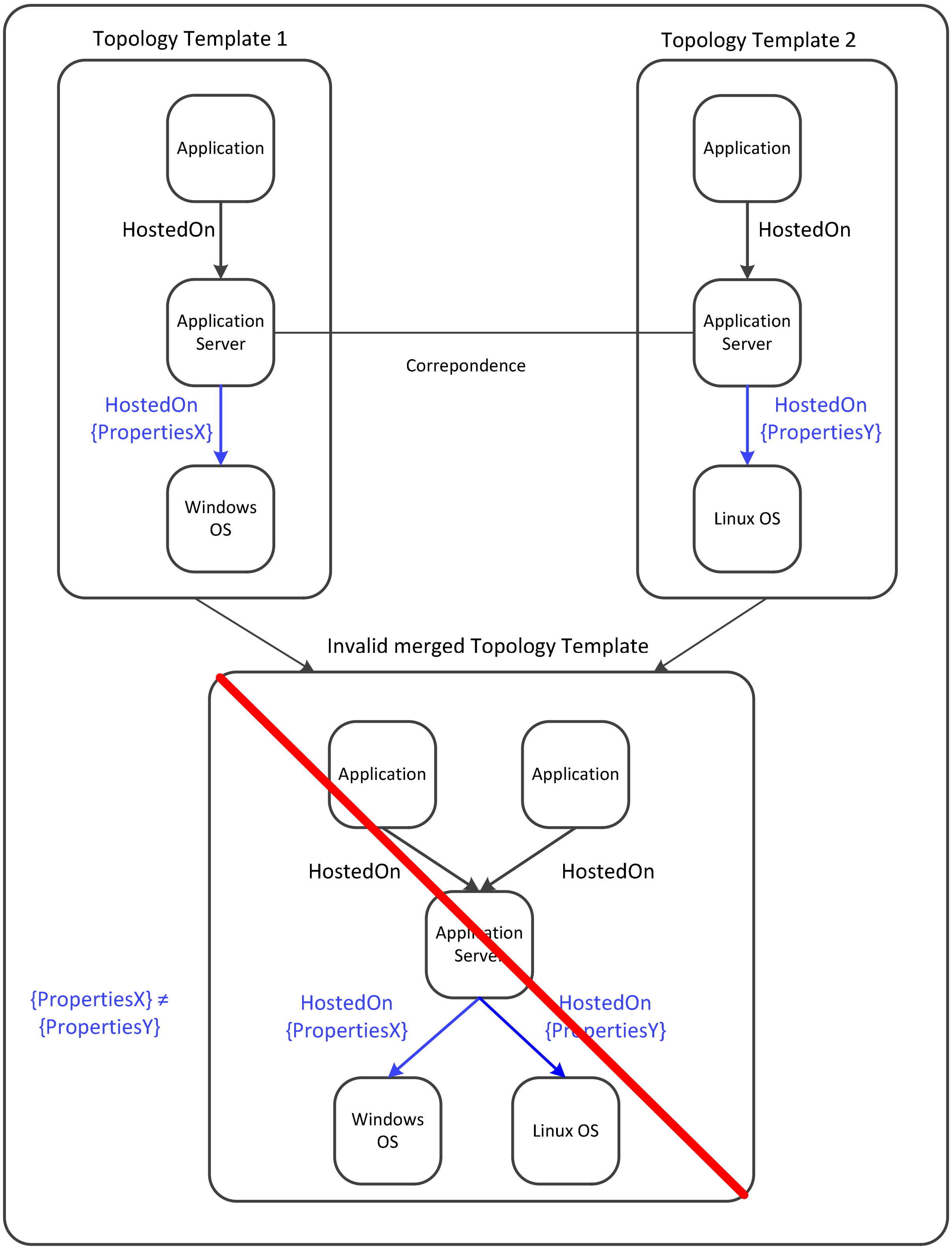


Fig. 5.7: Possible, but invalid correspondence leads to undesired semantics

Communication Semantics

Another Relationship Type that has to be discussed in this work is the *Communication* Relationship Type. It describes the communication between two Node Templates and the associated properties and constraints. In contrast to Relationship Templates with HostedOn semantics, it is valid that several Relationship Templates with Communication semantics between two Node Templates exist. However, it is preferable that Communication-Relationship Templates that have non-contradictory properties and constraints are unified.

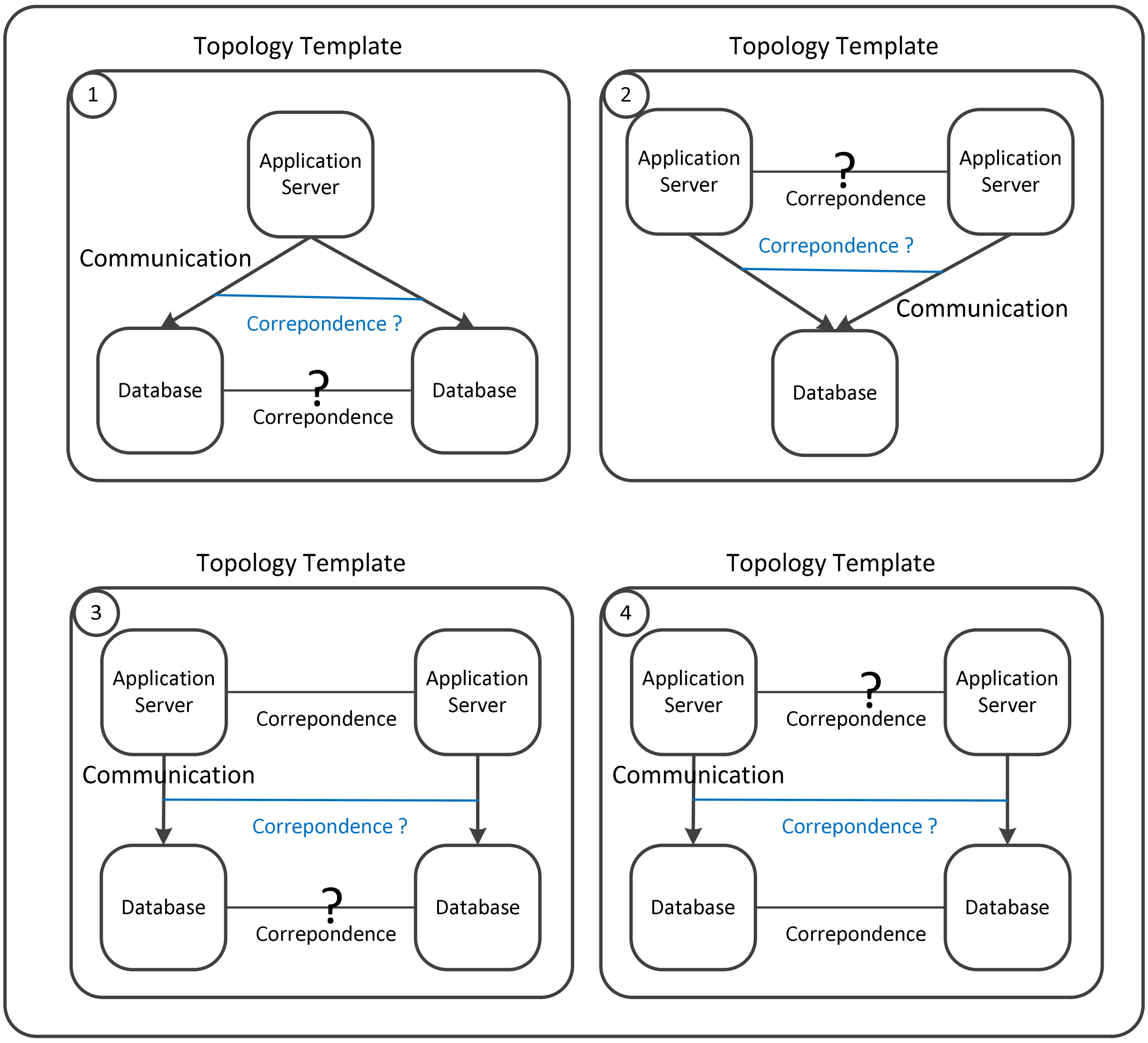


Fig. 5.8: Four matching cases of Relationship Templates with Communication semantics

Fig. 5.8 shows the four main matching cases of Relationship Templates that possess a Communication Relationship Type. Case 1 is a single Node Template as source for two Communication Relationship Templates that target two other Node Templates that are currently evaluated for matching. If the TOSCAMerge framework and the type-specific Communication Relationship Type plugin decide that the Relationship Templates match, a Relationship Template Correspondence is established. Case 2 shows the similar constellation where the two Node Templates under examination are the sources of the Communication Relationship Templates instead of the targets such as in case 1. The cases 3 and 4 depicted in Fig. 5.8 can be encountered if there is already an Node Template Correspondence either between both of the source Node Templates (case 3) or both of the target Node Templates (case 4). The already existing Node Template Correspondence has been found in an earlier iteration of the matching algorithm. Note that the algorithm should also be able to find Relationship Template Correspondences between Communication Relationship Templates that have the same source and target. Furthermore, Correspondences from one Relationship Template to more than one other Relationship Template of the same Relationship Type may occur.

Dependency Semantics

The last prototypical implemented Relationship Type is the *Dependency* Relationship Type. This Relationship Type has the semantics of a dependency of the source Note Template to the target Node Template, e.g. the Application Server needs the Database up and running before being able to start.

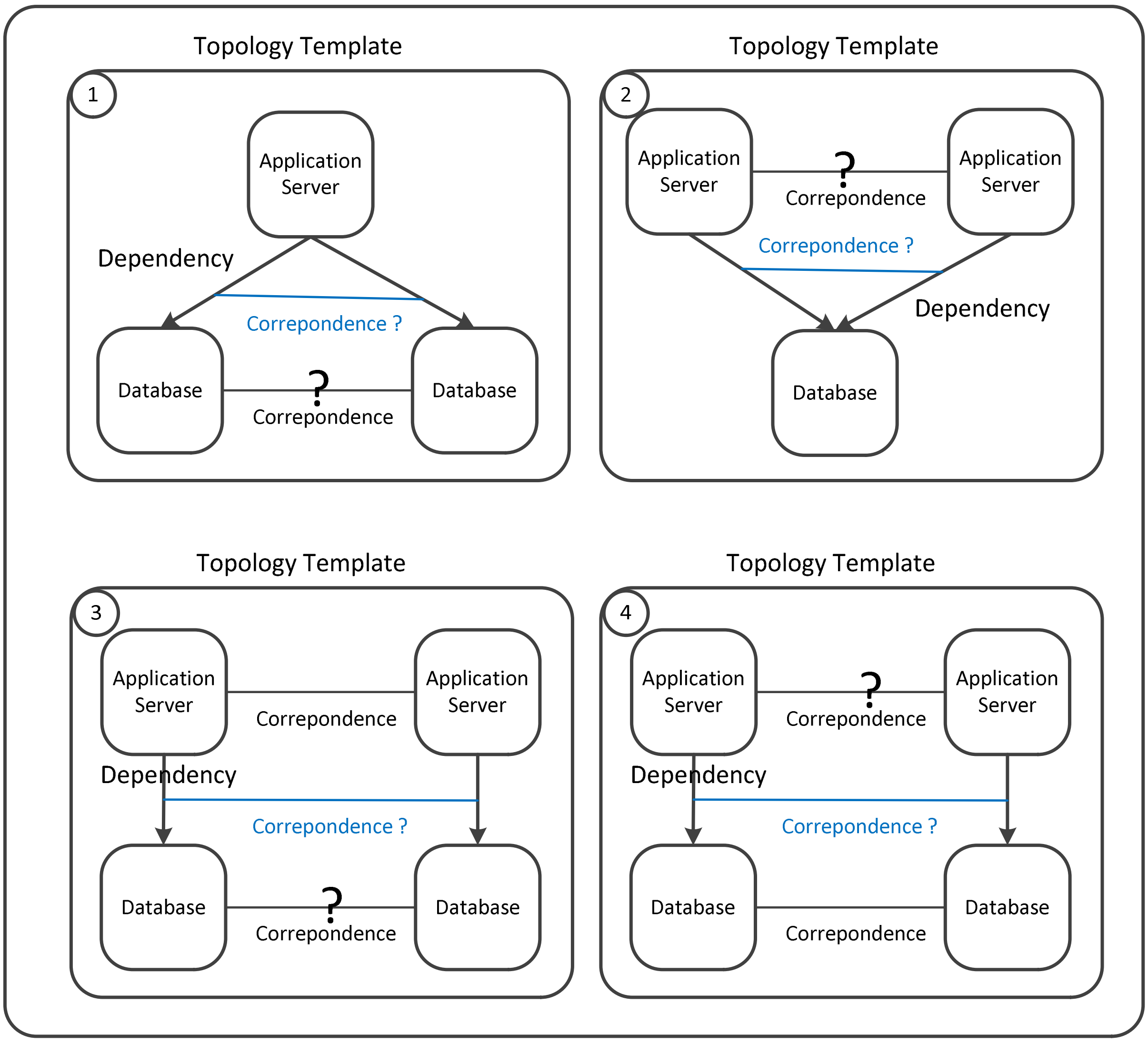


Fig. 5.9: Four matching cases of Relationship Templates with Dependency semantics

The four cases depicted in Fig. 5.9 exhibit the same Node Template and Node Template Correspondence constellations as discussed previously in the context of the Communication

Relationship Templates. However, there is a one substantial difference between the two Relationship Types. If two particular Relationship Templates do not match this does not only imply that there cannot be a Relationship Template Correspondence but also that a Node Template Correspondence must not be created in order to prevent a merged Topology Template with invalid semantics and to adhere to the *semantically correct result* requirement.

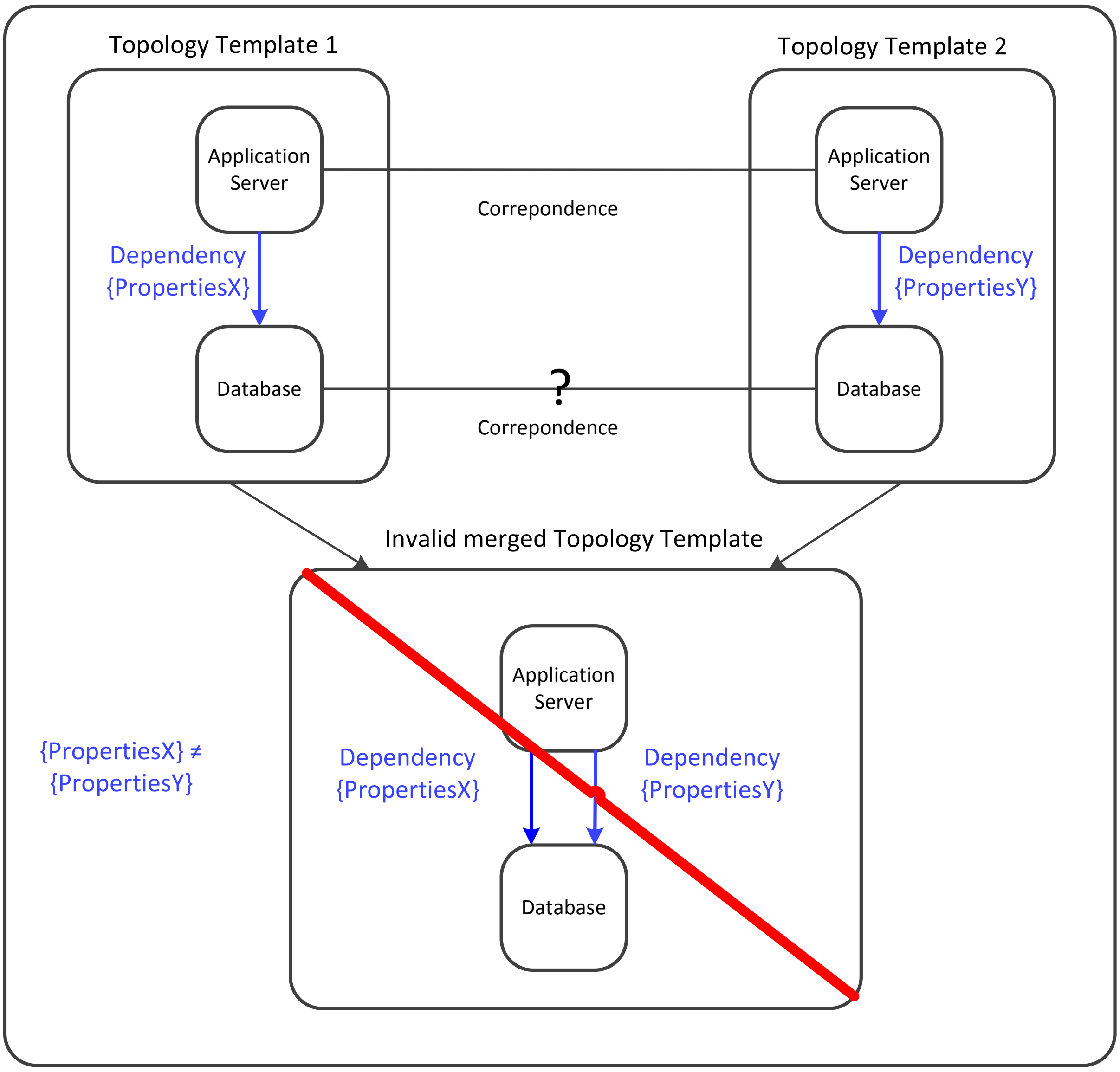


Fig. 5.10: Example of invalid merged Topology Template

Fig. 5.10 shows such a case where there already exists a Node Template Correspondence between two *Application Server* Node Templates and a Correspondence between the two *Database* Nodes is under evaluation. The algorithm must not allow the Node Template Correspondence as the properties of the Relationship Templates are contradictory in their content and a merge would lead to the depicted invalid structure where the merged Application Server Node Template is dependent of the merged Database Node Template in a contradicting way. Note that Correspondences from one Relationship Template to more than one other Relationship Template of the same Relationship Type may occur.

Algorithm for Matching HostedOn Relationship Templates

The following section discusses one algorithm representative for the Relationship Template matching cases analyzed above. The algorithm conducts the matching of two Relationship Templates with HostedOn semantics. Presenting the algorithms for Communication and Dependency semantics as well would go beyond the scope of this document, however, they are implemented in the TOSCAMerge framework. The proposed algorithm in Listing 5.9 is embedded in the function *handleRelationshipTemplates*. Revisiting the basic function *findMapping* for finding Node Template Correspondences in Listing 5.1, a concept named *RelationshipTemplateMatchingHandler* was introduced briefly. A RelationshipTemplateMatchingHandler offers a function named *handleRelationshipTemplates* that is capable of handling Relationship Templates of a certain Relationship Type. It can be seen as another plugin to the TOSCAMerge framework that implements Relationship-Type-specific high-level matching algorithms. In the *findMapping* function in line 31 the *handleRelationshipTemplates* subroutine of all available handlers are invoked. Listing 5.9 depicts the function for the HostedOn RelationshipTemplateMatchingHandler responsible for the HostedOn Relationship Type. The function has five input parameters: two Topology Templates, denoted by TT_1 and TT_2, holding all the Relationship Templates, two already positively matched Node Templates, denoted by n_1 and n_2, and a set M that holds the so far created Node Template Correspondences. M is not needed for the matching HostedOn Relationship Templates, however, as the signature of the *handleRelationshipTemplates* function is defined uniformly for all RelationshipTemplateHandlers specified here. For the matching of Communication and Dependency Relationship Templates M is necessary as seen in the cases 3 and 4 in Fig. 5.8 and Fig. 5.9. The output of the function is a set of Relationship Template Correspondences, although only one Correspondence can be found as only one pair of HostedOn Relationship Templates may exist. But again this is because of the uniform definition of the function's signature.

The first step in line 8-9 is to create two variables denoted by $hostedOn_1$ and $hostedOn_2$. These variables will hold the HostedOn Relationship Templates assuming that the Topology Templates are valid and only one with a particular Node Template as source exists (see Assumption 9). A set C that will hold the found Relationship Templates Correspondence is then created in line 10. Subsequently, each Relationship Template in TT_1 is checked whether it has n_1 as source and has a HostedOn Relationship Type at the same time. If so, the current Relationship Template r_1 is assigned to $hostedOn_1$ and the loop is immediately left (line 12-17).

```
High-level matching algorithm for HostedOn Relationship Templates
 1  Input: Topology Template $TT_1$
 2  Input: Topology Template $TT_2$
 3  Input: Node Template $n_1$
 4  Input: Node Template $n_2$
 5  Input: Mapping M
 6  Output: Set Relationship Template Correspondences
 7  handleRelationshipTemplates($TT_1$,$TT_2$,$n_1$,$n_2$,M)
 8    Relationship Template $hostedOn_1$
 9    Relationship Template $hostedOn_2$
10    Set C = new Set of Relationship Template Correspondences()
11
12    for each Relationship Template $r_1 \in TT_1$ do
13      if $r_1$.getSource == $n_1 \wedge r_1$.getRelationshipType() == “HostedOn” then
14        $hostedOn_1 = r_1$
15        break
16      end if
17    end for each
18
19    for each Relationship Template $r_2 \in TT_1$ do
20      if $r_1$.getSource == $n_2 \wedge r_2$.getRelationshipType() == “HostedOn” then
21        $hostedOn_2 = r_2$
22        break
23      end if
24    end for each
25
26    Matcher matcher = createRelTemplateMatcher(RelationshipType of
27    $hostedOn_1$)
28    if matcher.match($hostedOn_1$,$hostedOn_2$) then
29      Relationship Template Correspondence c = new Relationship Template
30      Correspondence()
31      C.add(c)
32    else
33      throw new NotCompatibleException()
34    end if
35    return C
36  end
```

Listing 5.9: Function handleRelationshipTemplates for HostedOn semantics

The iteration over all Relationship Templates is necessary, as Node Templates do not know their incident Relationship Template. The same steps are conducted over all Relationship Templates in TT_2 (line 19-24). Not shown in Listing 5.9, due to shortage of space, are optimizations that terminate the algorithm if the n_1 or n_2 is not source of a HostedOn Relationship Template.

In line 26 a Relationship Template matcher is created that matches the properties of a Relationship Template. The creation is conducted subject to the HostedOn Relationship Type of $hostedOn_1$ and $hostedOn_2$. If the invocation of the *match* subroutine in line 28 evaluates to true, a new Relationship Template Correspondence is created and added to C and C is returned. If $hostedOn_1$ and $hostedOn_2$ do not match, an exception is thrown in line 33 that indicates that no Node Template Correspondence between n_1 and n_2 must be established. Otherwise it would lead to the invalid semantics discussed above and depicted in Fig. 5.7. The *match* subroutine is explained in the next section when analyzing the matching of Relationship Template Properties.

5.2.2 Matching of Relationship Template Properties

Analysis of the Different Node Template Properties

The following section analyzes the three Relationship Template Properties *PropertyDefaults, PropertyConstraints* and *RelationshipConstraints.*

PropertyDefaults: The *PropertyDefaults* element of a Relationship Template has the same purpose as in a Node Template: initial values for the *RelationshipTypeProperties* attribute of the corresponding Relationship Type are provided to specify individual properties of a Relationship Template, i.e. of a relationship between two Node Templates. The schemata for the RelationshipTypeProperties are located in the *Types* section of the Service Template. Matching the PropertyDefaults of two Relationship Templates is not possible in a generic way without violating Assumption 3: no inexact matching techniques are considered in this work. Therefore, similar to the PropertyDefaults of Node Templates a type-specific plugin to the TOSCAMerge framework is proposed to match the PropertyDefaults of two Relationship Templates. The plugin compares the initial values provided by an XML fragment and determines if the values are compatible with each other. The framework itself provides functionality to check if the PropertyDefaults of both Relationship Templates under evaluation adhere to the corresponding schema. Note that if one Relationship Template does not possess any PropertyDefaults, the framework will decide that both Templates match.

PropertyConstraints: The *PropertyConstraints* element contains nested *PropertyConstraint* elements that define constraints on the initial property values specified by the PropertyDefaults of the corresponding Relationship Type. Whether the contents of two *PropertyConstraints* elements match has to be determined by a type-specific plugin to the framework that has the knowledge of the semantics of every *constraintType* attribute and the optional nested constraint specific XML fragments. The framework itself will provide capabilities to retrieve the concerned XML elements, respectively values, from the *PropertyDefaults* element and hand them over to the type-specific plugin. Note that if one or both Relationship Templates do not possess any PropertyConstraints, the framework will decide that the PropertyConstraints match.

RelationshipConstraints: The *RelationshipConstraints* element contains nested *RelationshipConstraint* elements that define constraints on the use of a Relationship Template. A type-specific plugin to the TOSCAMerge Framework will handle the evaluation whether the semantics of the *constraintType* URI and the nested constraint specific content do contradict each other. Note that if one or both Relationship Templates do not possess any RelationshipConstraints, the framework will decide that the RelationshipConstraints match.

Algorithm for Matching Relationship Template properties

After having analyzed the properties of Relationship Templates and which idiosyncrasies must be considered when matching them, an algorithm is proposed that handles the matching. Similar to the matching of Node Templates, the algorithm invokes a subroutine for each relevant property and evaluates to true if all subroutines on their own evaluate to true. Additionally to the function *match*, one exemplary subroutine will be shown to show the interaction of generic framework algorithms with user specific plugins.

Listing 5.10 depicts the matching algorithm for Relationship Template Properties. It is embedded in the *match* function and requires two Relationship Templates, denoted by r_1 and r_2 as input parameters. The Boolean return value indicates the outcome of the matching.

```
Algorithm to match properties of Relationship Templates
Input: Relationship Template $r_1$
Input: Relationship Template $r_2$
Output: Boolean
match($r_1$,$r_2$)
  if matchPropertyDefaults($r_1$,$r_2$)
    $\wedge$ matchPropertyConstraints($r_1$,$r_2$)
    $\wedge$ matchRelationshipConstraints($r_1$,$r_2$)
    then
      return true
  else
    return false
  end if
end
```

Listing 5.10: Function match for Relationship Templates

The *match* function invokes three subroutines: *matchPropertyDefaults*, *matchPropertyConstraints* and *matchRelationshipConstraints*. Only if all three return true, *match* itself returns true. Otherwise the two Relationship Templates do not match and false is returned.

Listing 5.11 exemplarily shows the function *matchRelationshipConstraints* to clarify the proposed concept of framework embedded functions invoking type-specific plugins with a simple example. The function requires two Relationship Templates, denoted by r_1 and r_2, and returns a Boolean value. First the RelationshipConstraints of both Relationship Templates are retrieved and stored in variables (line 5 and 6). If one of both variables is null, true is returned, since the two *RelationshipConstraint* elements cannot contradict each other (line 8-10). If both Relationship Templates have RelationshipConstraints, the function *matchRelationshipConstraintsTypeSpecificContent* is invoked in line 12 calling the plugin that is appropriate for the Relationship Type of r_1 and r_2.The resulting Boolean value is then returned to the calling *match* function (see Listing 5.10).

```
Algorithm to match RelationshipConstraints
Input: Relationship Template $r_1$
Input: Relationship Template $r_2$
Output: Boolean
matchRelationshipConstraints($r_1$,$r_2$)
  RelationshipConstraints $rc_1$ = $n_1$.getRelationshipConstraints()
  RelationshipConstraints $rc_2$ = $n_2$.getRelationshipConstraints()

  if $rc_1$ == null $\vee$ $rc_2$ == null then
    return true
  end if

  return matchRelationshipConstraintsTypeSpecificContent($rc_1$,$rc_2$)
end
```

Listing 5.11: Function matchRelationshipConstraints

5.3 Matching in the Context of Group Templates

Group Templates are subgraphs consisting of Node Templates, connected by Relationship Templates, and possibly more Group Templates. Assumption 8 states that Relationship Templates only target Node Templates in the Group Template and vice versa but not the Group Template directly. When matching inside a Topology Template or between two Topology Templates and the algorithm detects a Group Template, the matching algorithm has to be invoked recursively. In doing so, the contents of the Group Template are input for the algorithm together with the contents of either the second Topology Template or the whole Topology Template itself in case of inside matching. Fig. 5.11 shows two Topology Templates and a calculated Correspondence from the database's *Windows OS* inside the left-hand side Group Template to the applications server's *Windows OS* in the right-hand side Node Template. When matching Topology Templates that possess Group Templates, several different cases occur that must be discussed. On the one hand, these cases center on the question when the matching algorithm is allowed to recursively enter a particular Group Template to match its content inside the Group Template or with other Node Templates outside. On the other hand, an important question is how to deal with the matching of Node and Relationship Templates across different levels of nested Group Templates.

Furthermore, a concept that has to be introduced in this context is the notion of a Correspondence between Group Templates in order to unify corresponding Group Templates.

Definition 5.4 (Group Template Correspondence): *A Group Template Correspondence is defined as an indicator that two particular Group Templates have non-contradictory Policies and can be unified in a subsequent merging step.*

The Group Template that marks the source of the Correspondence also stores all found correspondences to other Group Templates.

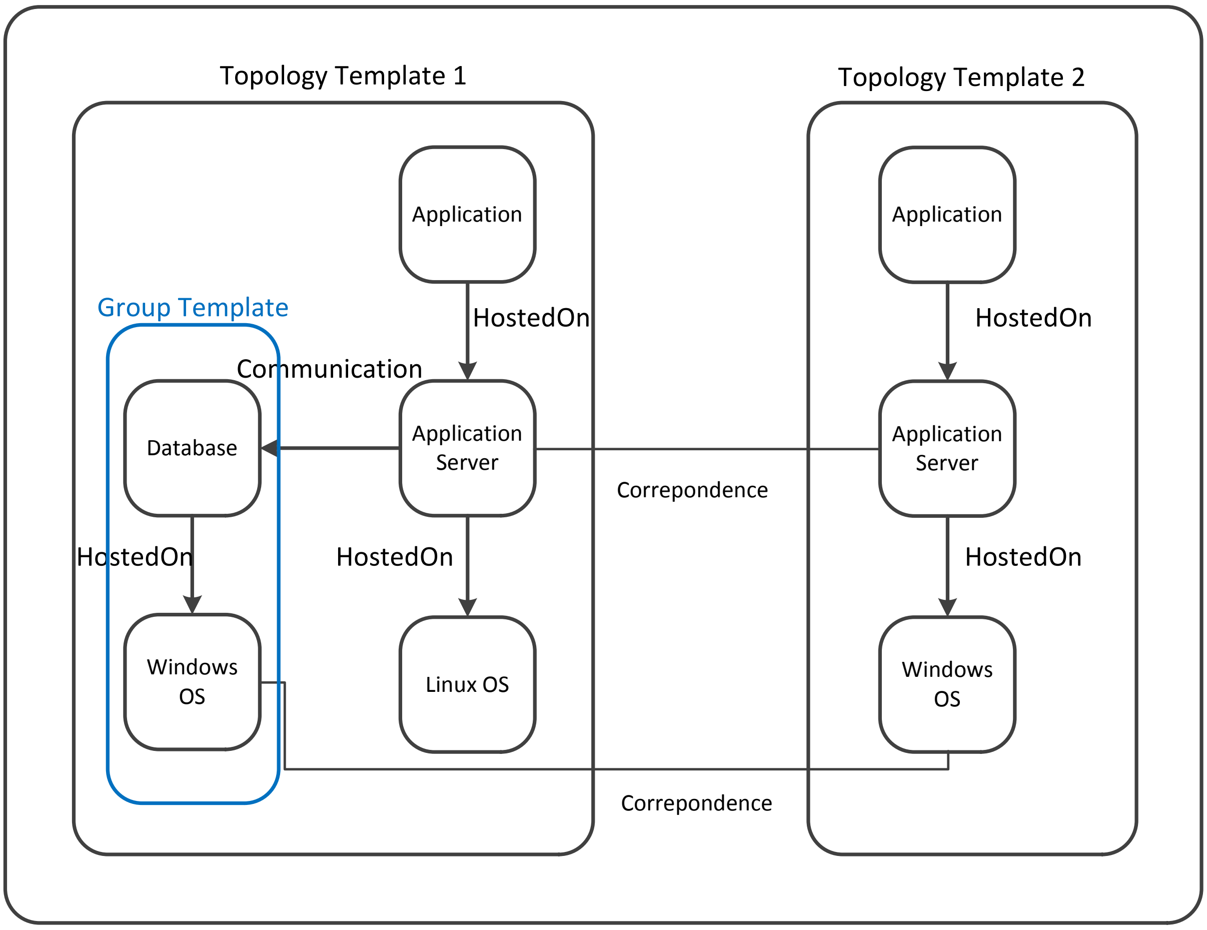


Fig. 5.11: Matching in the context of Group Templates

The Group Template Correspondence can be used to unify Policy-corresponding Group Templates in the merging step. Note that in order to establish a Correspondence between two Group Templates their "Nesting Level" has to be considered.

Definition 5.5 (Nesting Level and Group Template Hierarchy): *The Nesting Level is the number of Group Templates that must be traversed to reach a particular Node, Relationship or Group Template. The direct content of a Topology Template is defined to be on level* 0*, whereas level* n *indicates that an element is nested inside* n *Group Templates. The Group Templates from level* 0 *to level* $n-1$ *are called parents of the elements on level* n *and form a Group Template Hierarchy. A Group Template on level* 0 *does not have any parents. A Group Template on level* n *is called the child of a Group Template on level* $n-1$.

In order to be able to navigate through the Group Template Hierarchy and perform algorithms on it, the data structure GroupTemplateHierarchy, depicted in Listing 5.12, is introduced.

```
Data structure GroupTemplateHierarchy
data structure GroupTemplateHierarchy {
  GroupTemplateHierarchy child
  GroupTemplateHierarchy parent
  Group Template groupTemplate
  int nestingLevel
}
```

Listing 5.12: Data structure GroupTemplateHierarchy as double-linked list

The data structure is similar to a double-linked list [40]. Each GroupTemplateHierarchy element on Nesting Level n has a *child* field, which has a pointer to a GroupTemplateHierarchy element on a Nesting Level $n + 1$ and a *parent* field pointing to Nesting Level $n - 1$. The Group Template field points to a Group Template assigned to this particular GroupTemplateHierarchy element. Furthermore, to every element in a Topology Template, i.e. Node, Relationship and Group Templates, a GroupTemplateHierarchy element is assigned that represents its parent in the Group Template Hierarchy. Thus, for every element one can infer its Nesting Level $n + 1$ by examining the Nesting Level n of the parent element. If there exists no parent GroupTemplateHierarchy, the particular element is on Nesting Level 0. The assignment must be conducted in a step prior to the actual invocation of the *findMapping* function.

5.3.1 Extension of the Basic Matching Algorithm for Node Templates

After having introduced the concept of Group Template Correspondences and Hierarchies, the next step is to discuss the necessary conceptual extensions to the basic algorithm from Listing 5.1 and Listing 5.2 to incorporate the processing of Group Templates and their content. The extended version is displayed in Listing 5.13 and Listing 5.14. Note that some details, such as the creating of Node Template Correspondences and the handling of Relationship Templates, are left out due to shortage of space. The input parameters of the function *findMapping* have changed: Instead of requiring two Topology Templates now two sets, denoted by E_1 and E_2 are required. The sets contain *tExtensibleElements* which can be instances of Node, Relationship or Group Templates, i.e. the content of Topology Templates as well as Group Templates. This modification is necessary as the *findMapping* function is to be invoked for mapping the content of two Topology Templates as well as mapping the content of Group Templates recursively. The second modification comprises of the adding of two Group Templates, denoted by $g_{parent1}$ and $g_{parent2}$. These Group Templates are the parent Group Templates of the respective recursion depth of the algorithm, i.e. the parent of the Nesting Level the algorithm currently operates on. The basic principle of the algorithm remains the same. There are two nested loops where each Node Template of Topology Template 1 is compared with each Node of Topology Template 2. However, it is important to remember that in the first and in the second invocation of *findMapping* both Topology Templates are identical to conduct an inside matching and not until the third invocation the different, and already merged, Topology Templates are input for the function.

```
Extended algorithm for finding a mapping between Node Templates 1
1  Input: Set of Elements $E_1$
2  Input: Set of Elements $E_2$
3  Input: Group Template $g_{parent1}$
4  Input: Group Template $g_{parent2}$
5  Input: Mapping M = ∅
6  Output: Mapping M
7  findMapping($E_1$,$E_2$,$g_{parent1}$,$g_{parent2}$,M)
8    for each Element $e_1 \in E_1$ do
9      if $e_1 \in$ Node Templates then
10       Node Template $n_1$ = (Node Template)$e_1$
11     else if $e_1 \in$ Group Templates then
12       Group Template $g_1$ = (Group Template)$e_1$
13       findMapping($g_1$.getElements(),$E_2$,$g_1$,$g_{parent2}$,M)
14     end if
15
16     for each Element $e_2 \in E_2$ do
17       if $e_2 \in$ Node Templates then
18         Node Template $n_2$ = (Node Template)$e_2$
19       else if $e_2 \in$ Group Templates then
20         Group Template $g_2$ = (Group Template)$e_2$
21           if isGroupTemplateAccessPossible($g_1$,$g_2$,$n_1$,$g_{parent1}$,$g_{parent1}$)
22           then
23             Set $E_{new}$ = new Set()
24             $E_{new}$.add($n_1$)
25             findMapping($E_{new}$,$g_2$.getElements(),$g_{parent1}$,$g_2$,M)
26           else
27             continue
28         end if
29       end if
30       ...
```

Listing 5.13: Extended function findMapping part 1

The algorithm starts in line 8 with the first loop that iterates over all elements of E_1. If the current element e_1 is an instance of a Node Template, it is converted into one and the algorithm proceeds to the second loop iterating over E_2. If, however, e_1 is an instance of a Group Template, it is converted into one (line 12) and the function is invoked recursively with the elements of that Group Template, denoted by g_1 forming the new set E_1 and g_1 as the new parent Group Template $g_{parent1}$. This illustrates that the algorithm will recursively enter the left-hand side Topology Template respectively Group Templates until the first Node Template is found. Only then the iteration through the second set may commence.

If the second loop beginning at line 16 finds a Node Template, two Node Templates are found altogether and the algorithm proceeds to the actual matching. If, however, the current element e_1 is instance of a Group Template, denoted by g_2 the subroutine *isGroupTemplateAccessPossible* is invoked. This function encapsulates the evaluation if the recursive invocation of *findMapping* with the content of Group Template 2 should be allowed. It checks which of several cases is at hand regarding the Nesting Level of Node Template n_1 and the entering of the Group Template g_2. The cases are discussed in the next section. If the subroutine returns true in line 20, the current Node Template n_1 is added to a new set denoted by E_{new} as single element[7] and *findMapping* is invoked recursively with E_{new} as new left-hand side set, the content elements of g_2 as new right-hand side set, the current $g_{parent1}$ and g_2 as the new $g_{parent2}$ Group Template. If the subroutine returns false, the processing of the current g_2 is skipped. The details of *isGroupTemplateAccessPossible* are shown below.

```
Extended algorithm for finding a mapping between Node Templates 2
30            ...
31          if $nt_1$ == $nt_2$
32          ∨ isDerivedFrom($nt_1$,$nt_2$)
33          ∨ isDerivedFrom($nt_2$,$nt_1$)
34          ∨ hasCommonAncestor($nt_1$,$nt_2$)
35          then
36            Matcher matcher = createNodeTemplateMatcher($nt_1$,$nt_2$)
37            if ¬isMatchingPossible($n_1$,$n_2$,$g_{parent1}$,$g_{parent1}$) then
38              continue
39            end if
40            if matcher.match($n_1$,$n_2$) then
41              ...
42            end if
43          end if
44        end for each
45      end for each
46    end
```

Listing 5.14: Extended function findMapping part 2

The last modification to be introduced is the invocation of the additional subroutine *isMatchingPossible* in line 37. Inside this function it is evaluated if the current Node Templates n_1 and n_2 having different Nesting Levels may be matched. If not, the algorithm must skip the processing of the two Node Templates. The different cases and the details of the subroutine are discussed below in Section 5.3.3.

[7]This is a necessary action to ensure that the first loop only finds one Node Template and does not start to invoke *findMapping* recursively by itself.

5.3.2 Different Cases of Recursive Access to Group Templates

The following section analyzes the different cases occurring when a left-hand side Node Template, as before denoted by Node Template 1 or n_1, is found during the iteration over set 1, denoted by E_1, and a Group Template g_2 during the iteration over set 2, denoted by E_2. These cases center on the question, which Nesting Level n_1 has in relation to g_2 and if the latter one may be entered recursively. The pseudo code implementation of the different cases is found below as function *isGroupTemplateAccessPossible*. Basically, it is an optimization of the *findMapping* algorithm to prevent the entering of Group Templates in the cases where it is clear that n_1 and the content of g_2 are incompatible from the beginning because of their Policies. The second goal of the function is to find possible Group Template Correspondences. The Node Template n_1 and the Group Template g_2 currently under consideration are indicated by means of green borders and the recursive entering of g_2 is depicted by a dotted arrow in the following examples.

Case1: Matching Inside one Group Template

The first case to be discussed is depicted in Fig. 5.12. If a Topology Template has at least one Group Template, the matching algorithm must be able to recursively step into the Group Template in order to match its contents.

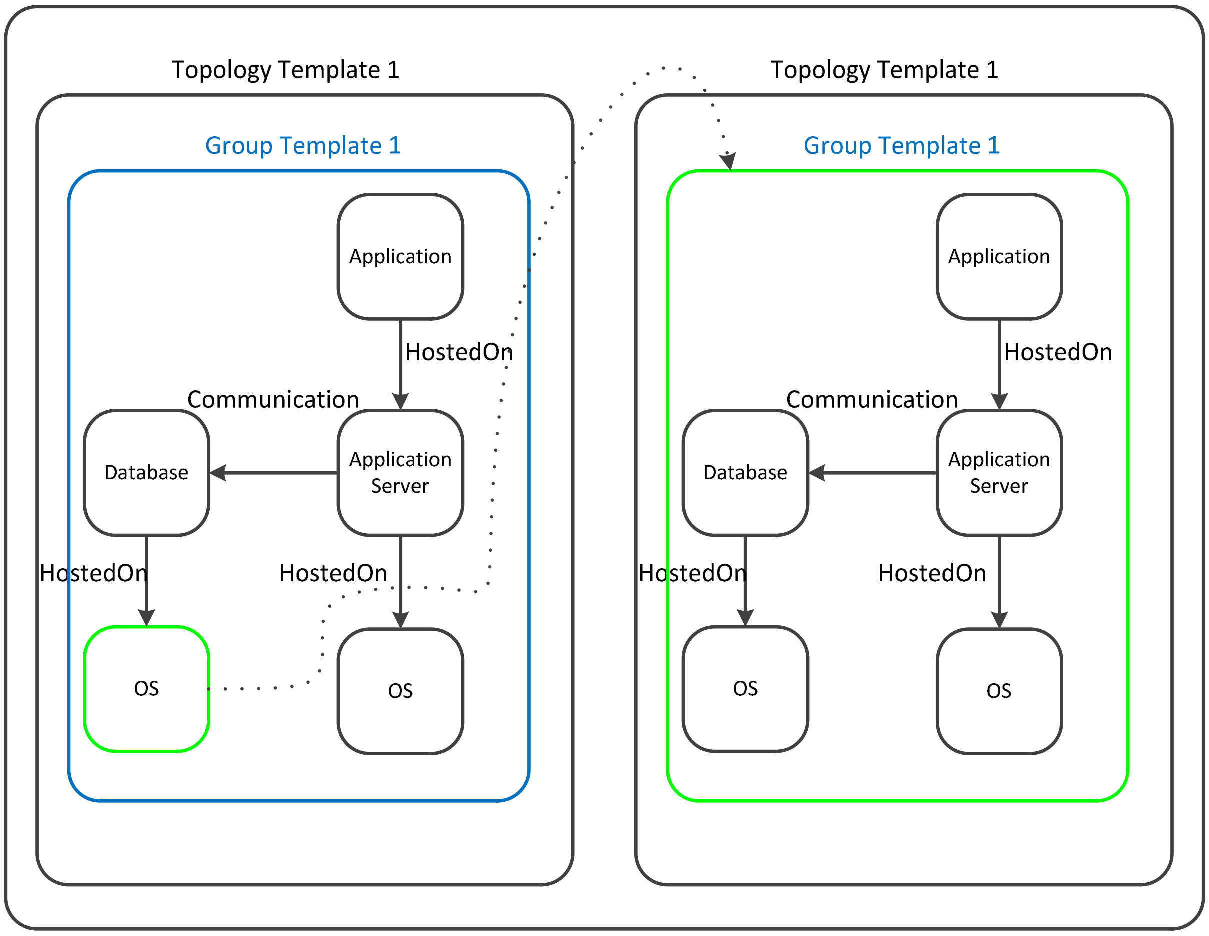


Fig. 5.12: Example of matching inside a Group Template

The example in Fig. 5.12 shows a green Node Template on the left-hand side, i.e. the current value of the variable n_1 of the *findMapping* function, that has to recursively enter its own parent Group Template 1 to gain access to other Node Templates and compare their properties with its own.

Case 2: Correspondences Between Group Templates on the Same Nesting Level
As presented in Section 2.3.1 a Group Template can also have Policies that specify the management practices concerning a particular Group Template. If the content, i.e. predominantly the Node Templates of two Group Templates, are matched, regardless if the Group Templates are residing inside one Topology Template or in two different ones, their Policies have to be evaluated. Only if the individual *Policy* elements do not contradict each other, the matching of the Group Templates content can be allowed by the TOSCAMerge framework, i.e. the algorithm invokes itself recursively with the content of the Group Templates. To indicate the equivalence of the Policies a Group Template Correspondence between the two Group Templates under investigation is established (Fig. 5.13).

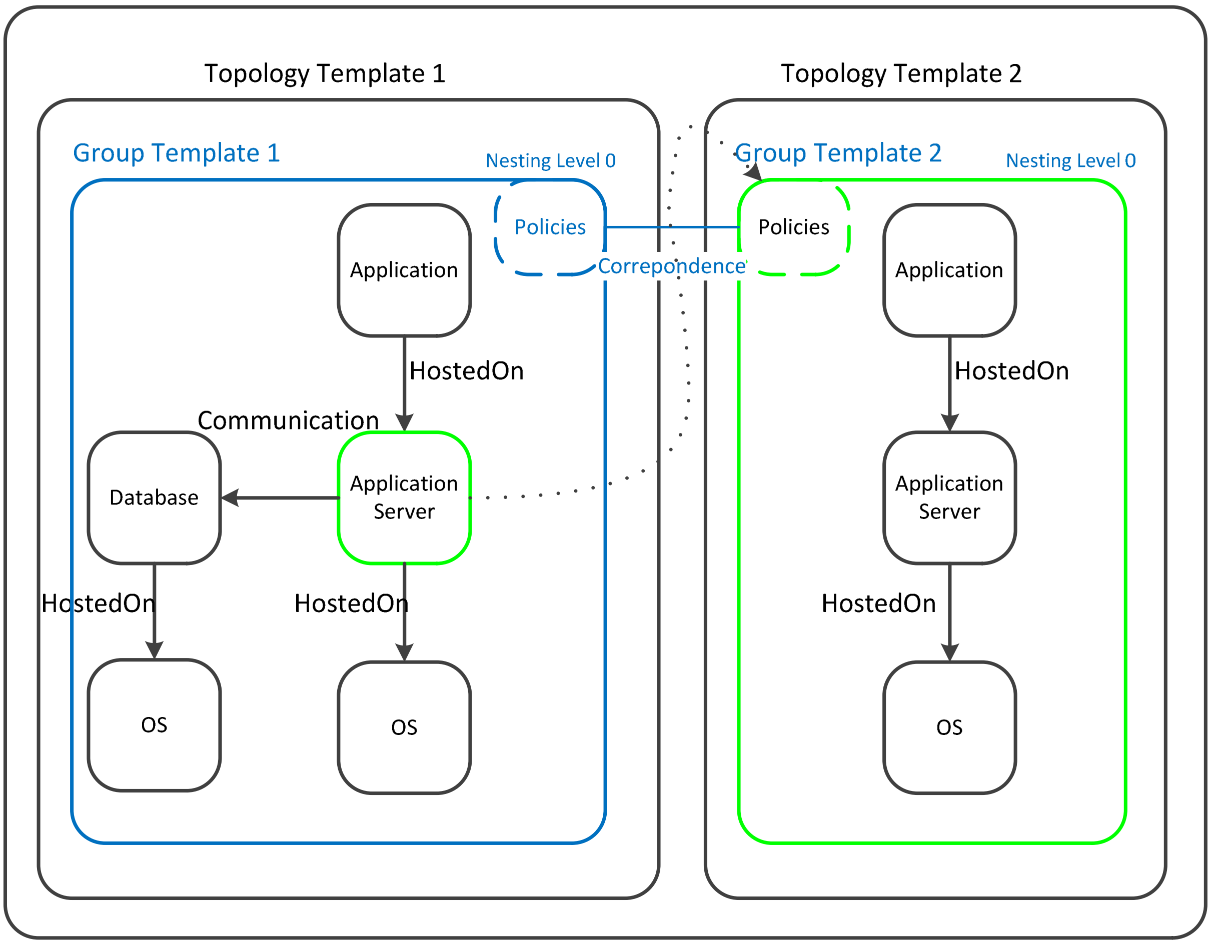


Fig. 5.13: Example of Group Template Correspondence on Nesting Level 0

A Group Template Correspondence can only be established if and only if they are on the same Nesting Level and also all their parent Group Templates have a Correspondence to their counterpart on the same level. The depicted example also illustrates the *Application Server* Node Template in Group Template 1 can only be matched to other Node Templates in

Group Template 2 if the mentioned Group Template Correspondence exists. Otherwise the findMapping algorithm would not enter Group Template 2 recursively.

Fig. 5.13 shows two Group Templates that are on Nesting Level 0, whereas Fig. 5.14 exemplifies that in order to analyze if Group Template 3 and 4 on Nesting Level 1 correspond to each other, a Group Template Correspondence on Nesting Level 0 has to be present. Group Template 6 on Nesting Level 2 cannot be matched with another Group Template in this example as there is no counterpart Group Template on the same Nesting Level in Topology Template 1. From the viewpoint of left-hand side Node Template, marked with the green border, Group Template 4 can be entered since it has a Correspondence with Group Template 3, the parent of the Node Template, on the same Nesting Level. However, this requires that the algorithms current recursion depth is already inside Group Template 2. Otherwise it is about case 3, which covers the entering of Group Template when the left-hand side Node Template's parent Group Template is on a different Nesting Level than the Group Template to be entered.

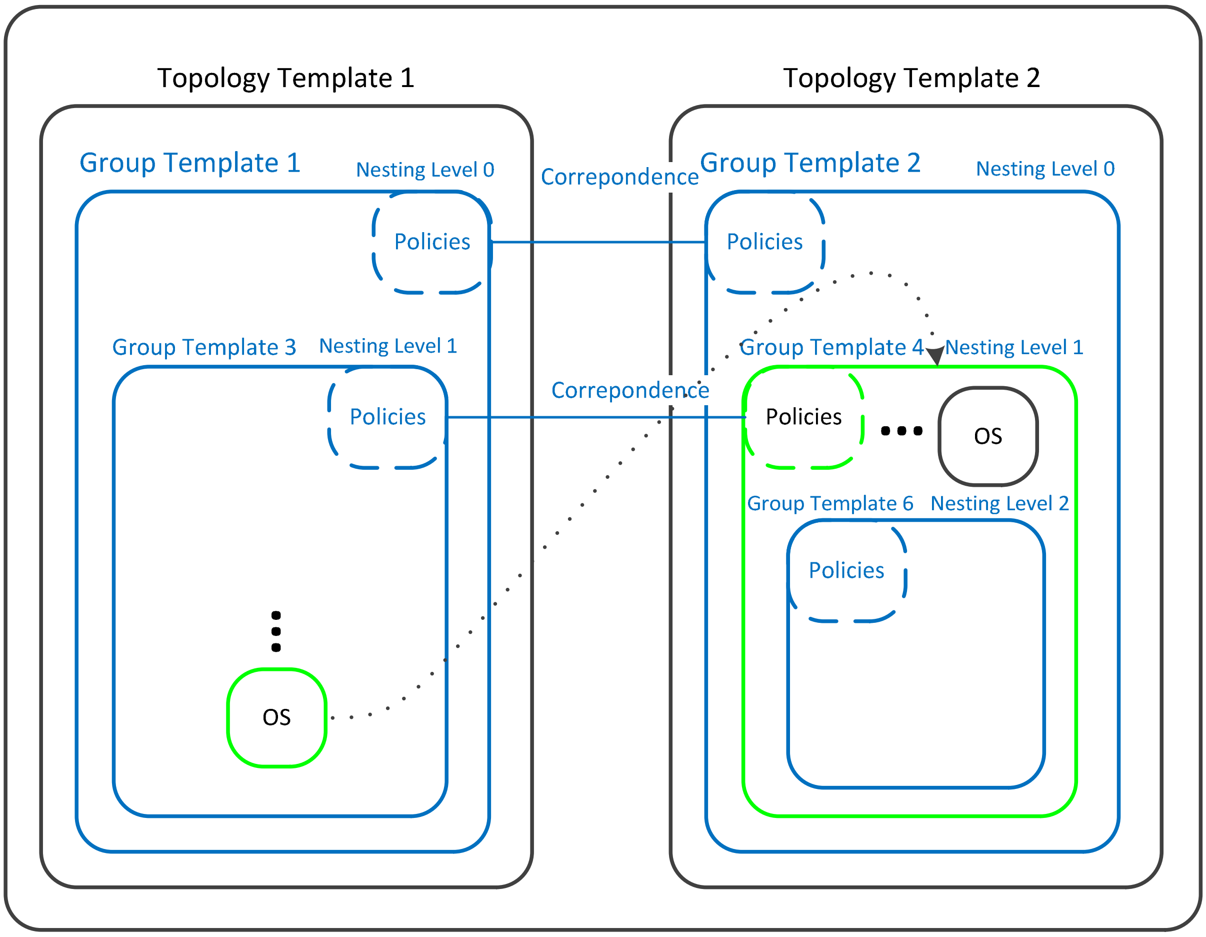


Fig. 5.14: Example of correspondences on a deeper Nesting Level

Case 3: Accessing Group Template Located on Different Nesting Levels

The next case checks the access to the right-hand side Group Templates when the already found Node Template on the left-hand side is nested in a Group Template that is (1) not on Nesting Level 0 and (2) not on the same Nesting Level as the right-hand side Node Template to be accessed. Fig. 5.15 shows such a case: To compare the found Node Template, highlighted with green borders, to Node Templates inside Group Template 2, the latter has to be accessed recursively. However, access should only be granted if Group Template 1 and 2 have non-contradictory Group Template Policies and can be merged in a subsequent step. Thus, it must be evaluated if a Group Template Correspondence between Group Template 1 and Group Template 2 can be established or already exists. If so, access must be granted by the algorithm. The example presented in Fig. 5.15 is called case 3a and the Group Template to be accessed is on Nesting Level 0.

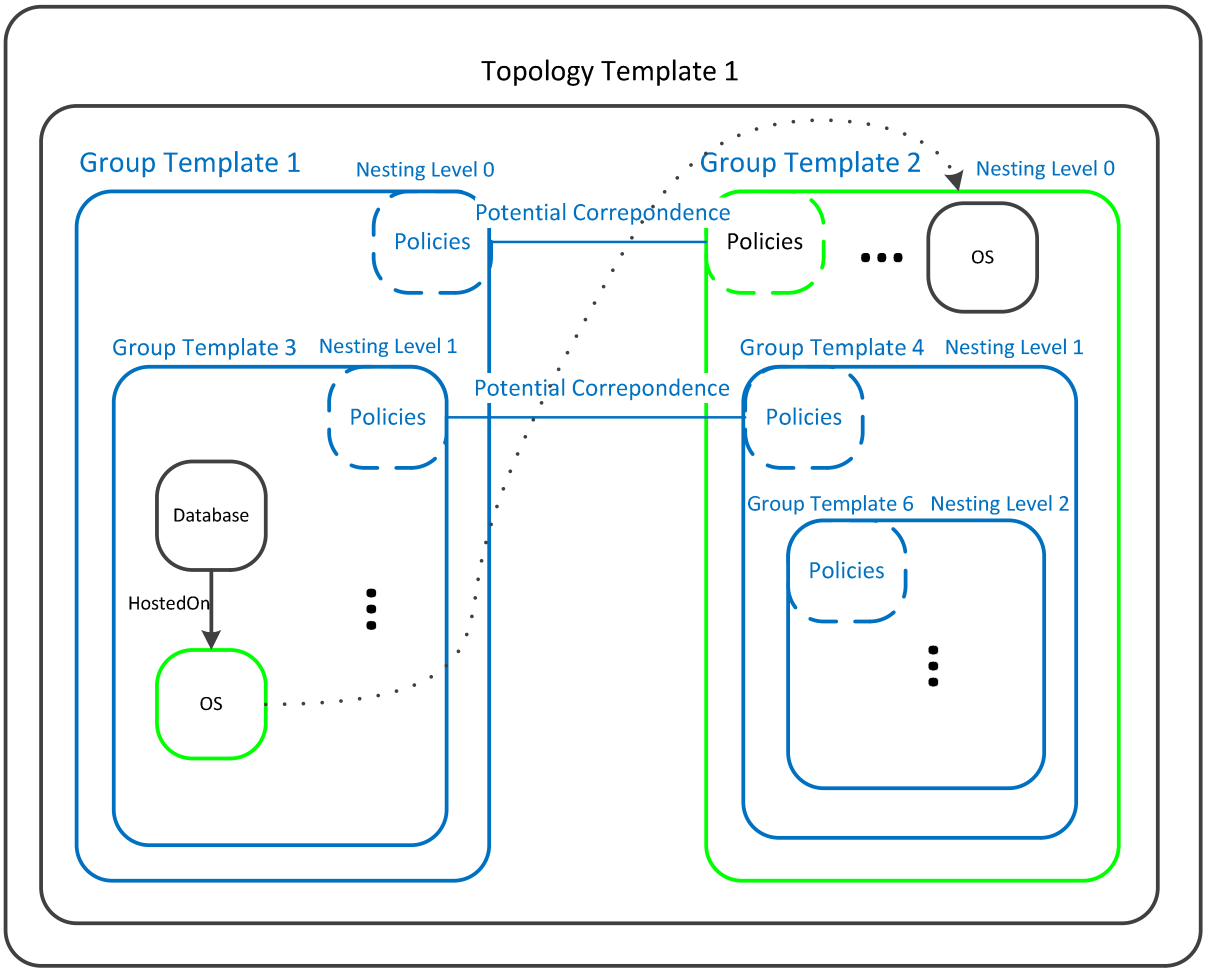


Fig. 5.15: Example of Group Template access case 3a

There also exists a derivation of this case, shown in Fig. 5.16. In this case 3b, the right-hand side Group Template 6 to be accessed still fulfills condition (1) and (2), but this time it is not on Nesting Level 0 but residing deeper on Nesting Level 2. The comparison of its content with the left-hand side Node Template can only take place if the algorithm permitted the access to the parent Group Templates of Group Template 6 and the current focus is now on entering the latter one. Access is decided by evaluation of the accumulated Policies[8] of the left-hand side Node Template and the right-hand side Group Template. If they match, the Group Template is entered. Otherwise the policies contradict and entering would only cost

[8] Accumulated Policies are the combined Policies of all parent Group Templates of one element including its own. A detailed definition follows below.

computing resources, although it is already clear that no matching Node Template will be found, either directly in the Group Template or even nested deeper.

Additionally, on its way down through the Group Template Hierarchy the algorithm implementing the cases must check if a Correspondence between the Group Template and its counterpart on the same Nesting Level in the right-hand side Node Template's Hierarchy is existing or can be established. This ensures that in the end all possible Group Template Correspondences have been found. Relying on the same level comparison mentioned in case 2 is not sufficient: if on a particular Nesting Level on the left-hand side no Node Templates but only Group Templates exist, the extended *findMapping* algorithm would immediately enter those Group Template and no same Nesting Level comparison to the right-hand side could be conducted. Therefore, the comparison on the way down is immanent for finding all Group Template Correspondences.

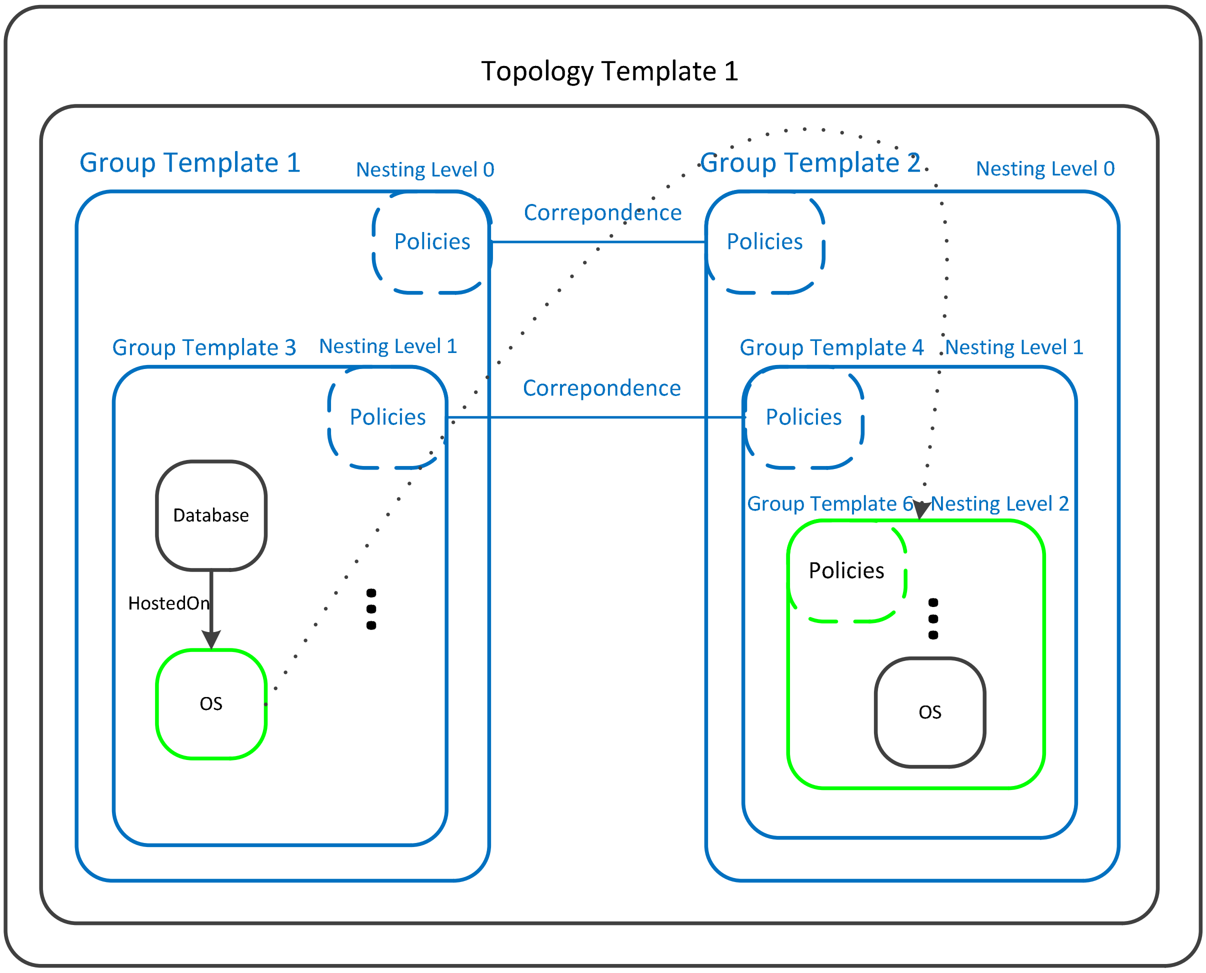


Fig. 5.16: Example of Group Template Access case 3b

Case 4: Left-hand Side Node Template is on Nesting Level 1

The fourth case assumes the left-hand side Node Template on Nesting Level 1, i.e. its parent Group Template is on Nesting level 0, and the right-hand side Group Template on a Nesting Level greater than 1.[9] The case is very similar to case 3, but has the slight difference that it is obvious from the outset that no Group Template Correspondence can exist between the Node Template's parent, i.e. Group Template 1 and the Group Template to be entered, i.e. Group Template 6 in the present example. Therefore, before entering again the accumulated Policies of the Node Template have to be matched with those of the Group Template.

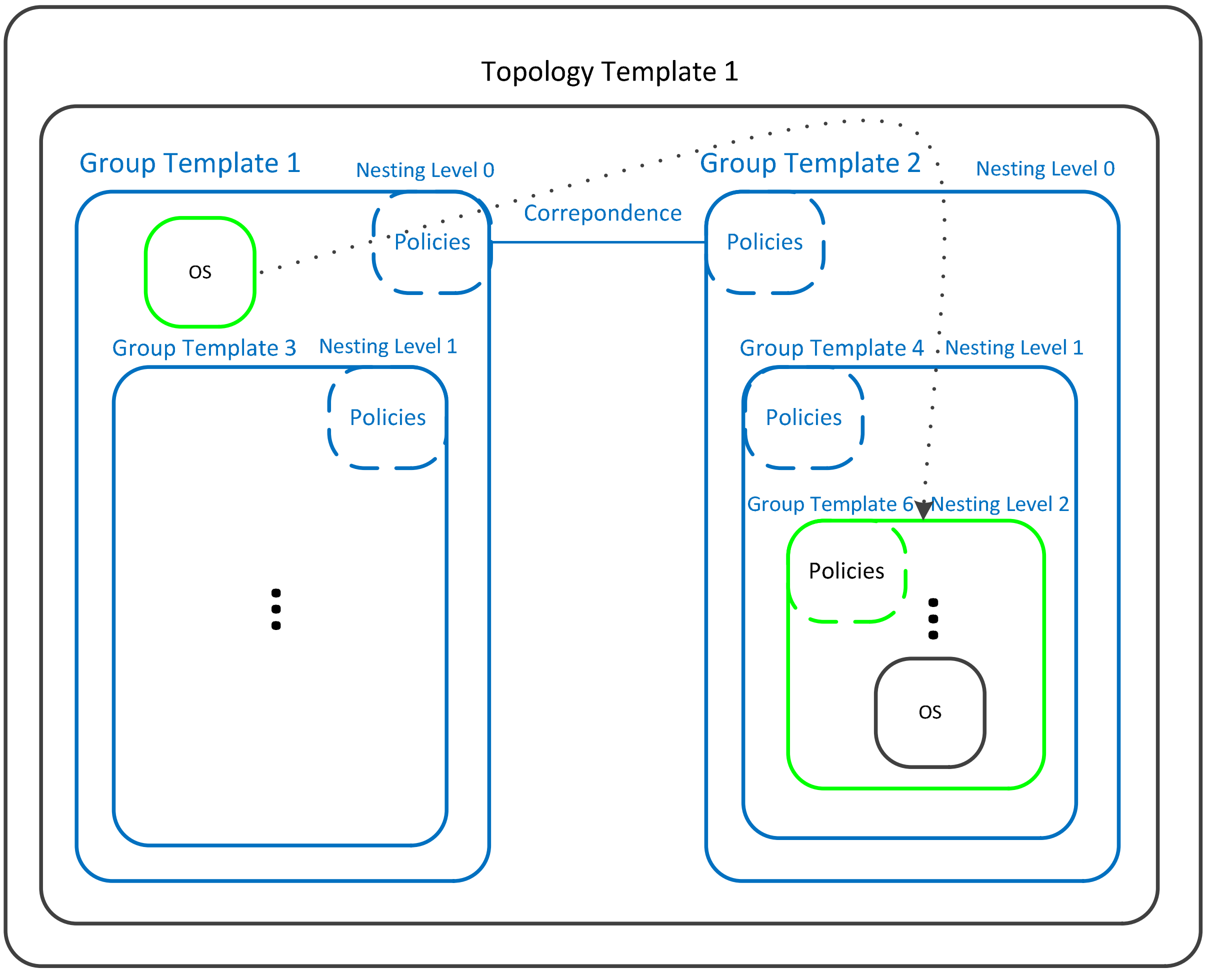


Fig. 5.17: Example of Group Template access case 4

Case 5: Left-hand Side Node Template is on Nesting Level 0

The last relevant case is depicted in Fig. 5.18. The left-hand side Node Template is on Nesting Level 0 and Group Template 1 or any deeper nested Group Template is to be entered recursively. The framework must once again compare the accumulated Policies of both elements to decide on entering.

[9] Otherwise case 2 would be at hand.

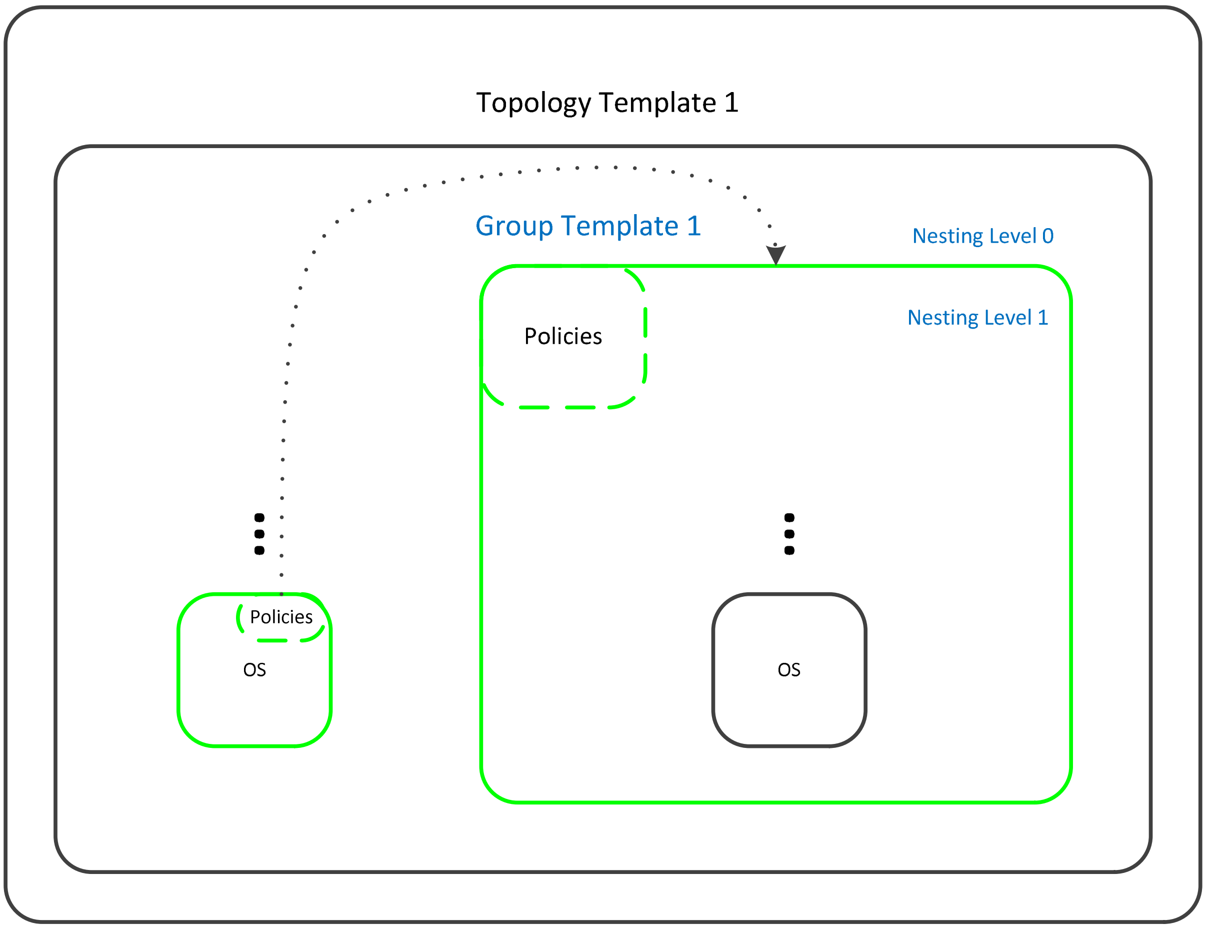


Fig. 5.18: Example of Group Template access case 5

Algorithm for Evaluating the Recursive Access to Group Templates

After having analyzed the different cases when accessing a Group Template on the right-hand side, the following section discusses the algorithms necessary to implement the cases. Listing 5.15 and Listing 5.16 show the function *isGroupTemplateAccessPossible* separated in two parts due to the size of the algorithm. The function requires five input parameters: One Node Template, denoted by n_1, and three Group Templates, denoted by g_2, $g_{parent1}$ and $g_{parent2}$. The Node Template represents the left-hand side Node Template referred to in the discussion of the different cases, g_2 represents the Group Template over which the access decision has to be made. $g_{parent1}$ and $g_{parent2}$ represent the parent Group Templates of n_1 respectively g_2. The last input parameter is a set of Group Template Correspondences denoted by GTC. The output is a Boolean value.

The first step of the algorithm in line 8 is to create a new Group Template matcher that is able to match the Policies of either two Group Templates located on the same Nesting Level or a Node Template and a Group Template possibly located on different Nesting Levels.

```
Algorithm to decide if Group Template access is possible part 1
1  Input: Node Template $n_1$
2  Input: Group Template $g_2$
3  Input: Group Template $g_{parent1}$
4  Input: Group Template $g_{parent2}$
5  Input: Set of Group Template Correspondences GTC
6  Output: Boolean
7  isGroupTemplateAccessPossible($n_1$,$g_2$,$g_{parent1}$,$g_{parent2}$,GTC)
8    Matcher matcher = createGroupTemplateMatcher()
9
10   if $g_{parent1} \neq$ null then
11     if $g_{parent1}$ == $g_2$ then
12       return true
13     else if isOnSameLevel($g_{parent1}$,$g_2$) then
14
15       if matcher.match($g_{parent1}$,$g_2$) then
16         Group Template Correspondence c = new Group Template
17           Correspondence($g_{parent1}$,$g_2$)
18         GTC.add(c)
19         $g_{parent1}$.addCorrespondences(c)
20         return true
21       else
22         return false
23       end if
24
25     else if $g_{parent1}$.getParent() $\neq$ null $\wedge$
26       $g_{parent1}$.getParent().checkGroupTemplateCorrespondence($n_1$,$g_2$,GTC)
27     then
28       return true
29         ...
```

Listing 5.15: Function isGroupTemplateAccessPossible part 1

The first condition to be evaluated spans from line 10 over the whole part 1 of the algorithm up to line 33 in part 2 (see Listing 5.16). It checks if $g_{parent1}$, i.e. the parent Group Template of Node Template n_1, is available and therefore n_1 is at least on Nesting Level 1. If so, in line 11 of Listing 5.15 case 1 is implemented. If $g_{parent1}$ is equal to g_2 it is about the inside mapping of a Group Template and therefore, the algorithms returns true in line 12. Case 2 is represented by the lines 13-23. A subroutine, called *isOnSameLevel*, evaluates if $g_{parent1}$ and g_2 are on the same Nesting Level. The subroutine is not described in detail here due to space limitations, but in a nutshell, it checks if both GroupTemplateHierarchy elements $g_{parent1}$ and g_2 are on the same Nesting Level. If so, the Group Template matcher checks the accumulated Policies of both Group Templates. If they are not contradictory, a Group

Template Correspondence can be created and added to set GTC. Otherwise g_2 should not be entered and false is returned. Case 3 is represented by the lines 25-28. If $g_{parent1}$ has a parent GroupTemplateHierarchy element, i.e. Node Template n_1 resides at least on Nesting Level 2, and the subroutine *checkGroupTemplateCorrespondence* returns true, then access to g_2 can be granted. Also the distinction between case 3a and 3b happens inside the subroutine. A detailed discussion of the subroutine follows below in the context of Listing 5.17.

```
Algorithm to decide if Group Template access is possible part 2
29        ...
30        else if $g_{parent1}$.getParent() == null $\wedge$ $g_{parent2} \neq$ null then
31
32          return matcher.match($g_2$,$n_1$)
33        end if
34
35      else if $g_{parent1}$ == null then
36
37        return matcher.match($g_2$,$n_1$)
38      end if
39      return true
40    end
```

Listing 5.16: Function isGroupTemplateAccessPossible part 2

Line 30-33 implements case 4. If $g_{parent1}$ does not have any parent GroupTemplateHierarchy element, i.e. Node Template n_1 is on Nesting Level 1, and at the same time $g_{parent2}$ is not null, i.e. Group Template g_2 is on a Nesting Level greater than 1, the matcher evaluates the accumulated Policies of n_1 and g_2 and the appropriate Boolean value is returned.

The last case is represented by the lines 35-38: if $g_{parent1}$ is not available and therefore Node Template n_1 is residing on Nesting Level 0, the matcher also evaluates the accumulated Policies. If up to now, none of the cases applied, it is optimistically returned true. This should prevent that some Node Templates are accidentally not matched, because some unexpected case arose. This behavior is acceptable as the accumulated Policies of two Node Templates are checked before matching in any case (see Section 5.3.3).

Listing 5.17 displays the function *checkGroupTemplateCorrespondence* that is invoked in Listing 5.15 line 26 as a subroutine. It is an algorithm operating on the GroupTemplateHierarchy data structure representing the implementation of the cases 3a and 3b of the Group Template access optimization. The function requires three input parameters: The left-hand side Node Template denoted by n_1, the right-hand side Group Template denoted by g_2 and a set of Group Template Correspondences denoted by GTC. The return value is of type Boolean. Case 3a is implemented by the lines 6-14: if g_2 does not have a parent GroupTemplateHierarchy element, i.e. it is on Nesting Level 0, the root GroupTemplateHierarchy element is retrieved first by using the function *traverseToRoot* which follows the parent elements of each GroupTemplateHierarchy element.

```
Algorithm on GroupTemplateHierarchy to check GT Correspondence
Input: Node Template $n_1$
Input: Group Template $g_2$
Input: Set of Group Template Correspondences GTC
Output: Boolean
checkGroupTemplateCorrespondence($n_1$,$g_2$,GTC)
  if $g_2$.getParent() == null then
    GroupTemplateHierarchy root = traverseToRoot()
    if root.checkCorrespondencesOfLevel($g_2$,GTC)
      $\wedge$ checkAccess($g_2$,$n_1$) then
        return true
    else
      return false
    end if
  end if

  GroupTemplateHierarchy h =
    getHierarchyElement($g_2$.getParent().getNestingLevel() + 1)

  if h $\neq$ null then
    if h.checkCorrespondencesOfLevel($g_2$,GTC)
      $\wedge$ checkAccess($g_2$,$n_1$) then
        return true
    else
      return false
    end if
  else
    return checkAccess($g_2$,$n_1$)
  end if
end
```

Listing 5.17: Function checkGroupTemplateCorrespondence

Note that it is the root of Node Template n_1's Hierarchy that is considered here. This is due to the fact that the function *checkGroupTemplateCorrespondence* was invoked on n_1's parent in Listing 5.15 line 26. On this root element it is tested if a Group Template Correspondence to g_2 can be established using the function *checkCorrespondencesOfLevel*[10] and at the same time n_1's accumulated Policies must be compatible to those of Group Template g_2. Dependent on the outcome of the evaluation an appropriate Boolean value is returned.

[10] The function will be discussed below.

In line 16 the implicit else branch begins implementing case 3b. The GroupTemplateHierarchy element h, which is the GroupTemplateHierarchy element in the Hierarchy of n_1 that is on the same Nesting Level as g_2 is retrieved. It can be found by the subroutine *getHierarchyElement* invoked in n_1's Group Template Hierarchy. The reason for adding 1 to the Nesting Level of g_2's parent is the following: The Nesting Level information is located in the GroupTemplateHierarchy data structure (see Listing 5.12) and not directly in a Group or Node Template. Group Template g_2 retrieves the Group Template of its parent GroupTemplateHierarchy element[11] and infers its own Nesting Level by adding 1.

If an element h is found, similar to the previous case the existence or creation of a Node Template Correspondence is evaluated by the invocation of the function *checkCorrespondencesOfLevel* combined with the check of the accumulated Policies of n_1 and g_2 to evaluate the access to g_2. If h cannot be found, it means that g_2 has a greater Nesting Level than the Group Template n_1 is nested in and no Group Template Correspondence can be established on that level. Therefore only the access has to be checked (line 27).

It is also possible that Group Template Correspondences are created on Nesting Levels deeper than those of n_1, but this is acceptable as the accumulated policies are considered when creating a Correspondence.

Listing 5.18 shows function *getHierarchyElement* that is invoked in Listing 5.17 line 17. It has only one input parameter of type integer, denoted by l, indicating the Nesting Level where the element can be found. The output is the demanded GroupTemplateHierarchy element. As the function works on the GroupTemplateHierarchy data structure, each field is depicted in blue. The implemented algorithm first evaluates if the element it is currently working on the same Nesting Level as l. If so, the current GroupTemplateHierarchy element returns itself (line 4-6). Otherwise it is evaluated if the Nesting Level of the current element is greater than l. That means that the search must follow the parent pointer in the direction of the root element. This is done by invoking the *getHierarchyElement* on the *parent* field recursively if it is not null (line 6-10). If the Nesting Level is smaller than l, the same invocation is done on the *child* pointer provided it is possible (line 12-16). If no element can be found, null is returned in line 19.

[11] The groupTemplate field of this element does not hold g_2 but the GroupTemplate g_2 is located in. This is also the reason that the cases 3a and 3b be are differentiated as in case 3a there is no parent that can be asked for the Nesting Level.

```
Algorithm on GroupTemplateHierarchy to get a particular element
Input: int l
Output: GroupTemplateHierarchy
getHierarchyElement(l)
  if nestingLevel == l then
    return self
  else if nestingLevel > l then

    if parent ≠ null then
      return parent.getHierarchyElement(l)
    end if

  else if nestingLevel < l then

    if child ≠ null then
      return child.getHierarchyElement(l)
    end if

  end if
  return null
end
```

Listing 5.18: Function getHierarchyElement

Listing 5.19 shows the last function proposed for optimization of the recursive access of Group Templates respectively the creation of Group Template Correspondences. The function *checkCorrespondencesOfLevel* requires two input parameters: a Group Template, denoted here by g_2, and a set of Group Template Correspondences denoted by GTC_{in}. The output is of type Boolean.[12] The first step of the function in line 5 is to retrieve all Group Template Correspondences from g_2 and store them in the temporary set C.

Subsequently, in the lines 7-12 each Group Template Correspondence $c \in C$ is evaluated if it either spans from g_2 to the current *groupTemplate* pointer of the GroupTemplateHierarchy element or vice versa. Remember, the function was invoked on the Hierarchy element h, which holds the Group Template that is on the same Nesting Level as g_2 and in the Group Template Hierarchy of the left-hand side Node Template. If such a Group Template Correspondence exists, the function returns true. Otherwise, a Group Template matcher is created in line 14. The matcher then checks with its implemented *match* function if the value of the *groupTemplate* field and g_2 have matching accumulated Group Template policies. If so, a new Group Template Correspondence is created and added to the set GTC_{in}. Furthermore, it is returned true in line 20. If the Policies do not match, it is returned false.

[12] Technically speaking the set GTC_{in} must also be returned. However as there can only be one return value without using some data structure, only the Boolean value is shown for the sake of simplicity and a call-by-reference semantics for the set is assumed, i.e. the changes to the set are directly visible to other references of the set. This is similar to the semantics in Java [23].

The function *checkCorrespondencesOfLevel* is necessary, as discussed above in the context of case 3, to capture all possible Group Template Correspondences when matching.

```
Algorithm on GroupTemplateHierarchy to check Correspondences
Input: Group Template $g_2$
Input: Set of Group Template Correspondences $GTC_{in}$
Output: Boolean
checkCorrespondencesOfLevel($g_2$, $GTC_{in}$)
  Set C = $g_2$.getCorrespondences()

  for each Group Template Correspondence c $\in$ C do
    if c.getFrom() == $g_2$ $\wedge$ c.getTo() == groupTemplate
      $\vee$ c.getFrom() == groupTemplate $\wedge$ c.getTo() == $g_2$ then
      return true
    end if
  end for each

  Matcher matcher = createGroupTemplateMatcher()
  if matcher.match(groupTemplate,$g_2$) then
    Group Template Correspondence c = new Group Template
      Correspondence(groupTemplate,$g_2$)
    $GTC_{in}$.add(c)
    $g_2$.addCorrespondences(c)
    return true
  end if
  return false
end
```

Listing 5.19: Function checkCorrespondencesOfLevel

5.3.3 Matching of Node Templates on Different Nesting Levels

Until now it has been discussed how and when the matching algorithm must be able to enter Group Templates recursively in order to minimize the overall number of Node Templates that have to be compared. The next step is to look at the different cases when actually matching two Node Templates that are not inside the same Group Template. The matching of Node Templates that reside on different Nesting Levels has four subcases, regardless if two different Topology Templates are involved or if the matching takes place inside one Topology Template. The following depicted cases, however, only show inside Topology Template matching to maintain the continuity and simplicity of the examples. Note that when matching Node Templates that are on the same Nesting Level no extra cases have to be considered as there must exist a Group Template Correspondence between the two parent Group Templates. Otherwise the algorithm would not have entered the parent Group Template of the second Node Template. However, the algorithm still must allow the matching to happen.

Case 1: Left-hand Side on Nesting Level 0, Right-hand Side on Nesting Level 1
Fig. 5.19 shows the case where the left-hand side Node Template of the comparison is located on Nesting Level 0, whereas the right-hand side Node Template is located inside a Group Template on Nesting Level 1.

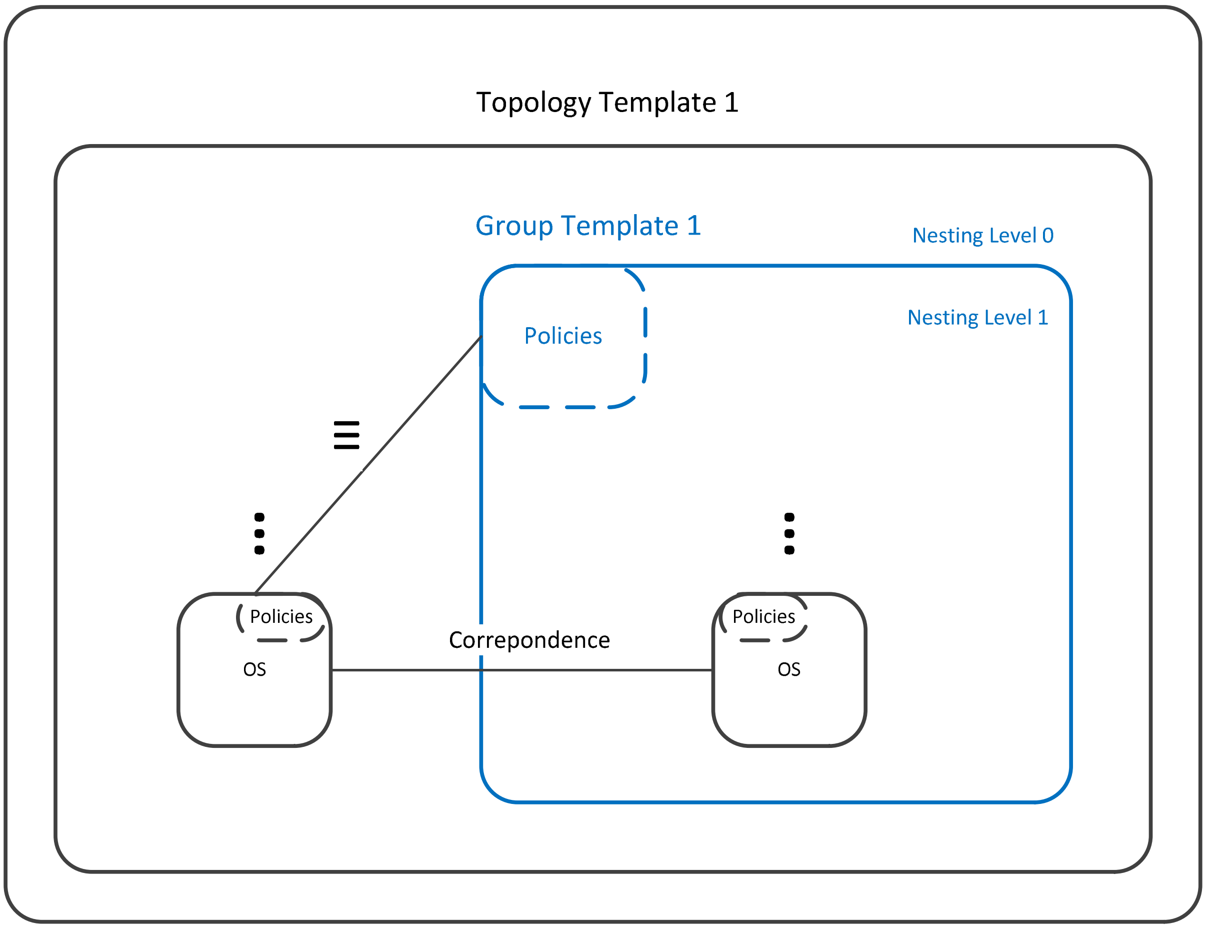


Fig. 5.19: Different Nesting Level case 1

The three dots above the Node Templates show that there might exist further Node and/or Relationship Templates, however, they do not matter for the depicted case. Before comparing the two *OS* Node Templates in Fig. 5.19, the Policies of the left-hand side Node Template must be compared with the Policies of the right-hand side Group Template. Only if they are compatible, as indicated by the equivalency symbol, the algorithm is allowed to proceed with comparing the two Node Templates.

Case 2: Right-hand Side on Nesting Level 0, Left-hand Side on Nesting Level 1
Very similar is the next case where the left hand side has a Nesting Level greater than 0 and the right hand side is on level 0. Looking at the example in Fig. 5.20, one observes that each time a Node Template of Group Template 1 is compared with a Node Template on the top Nesting Level 0, the Policies of Group Template 1 have to be compared to the Node Template outside and only if they are compatible, the comparison of the Node Templates under consideration may proceed.

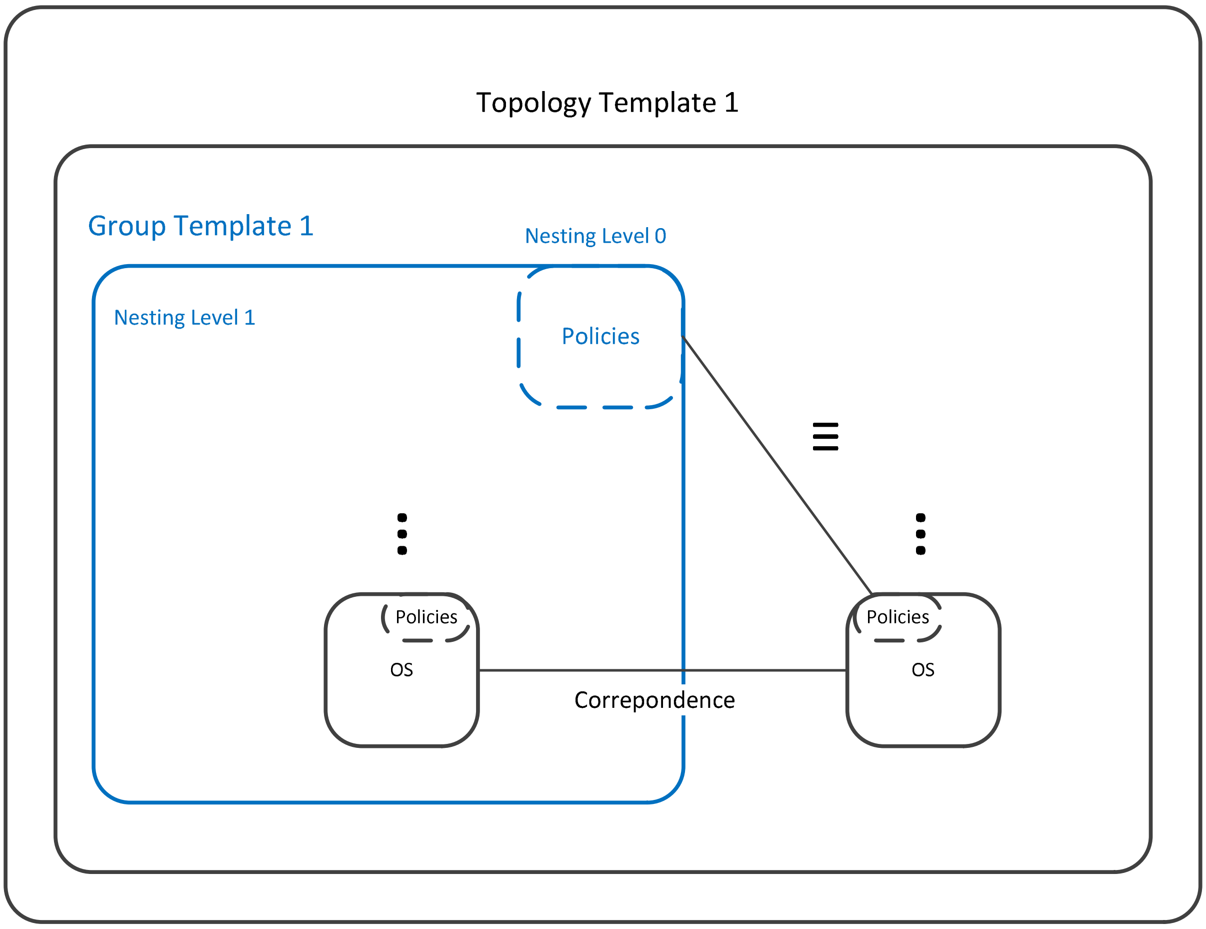


Fig. 5.20: Different Nesting Level case 2

Case 3: Left-hand Side on Deeper Nesting Level Than Right-hand Side

Fig. 5.21 shows case 3 when matching Node Template on different Nesting levels. This time, both Node Templates are on a Nesting Level greater than 0 but still not on the same level. However, it is not important for the discussion of this case, if the difference between Nesting Levels is one or more. Before matching the two *OS* Node Templates, the framework must calculate the accumulated Policies of the Group Template, in which the deeper nested left-hand side Node Template resides, and the accumulated Policies of the Node Template on the right-hand side that is on a lower nesting Level.

Definition 5.6 (Accumulated Policies): *In this work, the accumulated Policies of a policy attached entity, i.e. a Node or a Group Template, are the Policy elements of this entity on Nesting Level* n *and all Policy elements of the Group Templates located in a direct hierarchy from the Nesting Level* $n-1$ *to* 0.

In the example of Fig. 5.21, Group Template 3's accumulated Policies are its own and those of Group Template 1, whereas the right-hand side Node Template has its own Policies and also those of Group Template 1. Both dependencies are indicated by the dotted lines. With the calculated accumulated Policies it can be decided if the matching between the two *OS* Node Templates may proceed. If the accumulated Policies contain contradictory Policy elements, the framework must skip further processing. Note that the Group Template Policies along one hierarchy may have identical Policies, i.e. Policies that have identical values for their *name* and *type* attributes. When calculating the set of accumulated Policies the Policy elements must be inserted in a multiset and not overridden as in the case of derived Node Types. Subsequently, the type-specific plugin has the task to evaluate all the policies in the multiset.

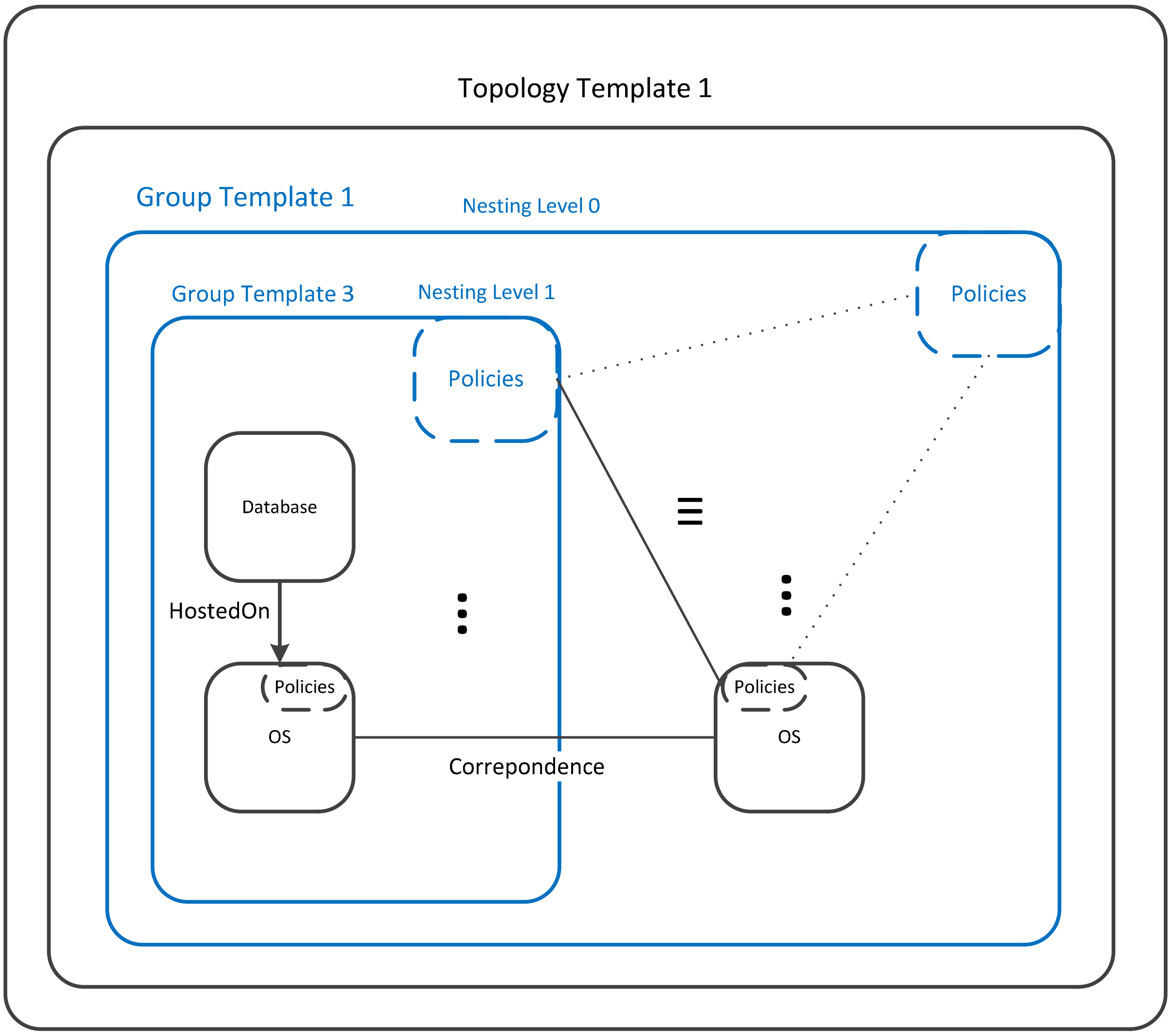


Fig. 5.21: Different Nesting Level case 3

Case 4: Right-hand Side on Deeper Nesting Level Than Left-hand Side

The last case is a derivation of case 3. The left-hand side Node Template that starts the comparison is now on a lesser Nesting Level than the right-hand side Node Template. Again the accumulated Policies have to be calculated by the framework before a matching can take place. If the accumulated Policies of the Node Template and the Group Template do not match, the overall matching algorithm skips the processing of two Node Templates and continues with the next iteration of the second loop.

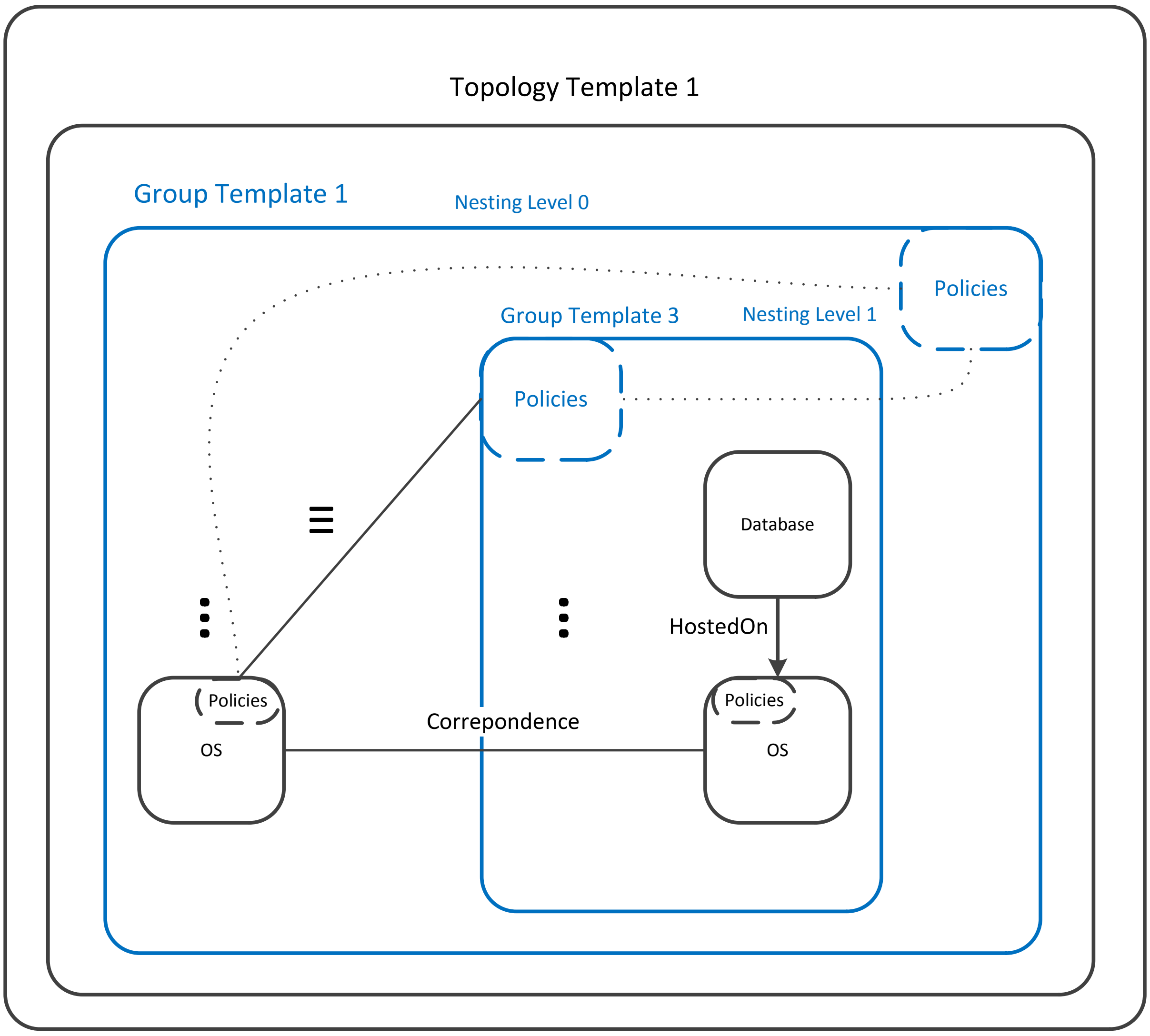


Fig. 5.22: Different Nesting Level case 4

Algorithm for Evaluating if Node Templates on different levels can be matched
In order to realize the delineated functionality the basic algorithm for finding a mapping introduced in Listing 5.1 has to be augmented in the way visible in Listing 5.13. Before the actual Node Template matcher is executed the function *isMatchingPossible* must be invoked in line 37. If it returns true the main algorithm may proceed, otherwise it skips one iteration. The function is depicted in Listing 5.20. It has four input parameters: two Node Templates, denoted by n_1 and n_2 and two Group Templates, denoted by g_1 and g_2. The Group Templates are the parent Group Templates mentioned above related to the extended *findMapping* function in Listing 5.13 and Listing 5.14. The return value is of type Boolean. In line 7-9 it is evaluated if both Node Templates have no parents at the same time, i.e. if they are not nested inside a Group Template. If so, the algorithm immediately returns true. Otherwise additional evaluations have to be conducted. In line 11, a Group Template matcher is created as it implements the framework's functionality for calculating the accumulated policies and providing a starting point for external domain-specific policy matching facilities.

```
Algorithm for deciding if the matching is possible
Input: Node Template $n_1$
Input: Node Template $n_2$
Input: Group Template $g_1$
Input: Group Template $g_2$
Output: Boolean
isMatchingPossible($n_1$,$n_2$,$g_1$,$g_2$)
  if $n_1$.getParent() == null $\wedge$ $n_2$.getParent() == null then
      return true
  else

    Matcher matcher = createGroupTemplateMatcher()

    if $n_1$.getParent() == null $\wedge$ $n_2$.getParent() $\neq$ null then
      if matcher.match($g_2$,$n_1$) then
        return true
      end if

    else if $n_1$.getParent() == null $\wedge$ $n_2$.getParent() $\neq$ null then
      if matcher.match($g_1$,$n_2$) then
          return true
      end if
    end if

    else if $n_1$.getParent() $\neq$ null $\wedge$ $n_2$.getParent() $\neq$ null then
      if $n_1$.getParent().getLevel() > $n_2$.getParent().getLevel() then
        if matcher.match($g_1$,$n_2$) then
          return true
        end if
      else if $n_1$.getParent().getLevel()<$n_2$.getParent().getLevel() then
        if matcher.match($g_2$,$n_1$) then
          return true
        end if
      else
        return true
      end if
    end if
  end if
  return false
end
```

Listing 5.20: Function isMatchingPossible

The next step in line 13 relates to case 1 outlined above. It is evaluated if Node Template n_1 is outside any Group Template and n_2 is nested inside. If so, n_1 is matched with Group Template g_2 using the *match* function of the Group Template matcher. The function will be reviewed below before long. If the accumulated policies of g_2 and $n_1$match, the *isMatching-Possible* function returns true in line 15. The lines 18-22 evaluate case 2 of matching over different nesting levels. If this case is at hand, the accumulated policies of g_1 and n_2 are matched and the result is returned. At line 24 the evaluation starts if the cases 3 or 4 are at hand. First, it is checked if both Node Templates have parents indicating that both Node Templates are nested somewhere inside an arbitrary Group Template. If this is the case, case 3 is expressed by line 25-28 when the Nesting Level of n_1's parent is greater than the Nesting Level of n_2's parent. The matching of policies then takes place between $g_1$and n_2. The last case is case 4, expressed by the lines 29-32. Here the Nesting Level of n_1's parents and, therefore, its own Nesting Level is lesser than the Nesting Level of n_2's parent. The policy matching is conducted between $g_2$and n_1. The last else branch in line 33 indicates that both Node Templates are on the same Nesting Level. Thus, as already mentioned, the algorithm simply returns true.

If up to now neither of the four cases returned true, false is returned in line 38 indicating the main algorithm must not proceed with the processing of the Node Templates n_1 and n_2.

5.3.4 Matching of Group Template Policies

As mentioned before, the policy matching takes place in a Group Template matcher. A Group Template matcher is not typed as there is no such concept as a “Group Type” in the TOSCA specification. However, the TOSCAMerge framework must provide a way to plug in a domain-specific policy matcher. The generic part of the framework is responsible for calculating the accumulated policies of the corresponding Group and Node Template. These policies are then handed over to a domain-specific policy matching implementation. The generic part of the policy matching can be found in the function *matchPolicies* depicted in Listing 5.21. The function requires two input parameters: a Group Template, denoted by g and a Node Template, denoted by n.

First the Node Type of the Node Template is retrieved and stored in a variable denoted by nt (line 5). The Node Type is then used to determine the derived Policies as described in Section 5.1.2 in Listing 5.7. The derived Policies are stored in a set denoted by $NodeTypePolicies$ (line 6). Two additional sets, denoted by $NTPolicies_h$ and $GTPolicies_h$, are filled with the Policies of the Group Template Hierarchy. Both sets only include the Policies of their Group Template Hierarchy beginning with their parent. The detailed algorithm for the Group Template will be shown exemplarily below in Listing 5.22 and Listing 5.23.

```
Algorithm to match the policies of Node and Group Templates
 1  Input: Group Template g
 2  Input: Node Template n
 3  Output: Boolean
 4  matchPolicies(g,n)
 5    Node Type nt = n.getNodeType()
 6    Set NodeTypePolicies = determineDerivedPolicies(nt)
 7    Set $NTPolicies_h$ = determineHierarchyPolicies(n)
 8    Set $GTPolicies_h$ = determineHierarchyPolicies(g)
 9    Set NTPolicies, GTPolicies = new Sets of Policies()
10
11    if n.getPolicies() $\neq$ null then
12      NTPolicies = n.getPolicies()
13    end if
14    if g.getPolicies() $\neq$ null then
15      GTPolicies = g.getPolicies()
16    end if
17
18    if NodeTypePolicies == $\emptyset \wedge$ NTPolicies == $\emptyset \wedge$ $NTPolicies_h$ == $\emptyset$ then
19      return true
20    end if
21    if GTPolicies == $\emptyset \wedge$ $GTPolicies_h$ == $\emptyset$ then
22      return true
23    end if
24
25    if NodeTypePolicies $\neq \emptyset$ then
26      for each Policy p $\in$ NodeTypePolicies do
27        if $p_1 \notin$ NTPolicies then
28          NTPolicies = NTPolicies $\cup$ p
29        end if
30      end for each
31    end if
32
33    NTPolicies.addAll($NTPolicies_h$)
34    GTPolicies.addAll($GTPolicies_h$)
35
36    return matchPoliciesTypeSpecificContent(NTPolicies,GTPolicies)
37  end
```

Listing 5.21: Function matchPolicies for Group and Node Templates

In the lines 11-16 of the function *matchPolicies*, the Policies of the n and g are retrieved and stored in the sets $NTPolicies$ respectively $GTPolicies$. Subsequently, it is evaluated if n or g do not have any Policies. For Node Template n this is true if all three sets

$NodeTypePolicies$, $NTPolicies$ and $NTPolicies_h$ are empty. In this case there are no Policies that are contradictory to the accumulated Policies of Group Template g, thus, the algorithm immediately returns true (line 18-20). The same evaluation is conducted for the sets $GTPolicies$ and $GTPolicies_h$. If they are both empty, the algorithm returns true. Otherwise the algorithm continues by comparing the Node Type Policies against the Node Template's Policies in the same way as discussed earlier with respect to Listing 5.7. In line 33 and 34 the effective set of Policies of the Node Template and the Policies of the Group Template are added to the overall accumulated Policies. These sets are then handed over to the *matchPoliciesTypeSpecificContent* function, which marks the starting point for any domain-specific matching implementation. The result is then returned to the caller of the generic function *matchPolicies*.

In Listing 5.22 the function of *determineHierarchyPolicies* for Group Templates, introduced

```
Algorithm to determine Group Template Policies of the Hierarchy
Input: Group Template g
Output: Set of Policies
determineHierarchyPolicies(g)
  Set P = new Set of Policies
  if g.getParent() ≠ null then
    P.addAll(g.getParent().getAllPolicies())
  end if
  return P
end
```

Listing 5.22: Function determineHierarchyPolicies for Group Templates

above, is depicted in detail. It is a very short function that basically only evaluates if the input Group Template g has a parent GroupTemplateHierarchy element and if so, invokes an algorithm on the data structure in Listing 5.12 that counts all Policies of overall the Group Template Hierarchy. The algorithm is located in the function *getAllPolicies* depicted in Listing 5.23. The fields of the data structure are displayed in blue color for easier visualization.

The function requires an empty set P as input and returns the same possibly non-empty set. The algorithm first evaluates in line 4 if the current GroupTemplateHierarchy element has a $parent$ GroupTemplateHierarchy element. If not, the Polices of the $groupTemplate$ are retrieved, if existing, and added to P. Then the algorithm returns P (line 5-8). If, however, a $parent$ GroupTemplateHierarchy element is found, the *Policy* elements of the corresponding Group Template are also retrieved, if existing, but the function *getAllPolicies* is invoked additionally on the $parent$ GroupTemplateHierarchy element. The nested invocation continues until it reaches one GroupTemplateHierarchy element that has no $parent$ element. In this case, P is returned with all found *Policy* elements.

```
Algorithm on GroupTemplateHierarchy to get all policies
Input: Set of Policies P = ∅
Output: Set of Policies
getAllPolicies(P)
  if parent == null then
    if groupTemplate.getPolicies() ≠ null then
      P.addAll(groupTemplate.getPolicies())
    end if
    return P
  else
    if groupTemplate.getPolicies() ≠ null then
      P.addAll(groupTemplate.getPolicies())
    end if
    return parent.getAllPolicies(P)
  end if
end
```

Listing 5.23: Function getAllPolicies

5.3.5 Relationship Templates in the Context of Group Templates

In this section the matching of Relationship Templates in the context of Group Templates is discussed briefly. Relationship Templates can point beyond the borders of Group Templates, i.e. from one Node Template on Nesting Level n to another Node Template on Nesting $n + 1$. However, the TOSCA specification does not state where the Relationship Templates have to be physically located: In this example they could either be located on Nesting Level n or on $n + 1$. To simplify the search for the incident Relationship Template to two corresponding Node Templates, it is proposed that all the Relationship Templates are stored in an extra set in a prior preparatory step. The step can be conducted in conjunction with the assignment to the GroupTemplateHierarchy where the two Topology Templates have to be traversed in a depth-first search [40] anyway.

Listing 5.24 shows that algorithm implementing that functionality. It requires three input parameters: a set of *tExtensibleElements*, denoted by E, which can hold Node, Relationship and Group Templates, an empty set of Relationship Templates denoted by R and a GroupTemplateHierarchy element, denoted by $parent$, which is null at the first invocation.

```
Algorithm building an overall set of Relationship Templates
 1  Input: Set of Elements E
 2  Input: Set of Relationship Templates R = ∅
 3  Input: GroupTemplateHierarchy parent
 4  Output: Set of Relationship Templates R
 5  buildRelationshipTemplateSet(E,R,parent)
 6    if parent ≠ null then
 7      parent.setNestingLevel(parent.getNestingLevel() + 1)
 8    end if
 9
10    for each Element e ∈ E do
11      e.setNumberOfMerges(0)
12      e.getCorrespondences().clear()
13      e.getCollectedCorrespondences().clear()
14
15      if e ∈ Node Templates then
16        e.setParent(parent)
17      else if e ∈ Relationship Templates then
18        e.setParent(parent)
19        R.add((Relationship Template)e)
20      else if e ∈ Group Templates then
21        Group Template g = (Group Template)e
22        g.setParent(parent)
23        GroupTemplateHierarchy h = new GroupTemplateHierarchy()
24        h.setParent(parent)
25        h.setGroupTemplate(g)
26
27        if parent ≠ null then
28          parent.setChild(h)
29          h.setNestingLevel(parent.getNestingLevel())
30        end if
31          return buildRelationshipTemplateSet(g.getElements(),R,h)
32      end if
33    end for each
34  end
```

Listing 5.24: Function buildRelationshipTemplateSet

The first step of the function is to evaluate if *parent* is not null. This is only the case if it is a recursive invocation of the function. If *parent* is not null, its Nesting Level is incremented by 1 (line 6-8). Subsequently, for each element $e \in E$, the number of merges, the sets of Correspondences and collected Correspondences are cleared. These fields or, respectively sets are important for merging in the next chapter but should be newly initialized after inside merging of a Topology Template.

Furthermore, it is tested if the current element e is either an instance of Node Template, Relationship Template or Group Template. If it is a Node Template, only the parent has to be added (line 15-16). If it is a Relationship Template, it is casted and added to R additionally (line 17-19). And finally, if it is a Group Template, some more steps have to be executed (line 20-32). First e is casted to a Group Template denoted by g. Then $parent$ is set as parent GroupTemplateHierarchy element to g. Additionally, a new GroupTemplateHierarchy, denoted by h, is created, adds $parent$ as parent element and g as the Group Template it manages. If $parent$ is not null, h can be added as child and the latter one can assume $parent$'s Nesting Level. In line 31 the function is invoked recursively with the content of g, the set R and h as new $parent$ element.

6 Concept for Merging of Topology Templates

After having discussed the different matching cases in the previous chapter, this chapter covers the application of the found Correspondences (the mapping), i.e. the merging of Node Templates, Relationship Templates and Group Templates and successively develops a merging concept and corresponding algorithms. Similar to the matching discussion a basic merging case with restricting assumptions will be proposed first and extended subsequently. Section 6.1 analyzes the merging of Node Templates, Section 6.2 does the same for Relationship Templates and Section 6.3 extends the merging concept and algorithms into the context of Group Templates.

6.1 Merging of Node Templates

6.1.1 Analysis of the Basic Case and its Derivations

Fig. 6.1 picks up the constellation from Fig. 5.1.

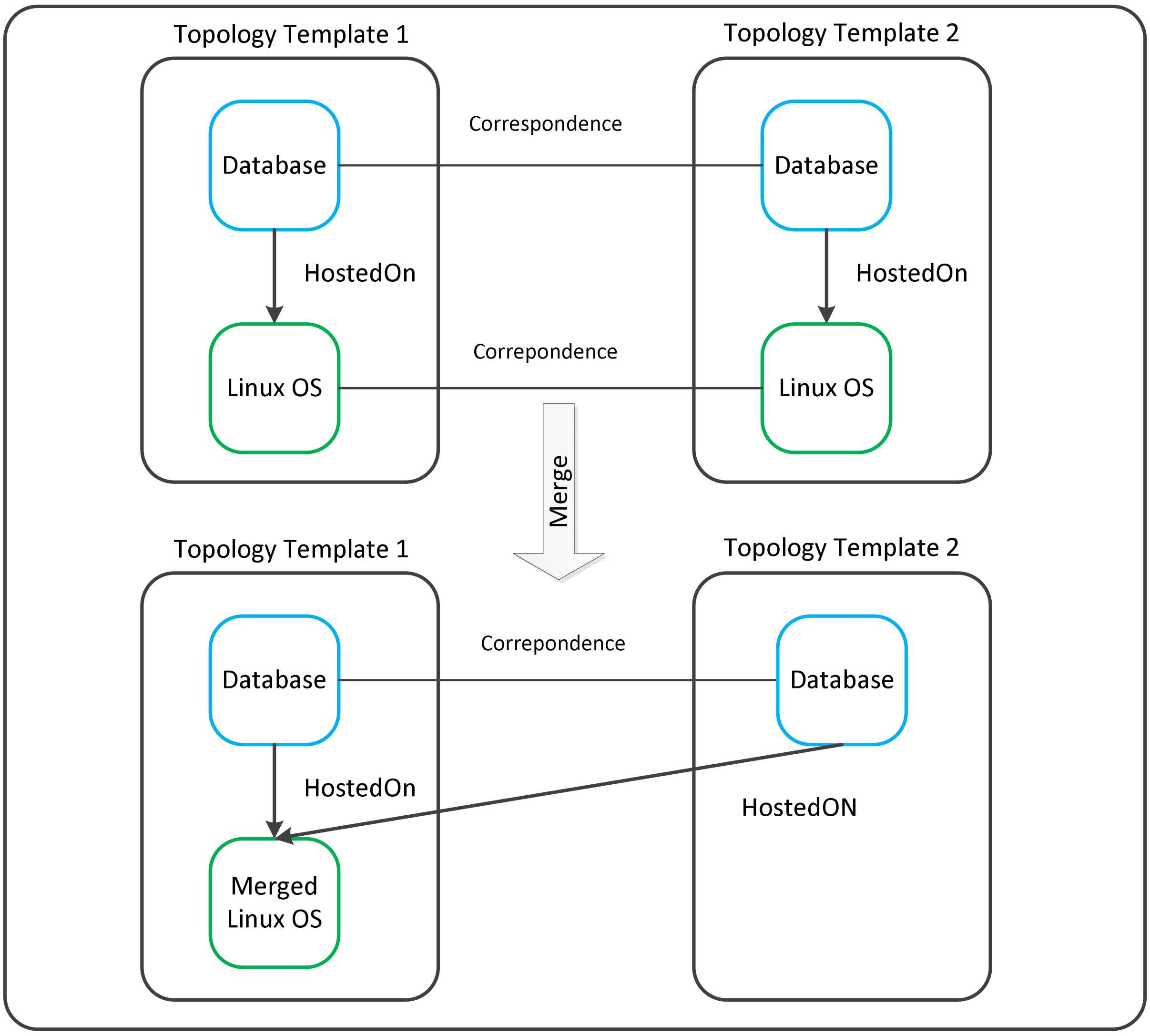


Fig. 6.1: Basic merging case

Two Node Template Correspondences have been found when matching the Node Templates. After conducting one merging step, i.e. merging the two OS Node Templates the result can be seen on the bottom of Fig. 6.1. The *OS* Node Templates are deleted in both Topology Templates and have been added to Topology Template 1. The target of the right-hand side HostedOn Relationship Template has been reassigned to the merged Node Template. Note that the physical location of the reassigned Relationship Template is still Topology Template 2 after the first merging step. It will be merged with the left-hand side Relationship Template in the next step and added to Topology Template 1.

Basic Merging Algorithm

In Listing 6.1 we can see the basic algorithm for the merging of Node Templates. Later the basic algorithm will be extended to include Group Template Hierarchy information. The depicted function is called *performNodeTemplateMerge* (line 5); another function called *performGroupTemplateMerge* will be shown later. The algorithm requires three input parameters: a mapping M calculated before and two Topology Templates denoted by TT_1 and TT_2. The merged Topology Template TT_{merged} forms the output parameter. The loop beginning in line 6 iterates over all Node Template Correspondences of the mapping M. In doing so, the first step is to retrieve the two Node Templates involved, denoted by n_1 and n_2 and stored in the respective Correspondence as variables $from$ and to. If the condition of line 7 evaluates to true, the correspondence has an identical source and target. This issue can arise in a later iteration of the main loop when some Node Templates have already been merged and Node Template Correspondences have been reconnected. In this case the algorithm will skip one iteration. It is revisited further down when discussing the *reconnectCorrespondences* function. Subsequently, another subroutine is called that determines whether one or both Node Templates have already been merged with other Node Templates and whether these Node Templates also had Correspondences that allow for the merging of n_1 and n_2. The details of the *hasCorrectNumberOfCorrespondences* subroutine are discussed further below. If the returned Boolean value equals false, the respective Correspondence is not processed.

Line 17 depicts the creation of a Node Template merger subject to the Node Types of Node Template n_1 and n_2. The merger contains the functionality to merge the two Node Templates of the same Node Type or Node Types related to each other via an inheritance tree. Line 18 shows the actual invocation of the merge function of the respective Node Template merger and the creation of a newly merged Node Template n_{merged}.The details of the merge function will be discussed below in Section 6.1.2. The lines 19-23 are necessary to handle the merging of Node Templates that have already been merged with another Node Template due to other already processed Node Template Correspondences. The four lines of code will be picked up again below. The basic merging algorithm also contains a subroutine to handle all the determined Relationship Template Correspondences. A special, so-called *RelationTemplateshipMergingHandler* is created that contains all the necessary functions to merge Relationship Templates incident to n_1 and n_2 (line 25-26). In Section 6.2 the subject of merging Relationship Templates is discussed in detail.

```
Basic merging algorithm for Node Templates
Input: Mapping M
Input: Topology Template $TT_1$
Input: Topology Template $TT_2$
Output: Topology Template $TT_{merged}$
performNodeTemplateMerge(M,$TT_1$,$TT_2$)
  for each Correspondence c $\in$ M do
    if c.getFrom() == c.getTo() then
      continue
    end if
    Node Template $n_1$ = c.getFrom()
    Node Template $n_2$ = c.getTo()

    if $\neg$hasCorrectNumberOfCorrespondences(c,$n_1$,$n_2$) then
      continue
    end if

    Merger merger = createNodeTemplateMerger(Node Types of $n_1$,$n_2$)
    Node Template $n_{merged}$ = merger.merge($n_1$,$n_2$)
    $n_{merged}$.setNumberOfMerges($n_1$.getNumberOfMerges + 1)
    $n_{merged}$.addToCollectedCorrespondences(
      $n_1$.getCollectedCorrespondences)
    $n_{merged}$.addToCollectedCorrespondences($n_1$.getCorrespondences)
    $n_{merged}$.addToCollectedCorrespondences($n_2$.getCorrespondences)

    Handler handler = createRelationshipTemplateMergingHandler()
    handler.handleRelationshipTemplates($TT_1$,$TT_2$,c,M)

    Set removeSet = reconnectEdges($TT_1$,$TT_2$,$n_1$,$n_2$,$n_{merged}$)
    removeSet.clear

    $TT_1$.remove($n_1$)
    $TT_2$.remove($n_2$)
    $TT_1$.add($n_{merged}$)

    M = reconnectCorrespondences(M,$n_1$,$n_2$,$n_{merged}$)
  end for each
    $TT_1$.addAll($TT_2$)
  return $TT_1$
end
```

Listing 6.1: Function performNodeTemplateMerge

In line 28 the invocation of the subroutine *reconnectEdges* is shown. It reallocates edges of the TOSCA graph, i.e. the Relationship Templates that begin or end at the Node Templates n_1 and n_2 to the new Node Template n_{merged}. The detailed algorithm will be discussed in the next paragraph. ReconnectEdges also returns a set of Relationship Templates that start and end at the same Node Template, i.e. form a loop, due to the merging process, and can be removed (line 26). The lines 31-33 illustrate that n_1 and n_2 are removed from their respective Topology Templates and Node Template n_{merged} is added to Node Template 1. The last subroutine is invoked in line 35 to reconnect Node Template Correspondences that begin or end at n_1 and n_2 with n_{merged}. The specific algorithm is also shown below.

After all Node Template Correspondences have been processed that remaining content of TT_2, which was not touched by the merging, is unified with TT_1 (line 37). This is stipulated by the *element* respective *relationship preservation* requirement. However, this step may not be executed in case of inside merging as TT_1 and TT_2 are identical and the unification would double the containing Elements.[13] This is postulated by the *extraneous item prohibition* requirement of Section 4.2.

Line 38 returns the merged Topology Template, i.e. Topology Template TT_1 after all Node Template Correspondences have been processed.

Subroutine reconnectEdges

The following section discusses the details of the *reconnectEdges* function depicted in Listing 6.2. The function is necessary to reconnect the edges of the TOSCA graph after two nodes have been merged. In TOSCA terminology that means that the source and target elements of Relationship Templates need to be reconnected with the merged Node Template n_{merged} if they previously pointed to one of the Node Templates n_1 or n_2. The reallocation of the Relationship Templates is necessary, as the main algorithm deletes the unmerged Node Templates n_1 or n_2 and adds n_{merged} to Topology Template TT_1 (see Listing 6.1, line 31-33).

The function *reconnectEdges* requires five input parameters and produces one output. The input parameters are: The two Topology Templates denoted by TT_1 and TT_2, the two Node Templates under consideration denoted by n_1 and n_2, and the merged Node Template denoted by n_{merged}. The output parameter is a set containing obsolete Relationship Templates denoted by $removeSet$. The function's core is a loop that iterates over all Relationship Templates located in the Topology Templates TT_1 and TT_2 (line 8). The iteration over the union of both sets is inevitable as Node Templates in TOSCA do not know their incident edges, i.e. Relationship Templates directly. Rather, Relationship Templates are "independent" elements that contain references to Node Templates as source and target elements. This is a concession to the XML-nature of TOSCA to avoid the extreme nesting of nodes and edges. Therefore, every Relationship Template of both Topology Templates has to be considered if their source and target elements are affected by the current merging step. In doing so, the source and target Node Templates of the current Relationship Template are retrieved

[13] This evaluation is not shown here due to space limitations.

(line 9 and 10). If the retrieved source Node Template equals either n_1 or n_2, the new source of the current Relationship Template is set to n_{merged} (line 13-15).

```
Algorithm to reconnect the edges after merging nodes
 1  Input: Topology Template $TT_1$
 2  Input: Topology Template $TT_2$
 3  Input: Node Template $n_1$
 4  Input: Node Template $n_2$
 5  Input: Node Template $n_{merged}$
 6  Output: Set removeSet
 7  reconnectEdges($TT_1$,$TT_2$,$n_1$,$n_2$,$n_{merged}$)
 8    for each Relationship Template rt $\in$ $TT_1$ $\cup$ $TT_2$ do
 9      Node Template source = rt.getSource()
10      Node Template target = rt.getTarget()
11      Set removeSet = new Set()
12
13      if source == $n_1$ $\vee$ source == $n_2$ then
14        rt.setSource($N_{merged}$)
15      end if
16
17      if target == $n_1$ $\vee$ target == $n_2$ then
18        rt.setTarget($N_{merged}$)
19      end if
20
21      if rt.getSource() == rt.getTarget() then
22        removeSet.add(rt)
23      end if
24    end for each
25    return removeSet
26  end
```

Listing 6.2: Function reconnectEdges

The same pattern is followed for the target Node Template of the current Relationship Template (lines 17-19). If these reallocations lead to a Relationship Template with identical source and target nodes, the current Relationship Template can be removed, respectively, added to the previously created *removeSet*. This issue can occur when two Node Templates are merged having a Relationship Type with Communication semantics between them. The possibly empty *removeSet* is returned in line 25.

Function reconnectCorrespondences

Listing 6.3 contains the algorithm of the function *reconnectCorrespondences*. It is necessary to handle Node Templates that have more than one incoming or outgoing Node Template Correspondence. Similar to the reconnection of the edges after the merging of two Node

Templates to the newly merged Node Template, Node Template Correspondences must also be reconnected from the deleted Node Templates to the merged results.

```
Algorithm to reconnect Node Template Correspondences
1   Input: Mapping M
2   Input: Node Template $n_1$
3   Input: Node Template $n_2$
4   Input: Node Template $n_{merged}$
5   Output: Mapping M
6   reconnectCorrespondences(M,$n_1$,$n_2$,$n_{merged}$)
7     for each Correspondence c $\in$ M do
8
9       if c.getFrom() == $n_1$ $\vee$ c.getFrom() == $n_2$ then
10        c.setFrom($n_{merged}$)
11      end if
12
13      if c.getTo() == $n_1$ $\vee$ c.getTo() == $n_2$ then
14        c.setTo($n_{merged}$)
15      end if
16    end for each
17    return M
18  end
```

Listing 6.3: Function reconnectCorrespondences for Node Templates

The function has three input and one output parameters. The input parameters are a mapping M, the two Node Templates n_1 and n_2, and the merged Node Template n_{merged}. The output is the corrected mapping M. For each Node Template Correspondence $c \in M$ two cases have to be evaluated: If the $from$ variable equals n_1 or n_2, it is set to n_{merged} (line 9-11) The same holds true for the to variable, i.e. the target of the correspondence c (line 13-15). The corrected mapping M is returned after the last iteration (line 17).

The reconnection of Node Template Correspondences is necessary since the original nodes n_1 and n_2 get deleted from their Topology Templates in the main function. Those correspondences of M which are not yet processed would then use Node Templates that no longer exist and cause failures. It is possible that the function *reconnectCorrespondences* causes pending correspondences to point to the same source and target due to prior merging iterations. In this case the main algorithm discards the respective correspondence (line 7-9). However, possible attached Relationship Template Correspondences must still be considered.

Correspondences to More Than one Node Template

Usually the mapping includes more than one Node Template Correspondence outgoing from a particular Node Template. Fig. 6.2 shows four generic Node Templates where every single one matches every other Node Template. This is called a *full mapping* in this thesis.

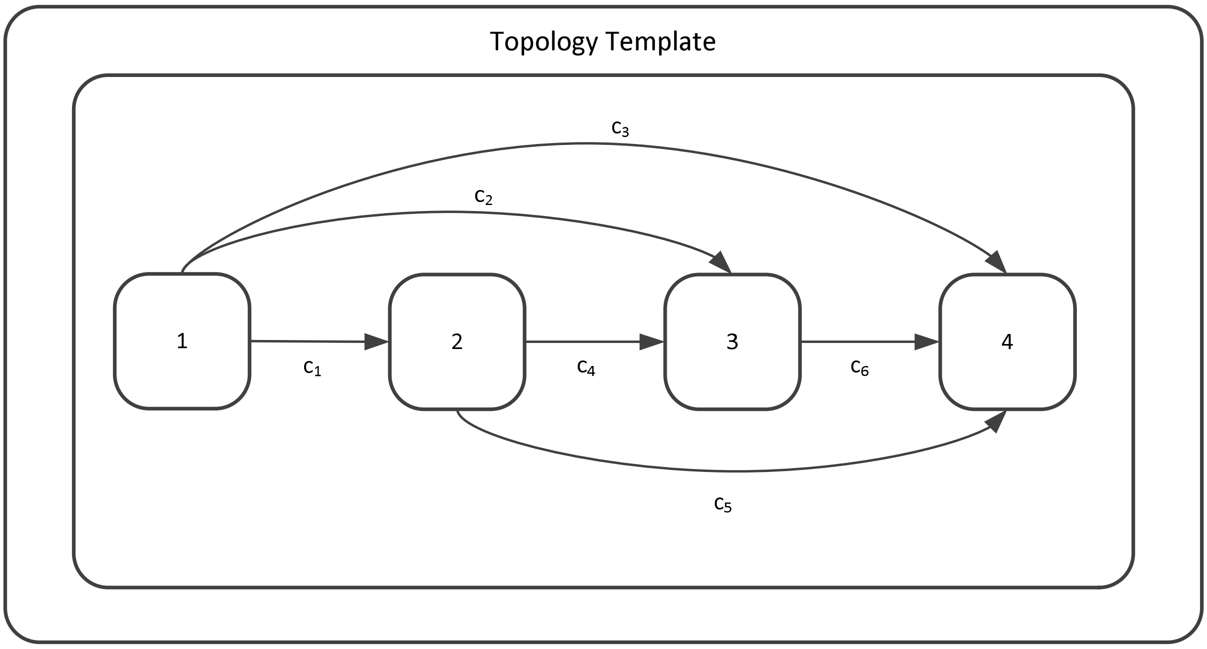


Fig. 6.2: Full mapping between all Node Templates

Note that the Node Template Correspondences are now depicted as directed edges in contrast to the previously shown matching case figures. The reason is that for the processing of the found Node Template correspondences the direction of the arcs must be considered in order to develop a merging algorithm covering all situations.

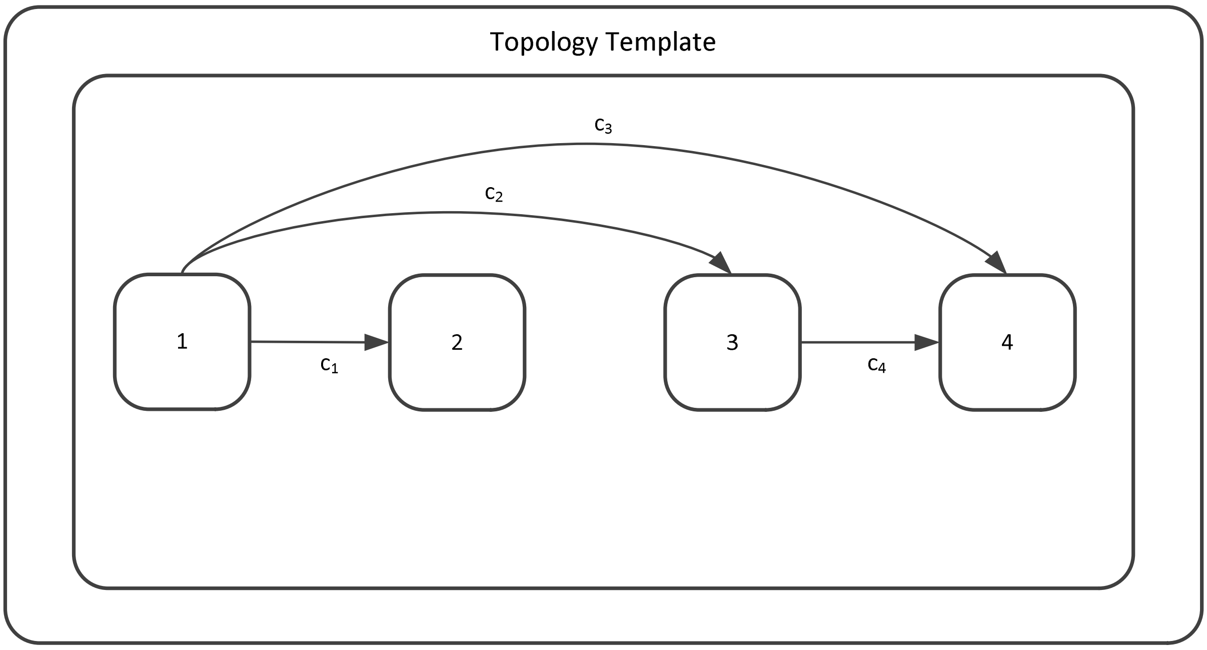


Fig. 6.3: Partial mapping between Node Templates

Furthermore, the merging algorithm must cope with missing Node Template Correspondences. In Fig. 6.3 we can see what is called a *partial mapping* between the Node Templates in this work. Node Template 2 is incompatible with Node Template 3 and 4 and vice versa. Therefore, if e.g. the algorithm starts merging Node Template 1 with Node Template 2 it may not merge with 3 and 4 in order to preserve a valid result. It is also forbidden to merge the unified Node Templates 1, 3 and 4 with Node Template 2. This makes clear that merged Node Templates must preserve their assigned Node Template Correspondences helping the algorithm to decide if an already merged Node Template also can be merged with further Node Templates.

As we can see in Fig. 6.4 the Node Template Correspondences can also point against the general comparison and matching direction. This is a phenomenon that surfaces when matching Topology Templates including Group Templates. When the matching algorithm steps recursively into a Group Template it may happen that a correspondence is established in the way as pointing from Node Template 4 to 3 and 2 against the direction of the other correspondences. A robust merging algorithm must cope with different directions as well as arbitrary Node Template Correspondences as a starting point of the merging process. It should be irrelevant if the algorithm starts with the correspondence from 1 to 2 or with 4 to 2 in order to produce a correct result. [14]

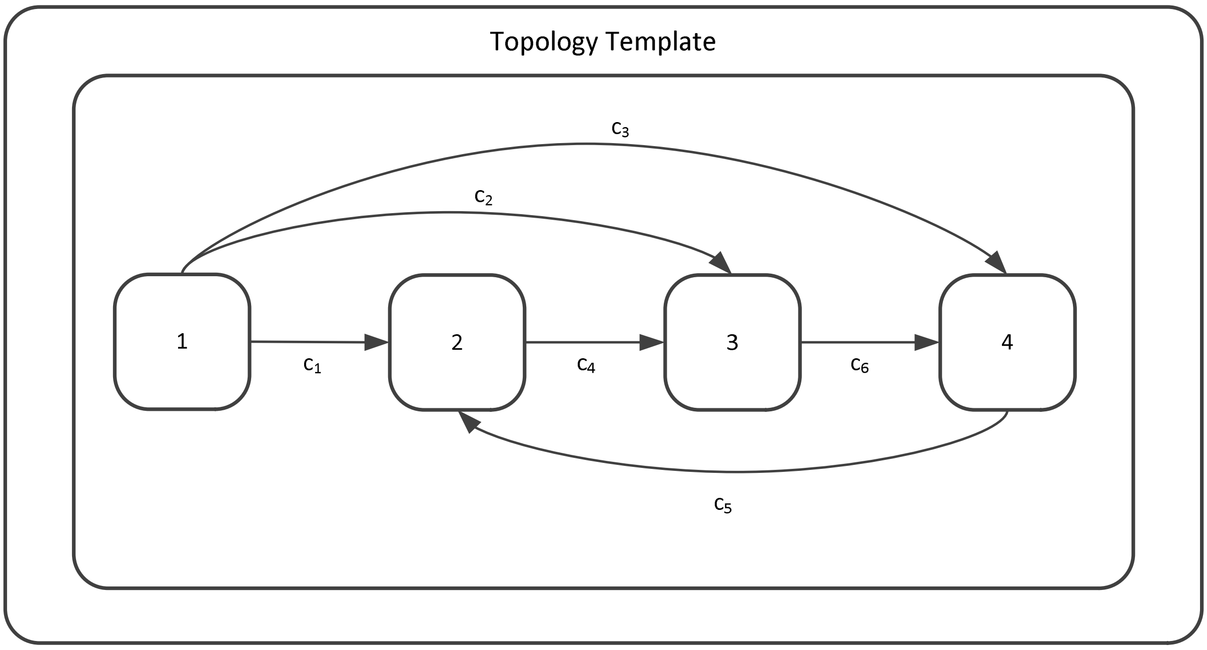


Fig. 6.4: Full mapping but inverted directions

Listing 6.4 contains the first part of the algorithm that contributes to the solution of the merging cases discussed above. The function *hasCorrectNumberOfCorrespondences* is necessary to deal with the presented partial mappings in Fig. 6.3. In Section 5.1.1 we have seen that every Node Template that matches another Node Template, i.e. that is the source of a Node Template Correspondence, stores that particular Correspondence in addition to the

[14] However, for finding a global optimum the order of correspondence processing is important. See Section 9.2.1 for a discussion of this.

insertion of the Correspondence in the mapping set. The author of this work calls these Correspondences *innate*. The target Node Template does not store that particular Correspondence.

The approach for handling partial mappings is based on the observation that when preserving the stored Node Template Correspondences of both Node Templates during a merging, these Correspondences can be used in a succeeding merging step to determine whether two particular Node Templates are allowed to be merged or not. Let us therefore revisit the lines 19-23 of the basic merging algorithm for Node Templates displayed in Listing 6.1. In order to track how often a particular Node Template has already been merged with a second Node Template, each Node Template has a counter variable named *numberOfMerges*. The variable of NodeTemplate n_{merged} takes the following value (line 19):

$$n_{merged}(numberOfMerges) = \begin{cases} n_1(numberOfMerges) + 1, if\ n_1(numberOfMerges) > 0 \\ 1, \quad if\ n_1(numberOfMerges) = 0 \end{cases}$$

That means the variable is incremented by 1 after every merging step based on Node Template n_1's variable value.

Furthermore, the merged NodeTemplate n_{merged} stores the *innate* Node Template Correspondences having Node Template n_1 respectively n_2 as source in a set called *collectedCorrespondences* (lines 22 and 23). Additionally, the already collected Node Template Correspondences from Node Template n_1 are added to the *collectedCorrespondences* list (line 20). To illustrate the behavior of the four discussed pseudo code lines consider once again Fig. 6.3. If e.g. the Node Templates 1 (resembling n_1) and 3 (resembling n_2) are merged first, the resulting Node Template would store the Node Template Correspondences c_1, c_2 and c_3 as outgoing Node Template Correspondences from n_1 and c_4 as outgoing Node Template Correspondence from n_2. As Node Template 1 has not yet been merged with another Node Template, its *collectedCorrespondences* will be empty. Furthermore, when merging the resulting Node Template n_{merged} in the second step with Node Template 4, the newly merged Node Template would collect all previously collected Node Template Correspondences from n_{merged} but not collecting any Correspondences from Node Template 4.

The *innate* and *collected* Node Template Correspondences of two Node Templates are evaluated for every processed Node Template Correspondence of the mapping. Listing 6.4 depicts the corresponding algorithm. It requires three input parameters: two Node Templates and the Node Template Correspondence, called initial Correspondence, which is currently processed by the main loop of the basic Node Template merging algorithm of Listing 6.1. The return value of the depicted function is of type Boolean and indicates if the merging of the particular Node Templates may continue. Line 6 contains a first condition: If both Node Templates n_1 and n_2 have not yet been subject to a merging step, i.e. both *numberOfMerges* variables are 0, then the algorithm returns true.

The rest of the algorithm has the following pattern: the *innate* and the *collected* Node Template Correspondences that point to the same target Node Template as the initial Node

Template Correspondence $corr_{init}$ and are not identical with $corr_{init}$ are counted. In other words, the first part of the *hasCorrectNumberOfCorrespondences* function counts all additional Node Template Correspondences that exist between the two Templates n_1 and n_2.

```
Algorithm to check if number of Correspondences is correct part 1
Input: Initial correspondence $corr_{init}$
Input: Node Template $n_1$
Input: Node Template $n_2$
Output: boolean
hasCorrectNumberOfCorrespondences($corr_{init}$,$n_1$,$n_2$)
  if $n_1$.getNumberOfMerges() == 0 $\wedge$ $n_2$.getNumberOfMerges() == 0 then
    return true
  end if

  int counter = 0

  for each innate Correspondence $c_1$ outgoing from $n_1$ do
    if $c_1$.getTo() == $corr_{init}$.getTo() $\wedge$ $c_1$ != $corr_{init}$ then
      counter = counter + 1
    end if
  end for each

  for each collected Correspondence $cc_1$ outgoing from $n_1$ do
    if $cc_1$.getTo() == $corr_{init}$.getTo() $\wedge$ $cc_1$ != $corr_{init}$ then
      counter = counter + 1
    end if
  end for each

  for each innate Correspondence $c_2$ outgoing from $n_2$ do
    if $c_2$.getTo() == $corr_{init}$.getTo() $\wedge$ $c_2$ != $corr_{init}$ then
      counter = counter + 1
    end if
  end for each

  for each collected Correspondence $cc_2$ outgoing from $n_2$ do
    if $cc_2$.getTo() == $corr_{init}$.getTo() $\wedge$ $cc_2$ != $corr_{init}$ then
      counter = counter + 1
    end if
  end for each
    ...
```

Listing 6.4: Function hasCorrectNumberOfCorrespondences part 1

In Listing 6.5 the second part of the function *hasCorrectNumberOfCorrespondences* is depicted. It utilizes the fact that in case of a *full mapping* between all Node Templates with identical or compatible Node Types, the number Node Templates Correspondences between any two of them can be determined unambiguously. At the same time, the direction of the Node Template Correspondence is not important.

Definition 6.1 (Number of additional necessary Node Template Correspondences): *Let $n_{nec\_add\_corr}$ be the number of necessary Node Template Correspondences between two Node Templates n_1 and n_2 in case of a full mapping. Furthermore, let $C_{n_1}^{out}$ be the set of outgoing Node Template Correspondences from Node Template n_1, both innate and collected, and $C_{n_2}^{out}$ be the set of outgoing Node Template Correspondences from Node Template n_2, both innate and collected. For $n_{nec\_add\_corr}$ the following holds true under the assumption that one or both Node Templates have already been merged*[15]:

$$n_{nec\_add\_corr} = \begin{cases} \left|C_{n_1}^{out}\right| + \left|C_{n_2}^{out}\right| + 1, & if\ n_1(numberOfMerges) > 0\ and\ n_2(numberOfMerges) > 0 \\ \left|C_{n_1}^{out}\right|, & if\ n_1(numberOfMerges) > 0\ and\ n_2(numberOfMerges) = 0 \\ \left|C_{n_2}^{out}\right|, & if\ \text{if}\ n_1(numberOfMerges) = 0\ and\ n_2(numberOfMerges) > 0 \end{cases}$$

Part 2 of the *hasCorrectNumberOfCorrespondences* function in Listing 6.5 uses the calculated value of the variable *counter* as indicator for the accumulated cardinality of the sets $C_{n_1}^{out}$ and $C_{n_2}^{out}$ and covers the three cases discussed above. If the actual value deviates from $n_{nec\_add\_corr}$, it can be deduced that no *full mapping* has been found and at least one of the Node Templates that was merged earlier and is now a part of n_1 or n_2 did not have a Node Template Correspondence to the currently processed Node Templates n_1 or n_2. Otherwise a suitable Node Template Correspondence would have been element of $C_{n_1}^{out}$ or $C_{n_2}^{out}$. Thus, n_1 or n_2 must not be merged and the function returns false.

[15] Otherwise the algorithm would have returned true beforehand. (see Listing 6.4 line 6, 7)

```
Algorithm to check if number of Correspondences is correct part 2
      ...
    if $n_1$.getNumberOfMerges() > 0 $\wedge$ $n_2$.getNumberOfMerges() > 0 then
      if counter == $n_1$.getNumberOfMerges() + $n_2$.getNumberOfMerges() +1
      then
        return true
      else
        return false
      end if
    end if

    if $n_1$.getNumberOfMerges() > 0 $\wedge$ $n_2$.getNumberOfMerges() == 0 then
      if counter == $n_1$.getNumberOfMerges then
        return true
      else
        return false
      end if
    end if

    if $n_1$.getNumberOfMerges() == 0 $\wedge$ $n_2$.getNumberOfMerges() > 0 then
      if counter == $n_2$.getNumberOfMerges then
        return true
      else
        return false
      end if
    end if
  end
```

Listing 6.5: Function hasCorrectNumberOfCorrespondences part 2

6.1.2 Merging of Properties on the Node Template Level

In this section a discussion of the actual merging of two Node Templates follows. In particular, this includes the assessment of the functionality the TOSCAMerge framework has to provide inside the *merge* subroutine of the main algorithm previously shown in Listing 6.1. In Section 5.1.2 we have seen that in addition to the Node Types of two Node Templates, several properties have to be evaluated. These properties are *MinInstances*, *MaxInstances* *PropertyDefaults*, *PropertyConstraints*, *Policies*, *EnvironmentConstraints*, *DeploymentArtifacts* and *ImplementationArtifacts*. The consideration of all properties is a requirement stipulated in Section 4.2 (*property preservation* requirement).

First, the author of this work will briefly review the merge function depicted in Listing 6.6, which is essentially a sequence of subroutine-calls that handle the merging of the different properties. Subsequently, the merging idiosyncrasies of the different Node Template Properties will be analyzed and the individual subroutines handling the merging will be covered.

Similar to the matching cases in Chapter 5, the TOSCAMerge framework will provide any merging operations that can be handled generically. However, when some properties cannot be merged automatically the framework provides the possibility to trigger Node Type specific plugins. These plugins are invoked and configured corresponding to the qualified name of the involved Node Types. The generic merging parts of the framework adhere to the proposed requirement of *value preference* and take the values from the left-hand side Node Template preferably. However, each generic function can be overridden in the type-specific plugin if the default behavior is not appropriate.

The *merge* function in Listing 6.6 has two input parameters: two Node Templates denoted by n_1 and n_2. The function's output is a merged Node Template resulting from n_1 and n_2 and denoted by n_{merged}.The function creates the new Node Template n_{merged} and sets n_1's *name* and *id*. (line 5-7). Thereupon, subroutine-calls provide the merged properties that are set to n_{merged}. They are reviewed below.

```
Algorithm to merge the Properties of two Node Templates
1   Input: Node Template $n_1$
2   Input: Node Template $n_2$
3   Output: Node Template $n_{merged}$
4   merge($n_1$,$n_2$)
5     Node Template $n_{merged}$ = new Node Template()
6     $n_{merged}$.setName($n_1$.getName())
7     $n_{merged}$.setId($n_1$.getId())
8     $n_{merged}$.setNodeType(decideNodeType(Node Types of $n_1$,$n_2$)
9     $n_{merged}$.setMinInstances(mergeMinInstances($n_1$,$n_2$))
10    $n_{merged}$.setMaxInstances(mergeMaxInstances($n_1$,$n_2$))
11    $n_{merged}$.setPropertyDefaults(mergePropertyDefaults($n_1$,$n_2$))
12    $n_{merged}$.setPropertyConstraints(mergePropertyConstraints($n_1$,$n_2$))
13    $n_{merged}$.setPolicies(mergePolicies($n_1$,$n_2$))
14    $n_{merged}$.setEnvironmentConstraints(
15      mergeEnvironmentConstraints($n_1$,$n_2$))
16    $n_{merged}$.setDeploymentArtifacts(mergeDeploymentArtifacts($n_1$,$n_2$))
17    $n_{merged}$.setImplementationArtifacts(
18      mergeImplementationArtifacts($n_1$,$n_2$))
19
20    return $n_{merged}$
21  end
```

Listing 6.6: Function merge for Node Templates

Node Types

If the Node Templates to be merged have and identical Node Type, the decision which Node Type to assign to the merged Node Template can be made by the TOSCAMerge framework. However, if a Node Template Correspondence between Node Types, somehow related by an inheritance tree, has been found, the decision which Node Template to preserve cannot be made generically. Rather, a type-specific plugin must be called to decide which Node Type to assign to the merged Node Template.

Listing 6.7 shows the algorithm of *decideNodeType*. If both NodeTypes are not identical the plugin is called by the subroutine *decideNodeTypeTypeSpecificContent*.

```
Algorithm to determine which Node Type to use
Input: Node Type $nt_1$
Input: Node Type $nt_2$
Output: Node Type $nt_{merged}$
decideNodeType($nt_1$,$nt_2$)
  if $nt_1$ == $nt_2$ then
    return $nt_1$
  else
    return decideNodeTypeTypeSpecificContent($nt_1$,$nt_2$)
  end if
end
```

Listing 6.7: Function decideNodeType

MinInstances and MaxInstances

The merging of the MinInstances and MaxInstances properties can be handled generically by the TOSCAMerge framework. However, users of the framework will be able to override the generic functionality and provide their own implementation. The merge is executed by the function *mergeMinInstances* respectively *mergeMaxInstances*. The algorithm of the functions is not shown here, it essentially sums up the values in each case.

PropertyDefaults

Merging the PropertyDefaults of two Node Templates meets the same problems as the matching. The TOSCAMerge framework is unable to understand the exact semantics of each XML element generically. Therefore, a type-specific plugin must handle the merging of the values. The framework itself provides facilities to easily manipulate, i.e. read and write, the XML fragments representing the PropertyDefaults. The *mergePropertyDefaults* function must only retrieve the PropertyDefaults from each Node Template and invoke a type-specific plugin that handles the actual merge. Listing 6.8 shows a simple example how the type-specific algorithm could look like. It extracts a heap size value, e.g. from an application server, out of both PropertyDefaults using a framework functionality and the tag name of the XML element (indicated by inverted comma). The simple merging strategy in this case adds the two heap sizes. Subsequently, the added value is written back.

```
Type specific algorithm to merge simple PropertyDefaults
Input: PropertyDefaults PD$_1$
Input: PropertyDefaults PD$_2$
Output: Property Defaults PD$_{merged}$
mergePropertyDefaultsTypeSpecificContent(PD$_1$,PD$_2$)
  int heapSize1 = getNodeValueByTagName(PD$_1$,“HeapSize”)
  int heapSize2 = getNodeValueByTagName(PD$_2$,“HeapSize”)

  heapSize1 = heapSize1 + heapSize2

  setNodeValueByTagName(PD$_1$,“HeapSize”,heapSize1)
  return PD$_1$
end
```

Listing 6.8: Example for type-specific PropertyDefaults merging

PropertyConstraints

In contrast to the matching of two set of *PropertyConstraints*, the merging can be achieved in a generic way. The TOSCAMerge framework provides an appropriate default algorithm; however, it can be overridden if a user has a more suitable merging strategy. Listing 6.9 shows the function *mergePropertyConstraints*. The function has two input parameters, the Node Templates to be merged, denoted by n_1 and n_2. The output parameter is a set of Property Constraints, denoted by PC_{merged}.

First, it is evaluated if one of the PropertyConstraints sets is empty. If so, the corresponding other is returned (line 5-11). Otherwise the two PropertyConstraints sets PC_1 and PC_2 are retrieved (line 13-14) and every PropertyConstraint of PC_2.is checked against the elements of PC_1. If it is not yet element of PC_1, it is added to latter. The comparison and adding of the PropertyConstraint elements is possible, as the TOSCA Specification states that Property-Constraints with the same *Property* and *ConstraintType* attributes are identical.

```
Algorithm to merge the Property Constraints
Input: Node Template $n_1$
Input: Node Template $n_2$
Output: Property Constraints $PC_{merged}$
mergePropertyConstraints($n_1$,$n_2$)
  if $n_1$.getPropertyConstraints() == null then
    return $n_2$.getPropertyConstraints()
  end if

  if $n_2$.getPropertyConstraints() == null then
    return $n_1$.getPropertyConstraints()
  end if

  Property Constraints $PC_1$ = $n_1$.getPropertyConstraints()
  Property Constraints $PC_2$ = $n_2$.getPropertyConstraints()

  for each Property Constraint pc $\in$ $PC_2$ do
    if pc $\notin$ $PC_1$ then
      $PC_1$.add(pc)
    end if
  end for each
  return $PC_1$
end
```

Listing 6.9: Function mergePropertyConstraints

Policies, EnvironmentConstraints, DeploymentArtifacts and ImplementationArtifacts

The remaining Property elements of a Node Template show the same pattern as seen in connection with the PropertyConstraints. A generic implementation can be provided, but it can easily be overridden if a different merging approach is more suitable. The algorithms follow the same idea as Listing 6.9, i.e. checking against a set if a particular element is already element of it. Under certain circumstances this simple approach might not suffice, e.g. if it has to be decided which DeploymentArtifacts and ImplementationArtifacts for non-identical but related Node Types are to be used furthermore.

6.2 Merging of Relationship Templates

In Section 5.2 the specifics of finding Correspondences between Relationship Templates, denoted by Relationship Template Correspondences, have been studied. In this section the attention is turned to the usage of the found Correspondences, i.e. the merging of Relationship Templates. As cases do not differ significantly from the Node Template merging they are not analyzed again. Instead an algorithm is proposed in 6.2.1. The consideration of their relative position in Group Templates is not analyzed here but is part of Section 6.3. Section 6.1.2 deals with the unification Relationship Template properties.

6.2.1 Basic Merging Algorithm for Relationship Templates

Listing 6.10 shows the basic merging algorithm for Relationship Templates. Later an extended variation will be discussed that also includes a consideration of the position in a Group Template Hierarchy. The function *handleRelationshipTemplates* picks up the subroutine invocation of line 26 of the basic merging algorithm for Node Templates (Listing 6.1). It is implemented in a so-called *RelationshipTemplateMergingHandler*, a concept similar to the *RelationshipTemplateMatchingHandler* of Section 5.2.1 with the difference that the *RelationshipTemplateMergingHandler* is generic for all Relationship Types.

The merging algorithm for Relationship Templates is very similar to the one for merging Node Templates. Therefore, subroutines that are basically identical will not be reviewed again. The algorithm requires four input parameters: two Topology Templates denoted by TT_1 and TT_2, a set of Node Template Correspondences, i.e. a mapping M, and a Node Template Correspondence denoted by c. The result of the algorithm is an updated Topology Template denoted by $TT_{updated}$. The first step of the algorithm is to retrieve the attached Relationship Template Correspondences (line 7). Subsequently, the algorithm's main loop iterates over all Relationship Template Correspondences. Even though the Relationship Templates concerned may have different Relationship Types, and therefore, different semantics, they are treated uniformly (with the exception of the actual merge of their properties). In line 10-12 it is checked if the Relationship Template Correspondence has the same head and tail, i.e. forms a loop. If so, the processing of the particular Correspondence is skipped. The reason for this issue is the reallocation of the Relationship Template Correspondences to already merged Relationship Templates, just as in the case of Node Template Correspondences. Subsequently, the two Relationship Templates to be merged, denoted by r_1 and r_2, are retrieved from the current Relationship Template (line 14-15). The *hasCorrectNumberOfCorrespondences* subroutine line 17 works exactly the same way as described above in the context of Node Template Correspondences; therefore, it is not shown once again. It has the task of checking if Relationship Templates already merged into new Relationship Templates also had the necessary correspondences to the current two Relationship Templates.

Subject to the Relationship Type of r_1 a merger is created that contains the functionality for merging the properties of two Relationship Templates (line 21). The next step is the actual invocation of the *merge* function for unifying two Relationship Templates properties. See the next section for a detailed review.

```
Basic merging algorithm for Relationship Templates
 1  Input: Topology Template $TT_1$
 2  Input: Topology Template $TT_2$
 3  Input: Mapping M
 4  Input: Node Template Correspondence c
 5  Output: Topology Template $TT_{updated}$
 6  handleRelationshipTemplates($TT_1$,$TT_2$,c,M)
 7    Set RC = c.getRelationshipTemplateCorrespondences()
 8
 9    for each Relationship Template Correspondence rc $\in$ RC do
10      if rc.getFrom() == rc.getTo() then
11        continue
12      end if
13
14      Relationship Template $r_1$ = rc.getFrom()
15      Relationship Template $r_2$ = rc.getTo()
16
17      if $\neg$hasCorrectNumberOfCorrespondences(rc,$R_1$,$R_2$) then
18        continue
19      end if
20
21      Merger merger = createRelTemplateMerger(Relationship Type of $r_1$)
22      Relationship Template $r_{merged}$ = merger.merge($r_1$,$r_2$)
23      $r_{merged}$.setNumberOfMerges($r_1$.getNumberOfMerges + 1)
24      $r_{merged}$.addToCollectedCorrespondences(
25        $r_1$.getCollectedCorrespondences)
26      $r_{merged}$.addToCollectedCorrespondences($r_1$.getCorrespondences)
27      $r_{merged}$.addToCollectedCorrespondences($r_2$.getCorrespondences)
28
29      $TT_1$.remove($r_1$)
30      $TT_2$.remove($r_2$)
31      $TT_1$.add($r_{merged}$)
32
33      M = reconnectCorrespondences(M,$r_1$,$r_2$,$r_{merged}$)
34    end for each
35    return $TT_1$
36  end
```

Listing 6.10: Function handleRelationshipTemplates

The lines 23-27 are also very similar to the Node Template merging counterpart. Relationship Correspondences are collected to handle possible partial mappings between the Relationship Templates incident to two corresponding Node Templates. After these steps r_1 and

r_2 are removed from their respective Topology Template and the merged Relationship Template r_{merged} is added to Topology Template 1 (lines 29-31).

The *reconnectCorrespondences* function for Relationship Templates, invoked in line 33, differs from the Node Template counterpart. Listing 6.11 shows this function. The main difference to Listing 6.3 is the nested loops. In order to correct every Template Relationship Correspondence, an iteration over all existing Node Template Correspondences and their nested Relationship Correspondences has to take place.

```
Algorithm to reconnect Relationship Template Correspondences
Input: Mapping M
Input: Relationship Template $r_1$
Input: Relationship Template $r_2$
Input: Relationship Template $r_{merged}$
Output: Mapping M
reconnectCorrespondences(M,$r_1$,$r_2$,$r_{merged}$)
  for each Node Template Correspondence c $\in$ M do
    Set RC = c.getRelationshipTemplateCorrespondences()

    for each Relationship Template Correspondence rc $\in$ RC do
      if rc.getFrom() == $r_1$ $\vee$ rc.getFrom() == $r_2$ then
        rc.setFrom($r_{merged}$)
      end if

      if rc.getTo() == $r_1$ $\vee$ rc.getTo() == $r_2$ then
        rc.setTo($r_{merged}$)
      end if
    end for each
  end for each
  return M
end
```

Listing 6.11: Function reconnectCorrespondences for Relationship Templates

6.2.2 Merging of Relationship Template Properties

Listing 6.12 shows the merging function for Relationship Templates and their Properties which is invoked in the basic merging algorithm for Relationship Templates in Listing 6.10, line 22. The function requires two Relationship Templates denoted by r_1 and r_2 as input parameters and returns a unified Relationship Template denoted by r_{merged}. The first step of the algorithm is to create a new Relationship Template that holds the unified properties (line 5). *Name*, *Id*, the *Relationship Type* and the *source* and target *elements* are adopted from r_1. The properties that have to be unified are the *PropertyDefaults*, the *PropertyConstraints* and the *RelationshipConstraints*. The subroutines *mergePropertyDefaults*, *mergePropertyConstraints* are very similar to their Node Template counterparts in Listing 6.8 and Listing 6.9.

```
Algorithm to merge the Properties of two Relationship Templates
Input: Relationship Template $r_1$
Input: Relationship Template $r_2$
Output: Relationship Template $r_{merged}$
merge($r_1$,$r_2$)
  Relationship Template $r_{merged}$ = new Relationship Template()
  $r_{merged}$.setId($r_1$.getId())
  $r_{merged}$.setName($r_1$.getName())
  $r_{merged}$.setRelationshipType($r_1$.getRelationshipType())
  $r_{merged}$.setSourceElement($r_1$.getSourceElement())
  $r_{merged}$.setTargetElement($r_1$.getTargetElement())
  $r_{merged}$.setPropertyDefaults(mergePropertyDefaults($r_1$,$r_2$))
  $r_{merged}$.setPropertyConstraints(mergePropertyConstraints($r_1$,$r_2$))
  $r_{merged}$.setRelationshipConstraints(
  mergeRelationshipConstraints($r_1$,$r_2$))

  return $r_{merged}$
end
```

Listing 6.12: Function merge for Relationship Templates

MergePropertyDefaults extracts the relevant XML-fragments from the Relationship Templates and invokes a subroutine called *mergePropertyDefaultsTypeSpecificContent*. This invokes a type-specific plugin, in the same way as in the case of Node Templates. For the merging of PropertyConstraints a build-in algorithm is provided, but it can easily be overridden if desired. At first sight, the only new merging subroutine is *mergeRelationshipConstraints*. However, it also follows the algorithm of Listing 6.9, i.e. two RelationshipConstraints are considered equal if their *ConstraintType* attribute is equal and only those Relationship Constraints from the second set which do not yet exist are transferred to the first set.

6.3 Merging in the Context of Group Templates

The previous concept for merging does not consider Group Templates holding Node, Relationship or other Group Templates. The following section will expand the merging concept by including the merging of Node and Relationship Templates inside and across the boundaries of Group Templates.

6.3.1 Merging of Node Templates

In Listing 6.1 the basic algorithm for merging was proposed. In the lines 31-33 the two Node Templates n_1 and n_2 are deleted from the Topology Templates TT_1 respectively TT_2. In doing so, it was assumed that the Topology Templates contained no Group Templates and all elements resided on Nesting Level 0. To overcome this limitation an extension to the previously proposed algorithm has to be made. The lines 31-33 from Listing 6.1 have to be replaced by the algorithm fragment depicted in Listing 6.13. The fragment adds four case distinctions. The four cases are analyzed and discussed without providing new graphical figures as the constellations of Node Templates on different Nesting Levels have been discussed in Section 5.3 at length. The only difference is that the Correspondences between them have been found now.

The first one from line 5 to line 8 represents the previous case where both Node Templates n_1 and n_2 are not nested inside a Group Template but residing on Nesting Level 0. This is expressed by the evaluation if both Node Templates' parent GroupTemplateHierarchy elements are non-existing, i.e. null. If this case applies, the same steps as in the original algorithm are executed: n_1 is removed from TT_1, n_2 from TT_2 and finally n_{merged} is added to TT_1.

The pseudo code fragment from line 9-15 represents the case where Node Template n_1 resides on Nesting Level 0 and Node Template n_2 on a Nesting Level ≥ 1. The first step is to set the new parent to Node Template n_{merged} using the one from Node Template n_2. Thereupon, the Group Template, denoted by g, which contains Node Template n_2 is retrieved via the parent GroupTemplateHierarchy element of n_2. The content elements (of type tExtensibleElements) of g are then stored in a set, created earlier in line 2. In this set, denoted by E_2, Node Template n_{merged} is inserted while Node Template n_2 is removed. Assuming call-by-reference semantics the Group Template g still has a pointer to the set E_1 and also learns of the manipulation. The last step in this case is to remove Node Template n_1 from Topology Template TT_1. This general approach is also used throughout the other cases: The Group Template on the higher Nesting Level "pulls" the merged Node Template into its region. Node Templates that are merged more than once across Nesting Levels move down to the greatest Nesting Level. A graphical example for the "pulling" can be found below in Fig. 6.5.

The lines 16-22 represent the inverse case where Node Template n_1 is on a Nesting Level ≥ 1 and Node Template n_2 is on Nesting Level 0. The approach is the same as described before.

The last case is shown in the lines 23-38: now both Node Templates are nested inside a Group Template but not necessarily on the same Nesting Level. This time both parent Group Templates, denoted by g_1 and g_2, respectively, are retrieved from the parent GroupTemplateHierarchy element as well as their content elements that are then stored in the sets E_1 and E_2. Subsequently, two subcases have to be considered.

```
Extension to the basic merging algorithm for Node Templates
   ...
 Set $E_1$ = new Set of Elements
 Set $E_2$ = new Set of Elements

 if $n_1$.getParent() == null $\wedge$ $n_2$.getParent() == null then
   $TT_1$.remove($n_1$)
   $TT_2$.remove($n_2$)
   $TT_1$.add($n_{merged}$)
 else if $n_1$.getParent() == null $\wedge$ $n_2$.getParent() $\neq$ null then
   $n_{merged}$.setParent($n_2$.getParent())
   Group Template g = $n_2$.getParent().getGroupTemplate()
   $E_2$ = g.getElements()
   $E_2$.add($n_{merged}$)
   $E_2$.remove($n_2$)
   $TT_1$.remove($n_1$)
 else if $n_1$.getParent() $\neq$ null $\wedge$ $n_2$.getParent() == null then
   $n_{merged}$.setParent($n_1$.getParent())
   Group Template g = $n_1$.getParent().getGroupTemplate()
   $E_1$ = g.getElements()
   $E_1$.add($n_{merged}$)
   $E_1$.remove($n_1$)
   $TT_2$.remove($n_2$)
 else if $n_1$.getParent() $\neq$ null $\wedge$ $n_2$.getParent() $\neq$ null then
   Group Template $g_1$ = $n_1$.getParent().getGroupTemplate()
   Group Template $g_2$ = $n_2$.getParent().getGroupTemplate()
   $E_1$ = $g_1$.getElements()
   $E_2$ = $g_2$.getElements()
   if $n_1$.getParent().getNestingLevel() $\geq$
       $n_2$.getParent().getNestingLevel() then
     $n_{merged}$.setParent($n_1$.getParent())
     $E_1$.add($n_{merged}$)
   else if ($n_1$.getParent().getNestingLevel() <
       $n_2$.getParent().getNestingLevel()) then
     $n_{merged}$.setParent($n_2$.getParent())
     $E_2$.add($n_{merged}$)
   end if
   $E_1$.remove($n_1$)
   $E_2$.remove($n_2$)
 end if
```

Listing 6.13: Extension of the function performNodeTemplateMerge

If Node Template n_1's Nesting Level is equal or greater than n_2's, n_1's parent Group Template g_1 will hold the merged Node Template n_{merged} hereafter. This is consistent with the *value preference* requirement discussed in 4.2 that in cases where two options are valid the left-hand side is chosen. If the Nesting Level of Node Template n_2's parent GroupTemplateHierarchy element is greater and, thus, that of n_2, the set E_2 respectively Group Template g_2 is the new parent of Node Template n_{merged}. In both cases n_1 and n_2 are removed from the sets containing them.

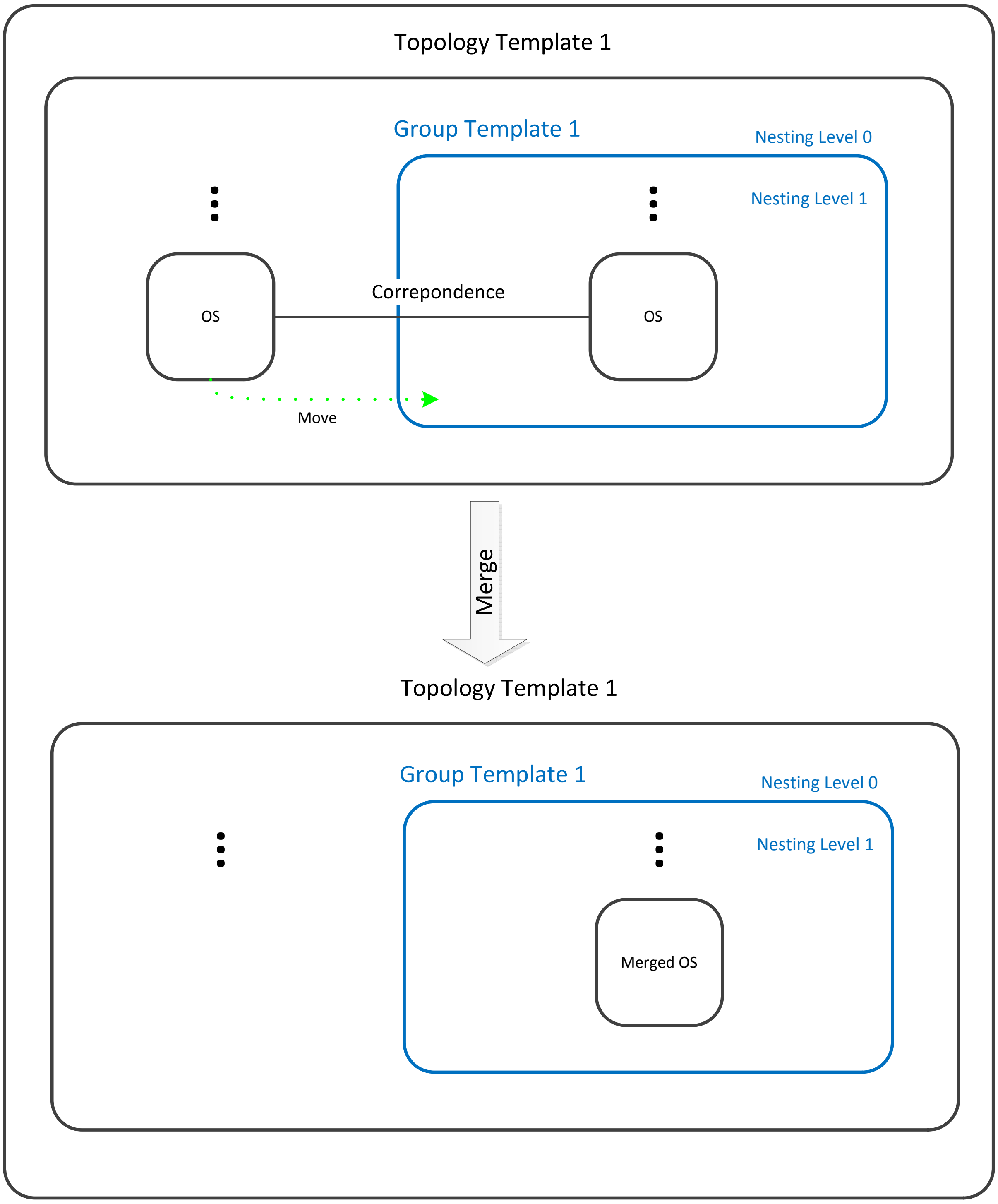


Fig. 6.5: Example of "pulling" a Node Template into a Group Template

Another extension that has to be mentioned is that for the function *reconnectEdges* in Listing 6.2 the Relationship Templates from the Topology Templates are no longer used; but just as in the case of matching Relationship Templates in Section 5.3.5, the previously build Relationship Template set containing all, even nested, Relationship Templates is used.

6.3.2 Merging of Relationship Templates

The basic algorithm for merging of Relationship Templates in Listing 6.10 must also be expanded to handle Relationship Templates located in different Group Templates. Therefore, the lines 29-31 have to be replaced by the same concept proposed in the previous section. The same four cases have to be considered. Let r_1 and r_2 be two Relationship Templates, then the four cases are the following ones: (1) r_1 and r_2 are on Nesting Level 0, i.e. nothing special has to be considered. (2) r_1 is on Nesting Level 0 and r_2 is on a Nesting Level ≥ 1. (3) r_2 is on Nesting Level 0 and r_1 is on a Nesting Level ≥ 1. (4) both Node Templates are on a Nesting Level greater ≥ 1.

All cases are handled in the same way as the Node Templates in the previous section by retrieving the parent Group Template from the parent GroupTemplateHierarchy element when necessary. In case (4), however, it is not decided in which Group Template a merged Relationship Template is transferred according to the Nesting Level. This is due to the fact that Relationship Templates point over the borders of Group Templates and, thus, they stay in the Group Template they were already in at the beginning. A new function called *relocateEdges*, depicted in Fig. 6.6, relocates Relationship Templates, whose source and target Node Templates have been relocated into the same Group Template after merging. The respective Relationship Template is then also moved to that particular Group Template. *relocateEdges* is invoked as subroutine of the function *reconnectEdges* in Listing 6.2. Every Relationship Template that is evaluated if its source and target Node Templates have changed is also evaluated if both Node Templates reside in the same Group Template now.

The function *relocateEdges* requires five input parameters: a Relationship Template r, two Node Templates denoted by n_1 and n_2, as well as two Topology Templates denoted by TT_1 and TT_2. The output is void. The lines 8-10 test if both Node Templates are located in a Group Template, indicated by non-null parent GroupTemplateHierarchy elements, and if they are identical. If so, the content elements of Node Template n_1's parent Group Template are retrieved and stored in a set denoted by E_1 (line 11). If r is not yet element of E_1, it can be added to it (line 12). The next steps, nested inside the latter evaluation, from line 15-22 check from which set of elements the Relationship Template r must be removed. If it has a parent GroupTemplateHierarchy element, i.e. it is nested inside a Group Template, it must be deleted there. If $r \in TT_1$, i.e. it is on Nesting Level 0, it is removed from the corresponding Topology Template, otherwise it can be inferred that $r \in TT_2$ and r is removed accordingly.

```
Algorithm for relocating Relationship Templates
Input: Relationship Template r
Input: Node Template $n_1$
Input: Node Template $n_2$
Input: Topology Template $TT_1$
Input: Topology Template $TT_2$
Output: void
relocateEdges($r_{merged}$,$TT_1$,$TT_2$)
  if $n_1$.getParent() $\neq$ null $\wedge$ $n_2$.getParent() $\neq$ null
    $\wedge$ $n_1$.getParent().getGroupTemplate() ==
    $n_2$.getParent().getGroupTemplate() then
      Set $E_1$ = $n_1$.getParent().getGroupTemplate().getElements()
      if r $\notin$ $E_1$ then
        $E_1$.add(r)

        if $r_{merged}$.getParent() $\neq$ null then
          $E_r$ = r.getParent().getGroupTemplate().getElements()
          $E_r$.remove(r)
        else if r $\in$ $TT_1$ then
          $TT_1$.remove(r)
        else
          $TT_2$.remove(r)
        end if
      end if
  end if
end
```

Fig. 6.6: Function relocateEdges

6.3.3 Merging of Group Templates

This last section discusses the merging of Group Templates on their own by utilizing the found Group Template Correspondences. The overall merging approach is very similar to the merging of Node and Relationship Templates. It is implemented in the function *performGroupTemplateMerge* depicted in Listing 6.14 and Listing 6.15. It will be pointed out which parts are identical with their Node and Relationship Template counterpart as they will not be discussed again in detail. Furthermore, the merging of the Group Template's properties and content will be analyzed.

Merging Algorithm for Group Templates

Listing 6.14 contains the first part of the function *performGroupTemplateMerge*, which is divided into two parts due to its size. The function requires three input parameters: a set of Group Template Correspondences, denoted by GTC, and two Topology Templates denoted by TT_1 and TT_2. The output of the function is a Topology Template containing the merged Group Templates.

The difference of the first part to its Template and Relationship Template counterparts is the fact that the correspondence is now an instance of Group Template Correspondence and not Node Template Correspondence respectively Relationship Template Correspondence. This is evident as the invocation of a subroutine called *hasCorrectNumberOfCorrespondences* and the handling of innate and collected correspondences are identical. Also, a Group Template merger is created in line 18 which handles the specifics of merging two Group Templates such as unifying their Element content.

Merging algorithm for Group Templates part 1

```
 1  Input: Set of Group Template Correspondences GTC
 2  Input: Topology Template $TT_1$
 3  Input: Topology Template $TT_2$
 4  Output: Topology Template $TT_{merged}$
 5  performGroupTemplateMerge(GTC,$TT_1$,$TT_2$)
 6    **for each** Correspondence c $\in$ GTC **do**
 7      **if** c.getFrom() == c.getTo() **then**
 8        **continue**
 9      **end if**
10
11      Group Template $g_1$ = c.getFrom()
12      Group Template $g_2$ = c.getTo()
13
14      **if** $\neg$hasCorrectNumberOfCorrespondences(c,$g_1$,$g_2$) **then**
15        **continue**
16      **end if**
17
18      Merger merger = createGroupTemplateMerger()
19      Group Template $g_{merged}$ = merger.merge($g_1$,$g_2$)
20      $g_{merged}$.setNumberOfMerges($g_1$.getNumberOfMerges + 1)
21      $g_{merged}$.addToCollectedCorrespondences(
22        $g_1$.getCollectedCorrespondences)
23      $g_{merged}$.addToCollectedCorrespondences($g_1$.getCorrespondences)
24      $g_{merged}$.addToCollectedCorrespondences($g_2$.getCorrespondences)
25        ...
```

Listing 6.14: Function performGroupTemplateMerge part 1

In Listing 6.15 the second part of the function *performGroupTemplateMerge* is depicted. This part handles the insertion and deletion of the Group Templates to be merged, denoted by g_1 and g_2, and the merged Group Template g_{merged}. This part is different to its Node and Relationship Template counterparts and is, therefore, analyzed in detail. When merging Group Templates, two different cases have to be considered: Either the Group Templates g_1 and g_2 are both on the same Nesting Level ≥ 1 or they are both on Nesting Level 0. Another case cannot exist as Group Templates Correspondences are only established between Group

Templates on the same Nesting Level. The first case is presented by the lines 26-37 in Listing 6.15. If both Group Templates are on the same Nesting Level ≥ 1, their parent Group Template's content is retrieved and stored in the sets E_1 and E_2. The merged Group Template g_{merged} receives the parent GroupTemplateHierarchy element of g_1 and is added to E_1 while g_1 and g_2 are removed from their respective parent sets. Subsequently, the set of Elements E_g of g_{merged} is retrieved and every element $e \in E_g$ gets g_{merged} assigned as its new managing Group Template.

```
Merging algorithm for Group Templates part 2
25          ...
26        **if** $g_1$.getParent() $\neq$ null $\wedge$ $g_2$.getParent() $\neq$ null **then**
27          Set $E_1$ = $g_1$.getParent().getGroupTemplate().getElements()
28          Set $E_2$ = $g_2$.getParent().getGroupTemplate().getElements()
29          $g_{merged}$.setParent($g_1$.getParent())
30          $E_1$.remove($g_1$))
31          $E_1$.add($g_{merged}$)
32          $E_2$.remove($g_2$))
33
34          Set $E_g$ = $g_{merged}$.getElements()
35          **for each** Element e $\in$ $E_g$ **do**
36            e.getParent().setGroupTemplate($g_{merged}$)
37          **end for each**
38        **else**
39          $TT_1$.remove($g_1$)
40          $TT_2$.remove($g_2$)
45          $TT_1$.add($g_{merged}$)
46
47          Set $E_g$ = $g_{merged}$.getElements()
48          **for each** Element e $\in$ $E_g$ **do**
49            e.getParent().setGroupTemplate($g_{merged}$)
50          **end for each**
51        **end if**
52        GTC = reconnectCorrespondences(GTC,$g_1$,$g_2$,$g_{merged}$)
53      **end for each**
54      **return** $TT_1$
55    **end**
```

Listing 6.15: Function performGroupTemplateMerge part 2

The second case is represented by the pseudo code lines 38-51. The Group Templates are not nested and can be added and removed from their Topology Templates straightforwardly. In line 52 a subroutine is invoked that reconnects the Group Template Correspondences from deleted Group Templates to the merged ones in case of more than one incoming or outgoing

Correspondences. This is done analogously to the Node and Relationship Template counterparts described in Listing 6.3 and Listing 6.11 and will not be discussed here again.

Merging Algorithm for Group Template Properties and Content

In this paragraph an algorithm for merging the properties of Group Templates and unifying their content will be proposed briefly. It is implemented in the function *merge* depicted in Listing 6.16. The function requires two Group Templates denoted by g_1 and g_2 as input parameters. The return value is a merged Group Template.

The first step of the algorithm is to create a new Group Template denoted by g_{merged} (line 5). Subsequently, all Elements from g_1 and g_2 are added to the Elements of g_{merged} (line 6 and 7). The *id* and *name* are taken from Group Template g_1 (*value preference*). The Policies are merged by a subroutine *mergePolicies* which works identical to its Node Template counterpart in Section 6.1.2. The minInstances and maxInstances are unified by the framework using a provided implementation that simply adds the number of instances in both cases. However, the functions can be overridden by user-provided plugins.

```
Algorithm to merge properties and content of two Group Templates
 1  Input: Group Template $g_1$
 2  Input: Group Template $g_2$
 3  Output: Group Template $g_{merged}$
 4  merge($g_1$,$g_2$)
 5     Group Template $g_{merged}$ = new Group Template()
 6     $g_{merged}$.getElements().addAll($g_1$.getElements())
 7     $g_{merged}$.getElements().addAll($g_2$.getElements())
 8     $g_{merged}$.setId($g_1$.getId())
 9     $g_{merged}$.setName($g_1$.getName())
10     $g_{merged}$.setPolicies(mergePolicies($g_1$,$g_2$))
11     $g_{merged}$.setMaxInstances(mergeMaxInstances($g_1$,$g_2$))
12     $g_{merged}$.setMinInstances(mergeMinInstances($g_1$,$g_2$))
13
14     return $g_{merged}$
15  end
```

Listing 6.16: Function merge for Group Templates

7 Architecture & Design of an Extendable Framework

The following chapter aims to design the architecture of a framework that incorporates the matching and merging concepts and algorithms proposed in Chapter 5 and Chapter 6 and is extendable by type- and domain-specific plugins. This chapter is based on the fundamentals discussed in Section 2.4 utilizing them to design an extendable framework. Section 7.1 gives and overview over the proposed high-level architecture. Section 7.2 then refines the architecture into a more detailed design. Finally, Section 7.3 explains the extension concept via type-specific plugins in detail. Both the high-level architecture and the more detailed design are depicted as UML class diagrams. However, with regard to the complexity of the framework not every diagram is shown in detail.

7.1 High-level Architecture

Fig. 7.1 and Fig. 7.2 both show the high-level architecture of the TOSCAMerge framework. The presentation of the architecture is divided into two parts due to the number of classes involved. Fig. 7.1 has its focus on the matching functionality of the framework, whereas Fig. 7.2 concentrates on the merging part. However, both illustrations depict the parts that serve as access point for an external invocation of the framework functionality: The interface *TOSCAMergeService* and one of its possible implementation class *TOSCAMergeServiceStatelessImpl*.

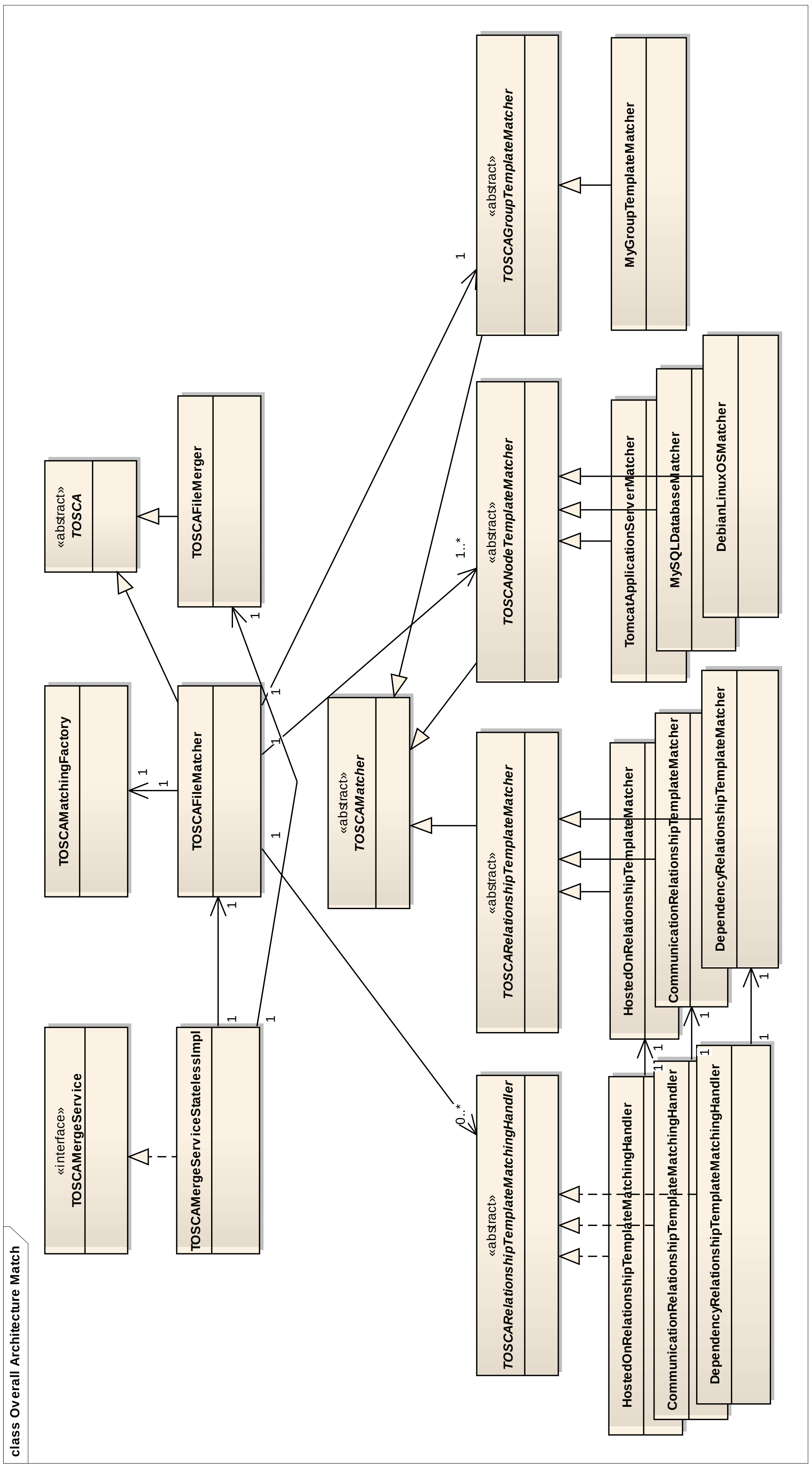

class Overall Architecture Match
«interface»
TOSCAMergeService
TOSCAMergeServiceStatelessImpl
TOSCAMatchingFactory
TOSCAFileMatcher
«abstract»
TOSCA
TOSCAFileMerger
«abstract»
TOSCAMatcher
«abstract»
TOSCARelationshipTemplateMatchingHandler
«abstract»
TOSCARelationshipTemplateMatcher
«abstract»
TOSCANodeTemplateMatcher
«abstract»
TOSCAGroupTemplateMatcher
HostedOnRelationshipTemplateMatchingHandler
CommunicationRelationshipTemplateMatchingHandler
DependencyRelationshipTemplateMatchingHandler
HostedOnRelationshipTemplateMatcher
CommunicationRelationshipTemplateMatcher
DependencyRelationshipTemplateMatcher
TomcatApplicationServerMatcher
MySQLDatabaseMatcher
DebianLinuxOSMatcher
MyGroupTemplateMatcher
1
1
1
1
1
1
1
1
1
1..*
0..*
1
1
1
1
1
1
1

Fig. 7.1: Overall architecture with matching part

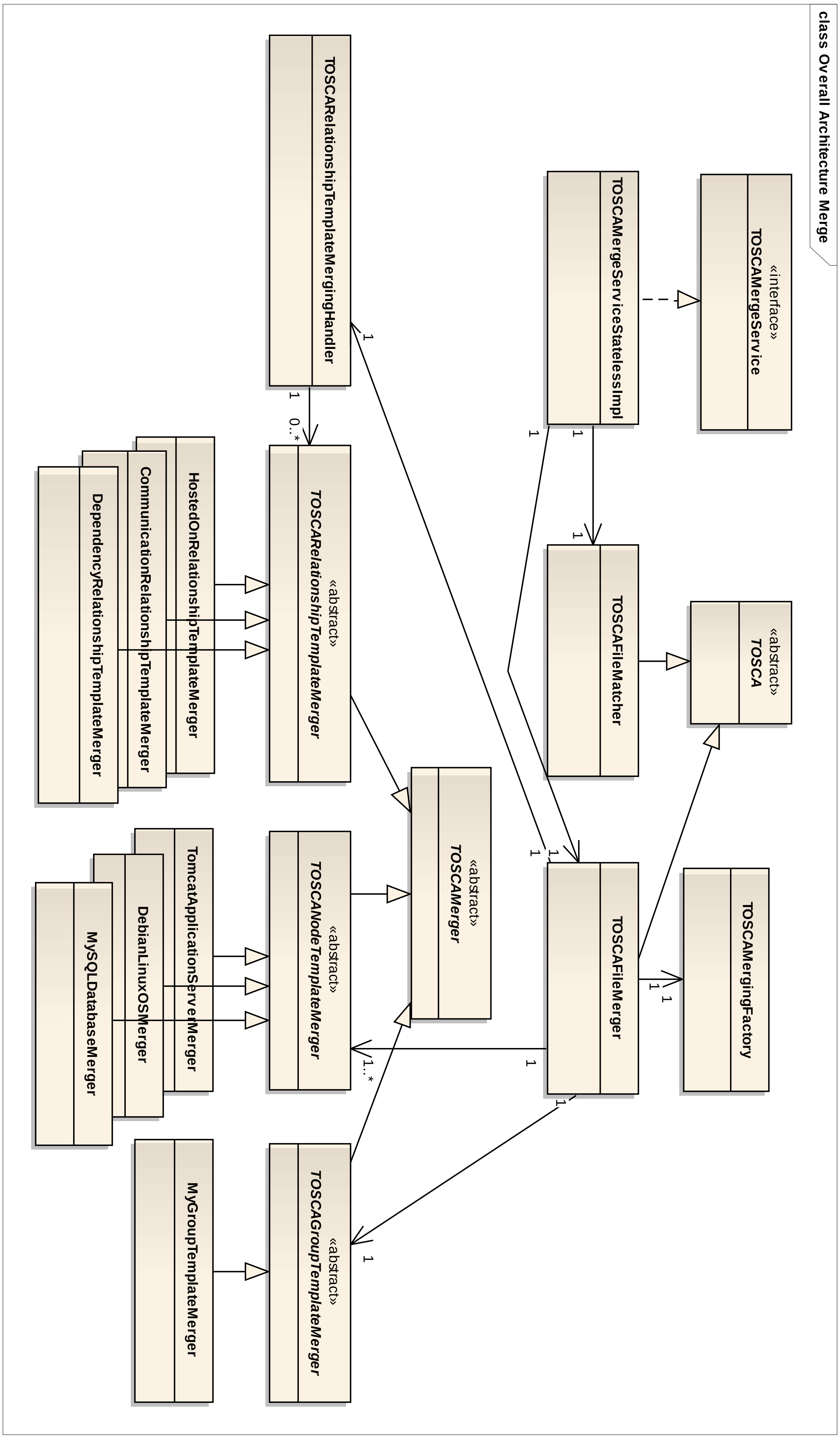


Fig. 7.2: Overall architecture with merging part

7.2 Refined Design

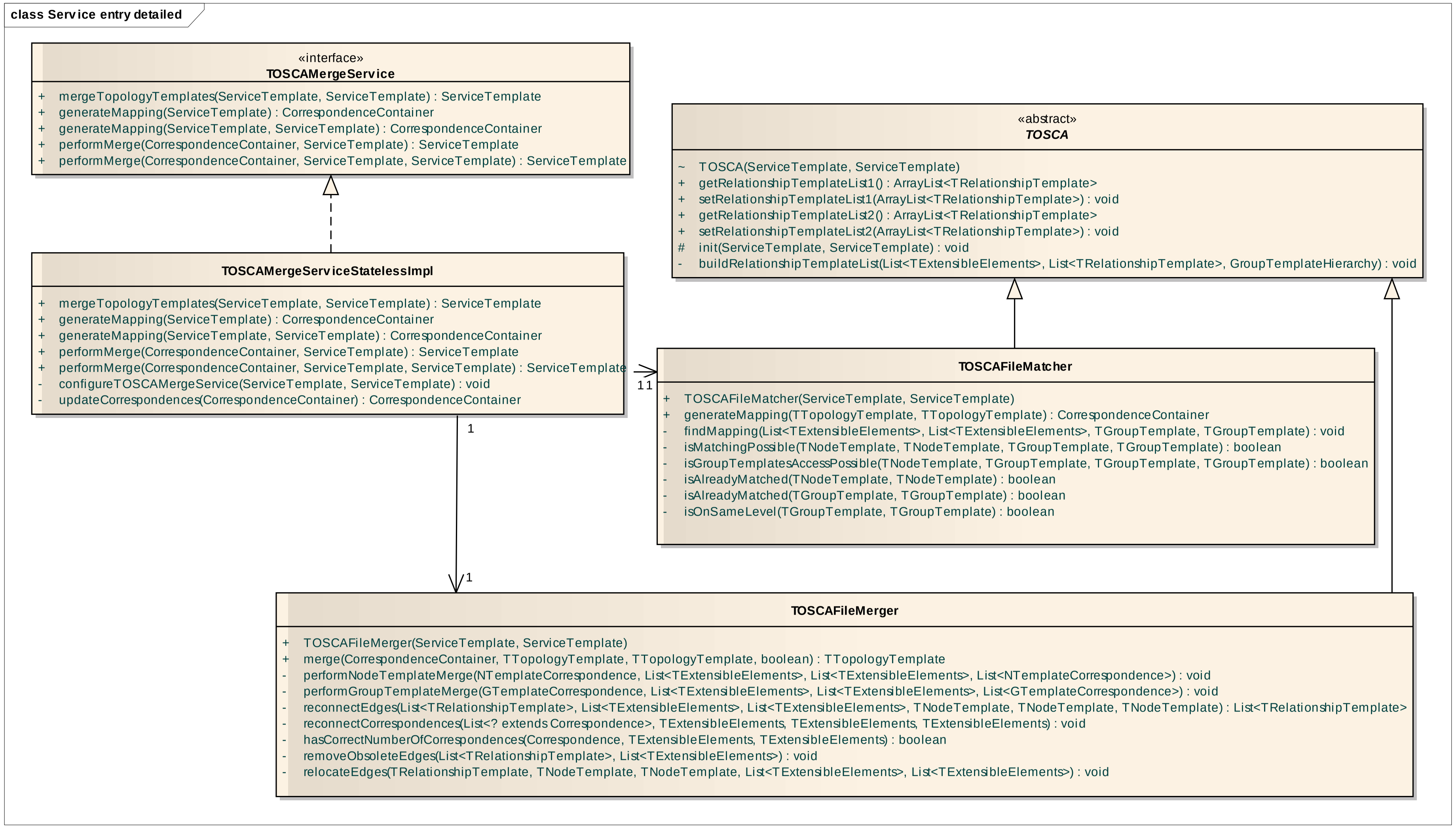

class Service entry detailed
«interface»
TOSCAMergeService
+ mergeTopologyTemplates(ServiceTemplate, ServiceTemplate) : ServiceTemplate
+ generateMapping(ServiceTemplate) : CorrespondenceContainer
+ generateMapping(ServiceTemplate, ServiceTemplate) : CorrespondenceContainer
+ performMerge(CorrespondenceContainer, ServiceTemplate) : ServiceTemplate
+ performMerge(CorrespondenceContainer, ServiceTemplate, ServiceTemplate) : ServiceTemplate
TOSCAMergeServiceStatelessImpl
+ mergeTopologyTemplates(ServiceTemplate, ServiceTemplate) : ServiceTemplate
+ generateMapping(ServiceTemplate) : CorrespondenceContainer
+ generateMapping(ServiceTemplate, ServiceTemplate) : CorrespondenceContainer
+ performMerge(CorrespondenceContainer, ServiceTemplate) : ServiceTemplate
+ performMerge(CorrespondenceContainer, ServiceTemplate, ServiceTemplate) : ServiceTemplate
- configureTOSCAMergeService(ServiceTemplate, ServiceTemplate) : void
- updateCorrespondences(CorrespondenceContainer) : CorrespondenceContainer
«abstract»
TOSCA
~ TOSCA(ServiceTemplate, ServiceTemplate)
+ getRelationshipTemplateList1() : ArrayList<TRelationshipTemplate>
+ setRelationshipTemplateList1(ArrayList<TRelationshipTemplate>) : void
+ getRelationshipTemplateList2() : ArrayList<TRelationshipTemplate>
+ setRelationshipTemplateList2(ArrayList<TRelationshipTemplate>) : void
# init(ServiceTemplate, ServiceTemplate) : void
- buildRelationshipTemplateList(List<TExtensibleElements>, List<TRelationshipTemplate>, GroupTemplateHierarchy) : void
TOSCAFileMatcher
+ TOSCAFileMatcher(ServiceTemplate, ServiceTemplate)
+ generateMapping(TTopologyTemplate, TTopologyTemplate) : CorrespondenceContainer
- findMapping(List<TExtensibleElements>, List<TExtensibleElements>, TGroupTemplate, TGroupTemplate) : void
- isMatchingPossible(TNodeTemplate, TNodeTemplate, TGroupTemplate, TGroupTemplate) : boolean
- isGroupTemplatesAccessPossible(TNodeTemplate, TGroupTemplate, TGroupTemplate, TGroupTemplate) : boolean
- isAlreadyMatched(TNodeTemplate, TNodeTemplate) : boolean
- isAlreadyMatched(TGroupTemplate, TGroupTemplate) : boolean
- isOnSameLevel(TGroupTemplate, TGroupTemplate) : boolean
1
1
1
1
TOSCAFileMerger
+ TOSCAFileMerger(ServiceTemplate, ServiceTemplate)
+ merge(CorrespondenceContainer, TTopologyTemplate, TTopologyTemplate, boolean) : TTopologyTemplate
- performNodeTemplateMerge(NTemplateCorrespondence, List<TExtensibleElements>, List<TExtensibleElements>, List<NTemplateCorrespondence>) : void
- performGroupTemplateMerge(GTemplateCorrespondence, List<TExtensibleElements>, List<TExtensibleElements>, List<GTemplateCorrespondence>) : void
- reconnectEdges(List<TRelationshipTemplate>, List<TExtensibleElements>, List<TExtensibleElements>, TNodeTemplate, TNodeTemplate, TNodeTemplate) : List<TRelationshipTemplate>
- reconnectCorrespondences(List<? extends Correspondence>, TExtensibleElements, TExtensibleElements, TExtensibleElements) : void
- hasCorrectNumberOfCorrespondences(Correspondence, TExtensibleElements, TExtensibleElements) : boolean
- removeObsoleteEdges(List<TRelationshipTemplate>, List<TExtensibleElements>) : void
- relocateEdges(TRelationshipTemplate, TNodeTemplate, TNodeTemplate, List<TExtensibleElements>, List<TExtensibleElements>) : void

Fig. 7.3: Class diagram of the TOSCAMerge framework service access

In the following sections the function of the selected classes and the relationships between them will be explained.

Access to the Service

Fig. 7.3 shows the interface *TOSCAMergeService* that offers several methods for external invocation that either execute the whole merging process of two Topology Templates in one step or separate them into discrete steps that allow for human inspection of the Correspondences as identified in the requirements section in 4.2 (*Assessability of the intermediary results* requirement). The interface is implemented by the class *TOSCAMergeServiceStatelessImpl* that provides a stateless implementation of the service interface. The implementation class has connections to the classes *TOSCAFileMatcher* and *TOSCAFileMerger* that contain the proposed generic algorithms of the Chapters 5 and 6. The common abstract super class, denoted by *TOSCA,* of both classes holds common functionality such as the buildRelationshipTemplateList[16] from Section 5.3.5.

Matching Functionality

Fig. 7.4 shows a detailed UML class diagram of the frameworks matching classes that hold the matching functionality. Not shown here are the type-specific classes that extend the framework and their creation by the TOSCAMatchingFactory. This is exemplarily discussed in the next section for the class *TOSCANodeTemplateMatcher.*

Once again the class *TOSCAFileMatcher* that implements the generic high-level matching algorithm is depicted. However, the generic actual matching of the properties of two particular Node, Relationship or Group Templates is done inside the abstract classes *TOSCANodeTemplateMatcher*, *TOSCARelationshipTemplateMatcher* and *TOSCAGroupTemplateMatcher.* These classes share the common abstract superclass *TOSCAMatcher* that holds common functionality such as determining derived Policies (see Chapter 5). The abstract class *TOSCARelationshipTemplateMatchingHandler* corresponds to the introduced concept of RelationshipTemplateMatchingHandlers that have a function *handleRelationshipTemplates* capable of finding Relationship Template Correspondences (see Section 5.2.1). All matchers declare the functions that were introduced before in Chapter 5.

[16] Actually the function was called *buildRelationshipTemplateSet*, but the sets are implemented as lists.

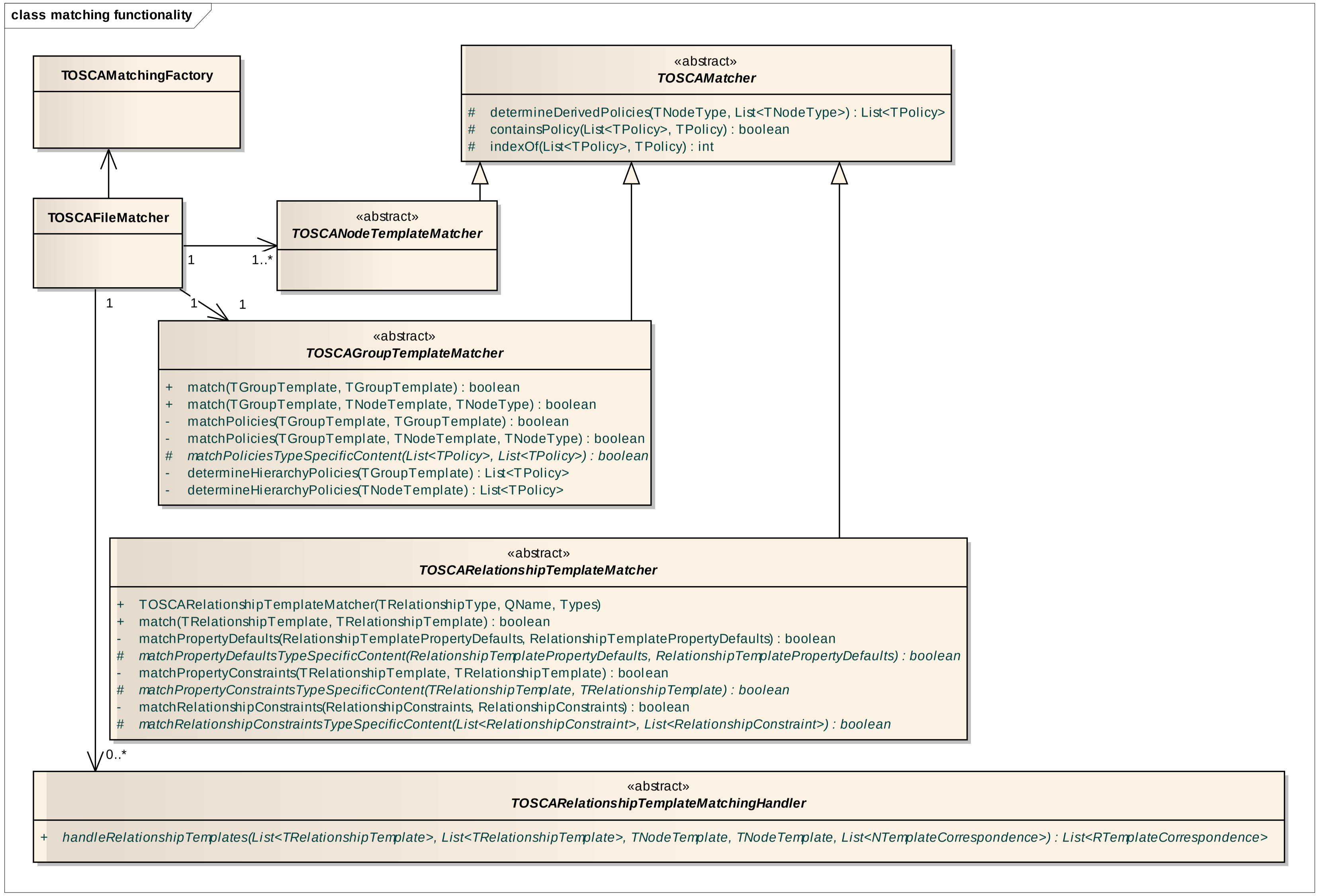

class matching functionality
TOSCAMatchingFactory
TOSCAFileMatcher
«abstract»
TOSCANodeTemplateMatcher
1
1..*
1
1
1
«abstract»
TOSCAMatcher
# determineDerivedPolicies(TNodeType, List<TNodeType>) : List<TPolicy>
# containsPolicy(List<TPolicy>, TPolicy) : boolean
# indexOf(List<TPolicy>, TPolicy) : int
«abstract»
TOSCAGroupTemplateMatcher
+ match(TGroupTemplate, TGroupTemplate) : boolean
+ match(TGroupTemplate, TNodeTemplate, TNodeType) : boolean
- matchPolicies(TGroupTemplate, TGroupTemplate) : boolean
- matchPolicies(TGroupTemplate, TNodeTemplate, TNodeType) : boolean
# matchPoliciesTypeSpecificContent(List<TPolicy>, List<TPolicy>) : boolean
- determineHierarchyPolicies(TGroupTemplate) : List<TPolicy>
- determineHierarchyPolicies(TNodeTemplate) : List<TPolicy>
«abstract»
TOSCARelationshipTemplateMatcher
+ TOSCARelationshipTemplateMatcher(TRelationshipType, QName, Types)
+ match(TRelationshipTemplate, TRelationshipTemplate) : boolean
- matchPropertyDefaults(RelationshipTemplatePropertyDefaults, RelationshipTemplatePropertyDefaults) : boolean
# matchPropertyDefaultsTypeSpecificContent(RelationshipTemplatePropertyDefaults, RelationshipTemplatePropertyDefaults) : boolean
- matchPropertyConstraints(TRelationshipTemplate, TRelationshipTemplate) : boolean
# matchPropertyConstraintsTypeSpecificContent(TRelationshipTemplate, TRelationshipTemplate) : boolean
- matchRelationshipConstraints(RelationshipConstraints, RelationshipConstraints) : boolean
# matchRelationshipConstraintsTypeSpecificContent(List<RelationshipConstraint>, List<RelationshipConstraint>) : boolean
0..*
«abstract»
TOSCARelationshipTemplateMatchingHandler
+ handleRelationshipTemplates(List<TRelationshipTemplate>, List<TRelationshipTemplate>, TNodeTemplate, TNodeTemplate, List<NTemplateCorrespondence>) : List<RTemplateCorrespondence>

Fig. 7.4: Detailed design of the matching part

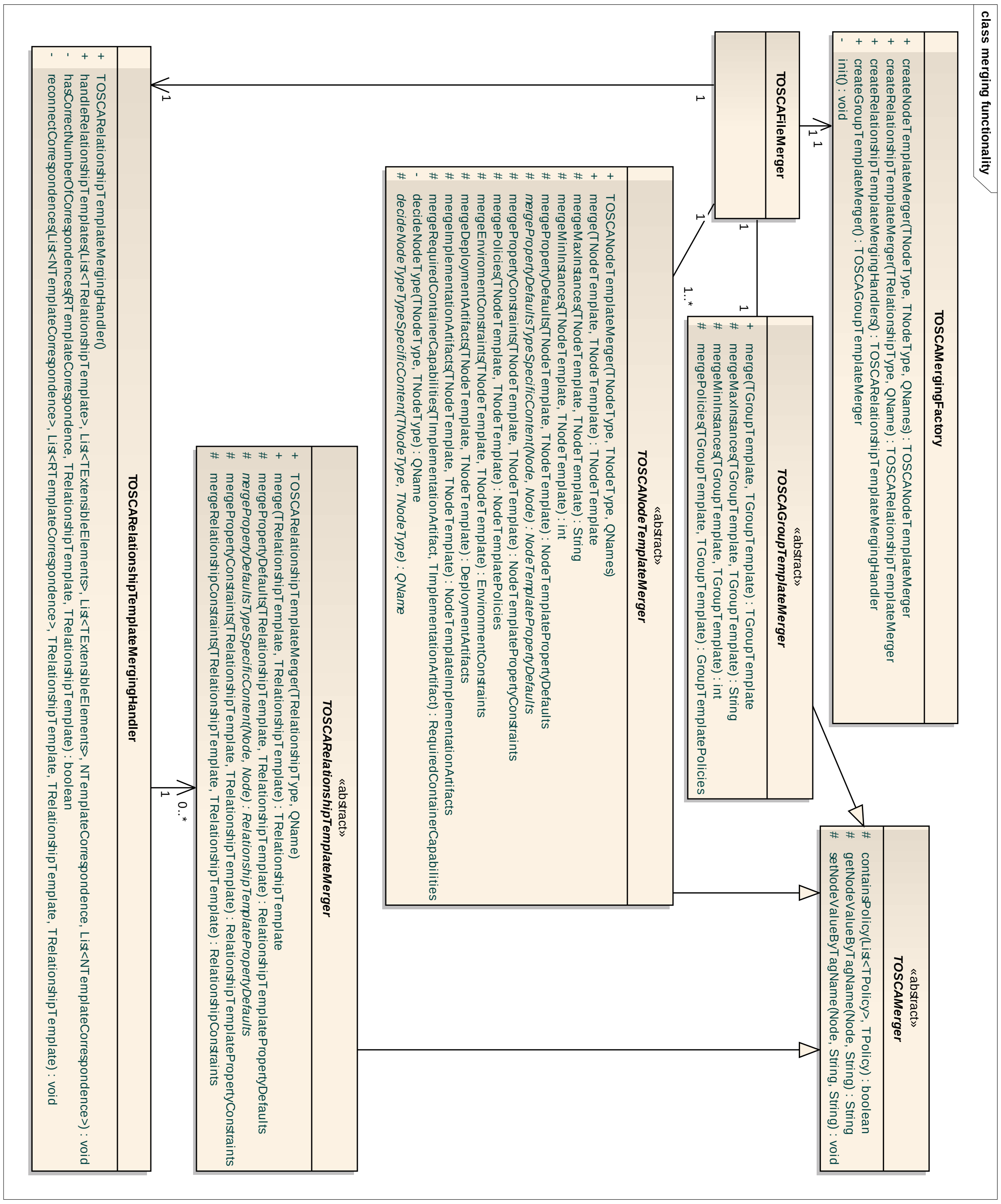

class merging functionality
TOSCAMergingFactory
+ createNodeTemplateMerger(TNodeType, TNodeType, QNames) : TOSCANodeTemplateMerger
+ createRelationshipTemplateMerger(TRelationshipType, QName) : TOSCARelationshipTemplateMerger
+ createRelationshipTemplateMergingHandler() : TOSCARelationshipTemplateMergingHandler
+ createGroupTemplateMerger() : TOSCAGroupTemplateMerger
- init() : void
TOSCAFileMerger
«abstract» TOSCAGroupTemplateMerger
+ merge(TGroupTemplate, TGroupTemplate) : TGroupTemplate
# mergeMaxInstances(TGroupTemplate, TGroupTemplate) : String
# mergeMinInstances(TGroupTemplate, TGroupTemplate) : int
# mergePolicies(TGroupTemplate, TGroupTemplate) : GroupTemplatePolicies
«abstract» TOSCANodeTemplateMerger
+ TOSCANodeTemplateMerger(TNodeType, TNodeType, QNames)
+ merge(TNodeTemplate, TNodeTemplate) : TNodeTemplate
# mergeMaxInstances(TNodeTemplate, TNodeTemplate) : String
# mergeMinInstances(TNodeTemplate, TNodeTemplate) : int
# mergePropertyDefaults(TNodeTemplate, TNodeTemplate) : NodeTemplatePropertyDefaults
# mergePropertyDefaultsTypeSpecificContent(Node, Node) : NodeTemplatePropertyDefaults
# mergePropertyConstraints(TNodeTemplate, TNodeTemplate) : NodeTemplatePropertyConstraints
# mergePolicies(TNodeTemplate, TNodeTemplate) : NodeTemplatePolicies
# mergeEnvironmentConstraints(TNodeTemplate, TNodeTemplate) : EnvironmentConstraints
# mergeDeploymentArtifacts(TNodeTemplate, TNodeTemplate) : DeploymentArtifacts
# mergeImplementationArtifacts(TNodeTemplate, TNodeTemplate) : NodeTemplateImplementationArtifacts
# mergeRequiredContainerCapabilities(TImplementationArtifact, TImplementationArtifact) : RequiredContainerCapabilities
- decideNodeType(TNodeType, TNodeType) : QName
# decideNodeTypeSpecificContent(TNodeType, TNodeType) : QName
«abstract» TOSCAMerger
# containsPolicy(List<TPolicy>, TPolicy) : boolean
# getNodeValueByTagName(Node, String) : String
# setNodeValueByTagName(Node, String, String) : void
«abstract» TOSCARelationshipTemplateMerger
+ TOSCARelationshipTemplateMerger(TRelationshipType, QName)
+ merge(TRelationshipTemplate, TRelationshipTemplate) : TRelationshipTemplate
# mergePropertyDefaults(TRelationshipTemplate, TRelationshipTemplate) : RelationshipTemplatePropertyDefaults
# mergePropertyDefaultsTypeSpecificContent(Node, Node) : RelationshipTemplatePropertyDefaults
# mergePropertyConstraints(TRelationshipTemplate, TRelationshipTemplate) : RelationshipTemplatePropertyConstraints
# mergeRelationshipConstraints(TRelationshipTemplate, TRelationshipTemplate) : RelationshipConstraints
TOSCARelationshipTemplateMergingHandler
+ TOSCARelationshipTemplateMergingHandler()
+ handleRelationshipTemplates(List<TRelationshipTemplate>, List<TExtensibleElements>, List<TExtensibleElements>, NTemplateCorrespondence, List<NTemplateCorrespondence>) : void
- hasCorrectNumberOfCorrespondences(RTemplateCorrespondence, TRelationshipTemplate, TRelationshipTemplate) : boolean
- reconnectCorrespondences(List<NTemplateCorrespondence>, List<RTemplateCorrespondence>, TRelationshipTemplate, TRelationshipTemplate) : void
1
1
1
1
1
1
1..*
1
1
0..*

Fig. 7.5: Detailed design of the merging part

Merging Functionality

Fig. 7.5 shows the classes that hold the merging functionality. They have a similar distribution of tasks as the matching counterpart. The starting point is the class *TOSCAFileMerger* that holds all the high-level merging functions such as *hasCorrectNumberOfCorrespondences* (see Listing 6.4, Listing 6.5 and Fig. 7.3). It has associations with the three abstract classes *TOSCANodeTemplateMerger*, *TOSCAGroupTemplateMerger,* and *TOSCARelationshipTemplateMergingHandler.* The latter one contains the high-level merging algorithm for Relationship Templates and uses subclasses of the abstract class *TOSCARelationshipTemplateMerger* to merge the properties Relationship Templates of a particular Relationship Type. Note that the *TOSCARelationshipTemplateMergingHandler* is not abstract like the matching counterpart since the high-level merging of Relationship Templates can be done generically and does not need type-specific extensions to the TOSCAMerge framework. The subclasses of *TOSCARelationshipTemplateMerger* for the merging of Relationship Template Properties are not shown here due to space limitations but the concept is discussed in the next section. *TOSCANodeTemplateMerger* and *TOSCAGroupTemplateMerger* are responsible for merging the properties of Node Templates and Group Templates as proposed in Section 6.1.2 and 6.3.3.

Furthermore, all the abstract classes for the merging of properties share a common abstract super class denoted by *TOSCAMerger* holding functionality commonly used by the subclasses. The last class to mention is the TOSCAMergingFactory that declares factory methods to create concrete instances of Node, Relationship and Group Template mergers. This concept is picked up again in Section 7.3.

Design of Correspondences

Fig. 7.6 shows a class diagram of the proposed Correspondence concept of this thesis. The abstract class *Correspondence* forms the superclass of all other types of Correspondences. The superclass defines that every type of Correspondence must have getter- and setter-methods for the elements it connects. This enables the use of the abstract class in a generic way utilizing polymorphism [23]. For example the functions *hasCorrectNumberOfCorrespondences* introduced in Listing 6.4 and Listing 6.5 for Node Templates can be implemented generically to fit for all types of Correspondences. The derived classes override the methods to fit to their corresponding member variables. Furthermore, the class *RTemplateCorrespondences* representing the concept of Relationship Template Correspondences is always attached to, and cannot exist without, a Node Template Correspondence represented by the class *NTemplateCorrespondence*. This is indicated by the UML composition relationship.

Design of the Group Template Hierarchy and the BaseClass

Fig. 7.7 depicts the interaction of different TOSCA elements with the proposed concept of a Group Template Hierarchy and the concept of a superclass for all TOSCA elements. Exemplarily, the classes representing Node and Group Templates are shown to illustrate these concepts. It is visible that the class *TNodeTemplate* representing a Node Template and its properties is the subclass of the class *TExtensibleElements* that realizes the concept of extensibility for all TOSCA elements as described in Section 2.3.1. The class *TExtensibleElements* is subclass of the class *BaseClass* that is introduced to be the abstract superclass for all TOSCA elements in the context of TOSCAMerge framework and this master's thesis. It holds the Correspondences outgoing from a particular element, be it a Node, Relationship or Group Template, the collected Correspondences and the number of already conducted merges during the merging process. Moreover, every element has a reference to its corresponding instance of the *GroupTemplateHierarchy* via its *BaseClass.*

An instance of *GroupTemplateHierarchy* can also be the parent of more than one TOSCA element.The *GroupTemplateHierarchy* class incorporates the concept of the data structure proposed in Section 5.3 including the functions working on it.

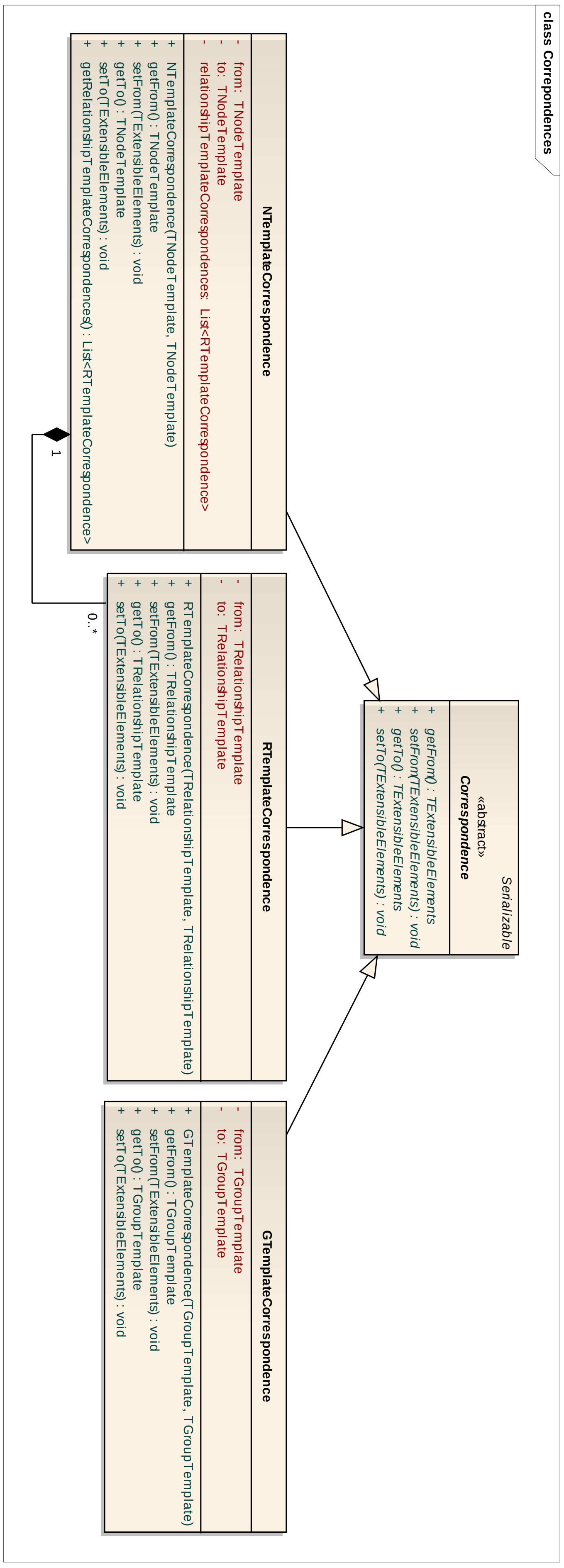
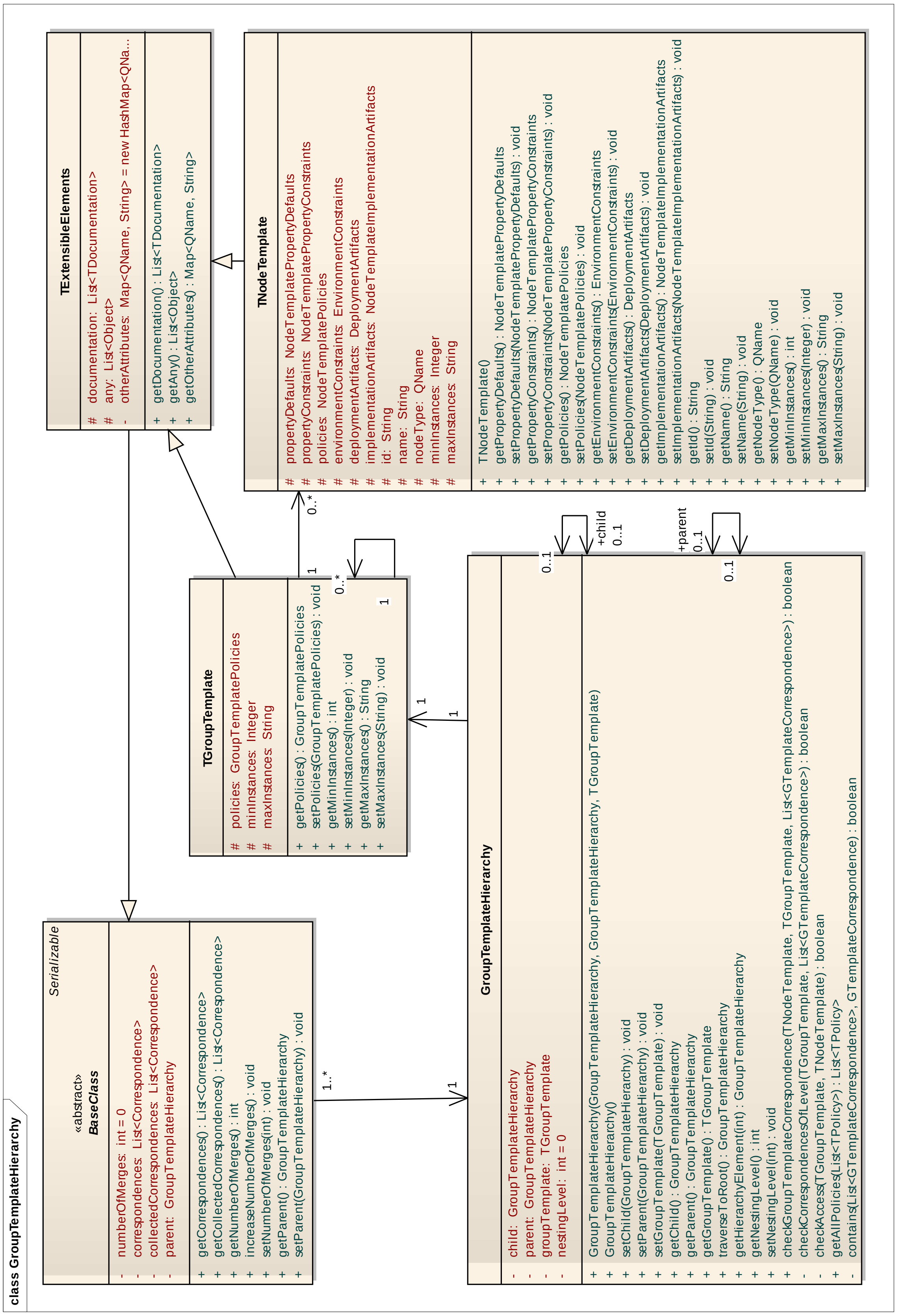

class Correspondences
Serializable
«abstract»
Correspondence
+ getFrom() : TExtensibleElements
+ setFrom(TExtensibleElements) : void
+ getTo() : TExtensibleElements
+ setTo(TExtensibleElements) : void
NTemplateCorrespondence
- from: TNodeTemplate
- to: TNodeTemplate
- relationshipTemplateCorrespondences: List<RTemplateCorrespondence>
+ NTemplateCorrespondence(TNodeTemplate, TNodeTemplate)
+ getFrom() : TNodeTemplate
+ setFrom(TExtensibleElements) : void
+ getTo() : TNodeTemplate
+ setTo(TExtensibleElements) : void
+ getRelationshipTemplateCorrespondences() : List<RTemplateCorrespondence>
1
0..*
RTemplateCorrespondence
- from: TRelationshipTemplate
- to: TRelationshipTemplate
+ RTemplateCorrespondence(TRelationshipTemplate, TRelationshipTemplate)
+ getFrom() : TRelationshipTemplate
+ setFrom(TExtensibleElements) : void
+ getTo() : TRelationshipTemplate
+ setTo(TExtensibleElements) : void
GTemplateCorrespondence
- from: TGroupTemplate
- to: TGroupTemplate
+ GTemplateCorrespondence(TGroupTemplate, TGroupTemplate)
+ getFrom() : TGroupTemplate
+ setFrom(TExtensibleElements) : void
+ getTo() : TGroupTemplate
+ setTo(TExtensibleElements) : void

Fig. 7.6: Class diagram of the Correspondences


class GroupTemplateHierarchy
Serializable
«abstract»
BaseClass
- numberOfMerges: int = 0
- correspondences: List<Correspondence>
- collectedCorrespondences: List<Correspondence>
- parent: GroupTemplateHierarchy
+ getCorrespondences() : List<Correspondence>
+ getCollectedCorrespondences() : List<Correspondence>
+ getNumberOfMerges() : int
+ increaseNumberOfMerges() : void
+ setNumberOfMerges(int) : void
+ getParent() : GroupTemplateHierarchy
+ setParent(GroupTemplateHierarchy) : void
TExtensibleElements
# documentation: List<TDocumentation>
# any: List<Object>
- otherAttributes: Map<QName, String> = new HashMap<QNa...
+ getDocumentation() : List<TDocumentation>
+ getAny() : List<Object>
+ getOtherAttributes() : Map<QName, String>
TNodeTemplate
# propertyDefaults: NodeTemplatePropertyDefaults
# propertyConstraints: NodeTemplatePropertyConstraints
# policies: NodeTemplatePolicies
# environmentConstraints: EnvironmentConstraints
# deploymentArtifacts: DeploymentArtifacts
# implementationArtifacts: NodeTemplateImplementationArtifacts
# id: String
# name: String
# nodeType: QName
# minInstances: Integer
# maxInstances: String
+ TNodeTemplate()
+ getPropertyDefaults() : NodeTemplatePropertyDefaults
+ setPropertyDefaults(NodeTemplatePropertyDefaults) : void
+ getPropertyConstraints() : NodeTemplatePropertyConstraints
+ setPropertyConstraints(NodeTemplatePropertyConstraints) : void
+ getPolicies() : NodeTemplatePolicies
+ setPolicies(NodeTemplatePolicies) : void
+ getEnvironmentConstraints() : EnvironmentConstraints
+ setEnvironmentConstraints(EnvironmentConstraints) : void
+ getDeploymentArtifacts() : DeploymentArtifacts
+ setDeploymentArtifacts(DeploymentArtifacts) : void
+ getImplementationArtifacts() : NodeTemplateImplementationArtifacts
+ setImplementationArtifacts(NodeTemplateImplementationArtifacts) : void
+ getId() : String
+ setId(String) : void
+ getName() : String
+ setName(String) : void
+ getNodeType() : QName
+ setNodeType(QName) : void
+ getMinInstances() : int
+ setMinInstances(Integer) : void
+ getMaxInstances() : String
+ setMaxInstances(String) : void
TGroupTemplate
# policies: GroupTemplatePolicies
# minInstances: Integer
# maxInstances: String
+ getPolicies() : GroupTemplatePolicies
+ setPolicies(GroupTemplatePolicies) : void
+ getMinInstances() : int
+ setMinInstances(Integer) : void
+ getMaxInstances() : String
+ setMaxInstances(String) : void
GroupTemplateHierarchy
- child: GroupTemplateHierarchy
- parent: GroupTemplateHierarchy
- groupTemplate: TGroupTemplate
- nestingLevel: int = 0
+ GroupTemplateHierarchy(GroupTemplateHierarchy, GroupTemplateHierarchy, TGroupTemplate)
+ GroupTemplateHierarchy()
+ setChild(GroupTemplateHierarchy) : void
+ setParent(GroupTemplateHierarchy) : void
+ setGroupTemplate(TGroupTemplate) : void
+ getChild() : GroupTemplateHierarchy
+ getParent() : GroupTemplateHierarchy
+ getGroupTemplate() : TGroupTemplate
+ traverseToRoot() : GroupTemplateHierarchy
+ getHierarchyElement(int) : GroupTemplateHierarchy
+ getNestingLevel() : int
+ setNestingLevel(int) : void
+ checkGroupTemplateCorrespondence(TNodeTemplate, TGroupTemplate, List<GTemplateCorrespondence>) : boolean
- checkCorrespondencesOfLevel(TGroupTemplate, List<GTemplateCorrespondence>) : boolean
- checkAccess(TGroupTemplate, TNodeTemplate) : boolean
- getAllPolicies(List<TPolicy>) : List<TPolicy>
+ contains(List<GTemplateCorrespondence>, GTemplateCorrespondence) : boolean
1..*
1
1
0..*
0..*
1
1
1
0..1
+child
0..1
+parent
0..1
0..1
Fig. 7.7: Class diagram of the Group Template Hierarchy concept

Its methods' names are taken from the functions' names one-to-one plus additional getter- and setter-methods for the member variables. Each instance of *GroupTemplateHierarchy* has a reference to a *TGroupTemplate* instance that can possibly contain instances of *TNodeTemplate* or *TGroupTemplate*.[17]

7.3 Extensibility of the TOSCAMerge Framework

In Section 2.4 the two design patterns Template Method and Factory Method have been introduced and discussed. In the following section it is now explored how these design patterns can be utilized to make the TOSCAMerge framework extensible for matching and merging new Node and Relationship Types. In Fig. 7.8 the extensibility of Node Template matching is depicted exemplarily. The Template Method design pattern (see Fig. 2.2) is used by the abstract class *TOSCANodeTemplateMatcher* and the derived class *TomcatApplicationServerMatcher*. The *template method* is the public *match* method which equates the *match* function introduced in Listing 5.6 in Section 5.1.2. The *match* method is declared *final* to comply with the *open/close-principle* of software architecture. It defines the control flow of the matching between two Node Templates having identical or related Node Types as proposed in the *match* function, i.e. the properties are compared successively by invoking the corresponding subroutines. If e.g. the control flow reaches the matching of Policies, the private *matchPolicies* method of the *TOSCANodeTemplateMatcher* is invoked. It executes the generic steps to calculate the effective set of Policies of both Node Templates as discussed before and then invokes the method *matchPoliciesTypeSpecificContent* passing the set of Policies. This method represents what was called a *hook method* before. Hence, the class *TomcatApplicationServerMatcher* is an implementation class for all the defined hook methods of *TOSCANodeTemplateMatcher*. All the matching decisions that cannot be made generically are handled by the subclasses that correspond to particular Node Types. Thus, the type-specific subclasses represent the *variation points* or *hot spots* of the TOSCAMerge framework.

The second design pattern reviewed above was the Factory Method. This pattern is not used in the TOSCAMerge framework in the classical way as depicted in Fig. 2.3. There, an abstract factory class declared a factory method that was implemented by deriving concrete factory classes for every specific class that has to be created. In the TOSCAMerge framework the Factory Method design pattern is used in a modified version [21], [29].

Instead of providing a concrete factory class for every identical Node Type and for every valid combination of different Node Types, a non-abstract factory is used that is able to create *TOSCANodeTemplateMatcher* implementation classes in a generic way. This is done by initially loading an XML file that specifies the designated qualified names of Node Types and the corresponding variation point implementation classes and instantiating these classes dynamically via Java Reflection API[18] when needed. Thus, for extending the TOSCAMerge framework, an additional entry in the configuration file in conjunction with an appropriate class implementing all hook methods is sufficient.

[17] The class diagram is simplified here for brevity. *TGroupTemplate* has a subclass *TTopologyElementCollection* that actually holds the references to TNodeTemplate and TGroupTemplate but the principle is the same.
[18] See next chapter for details.

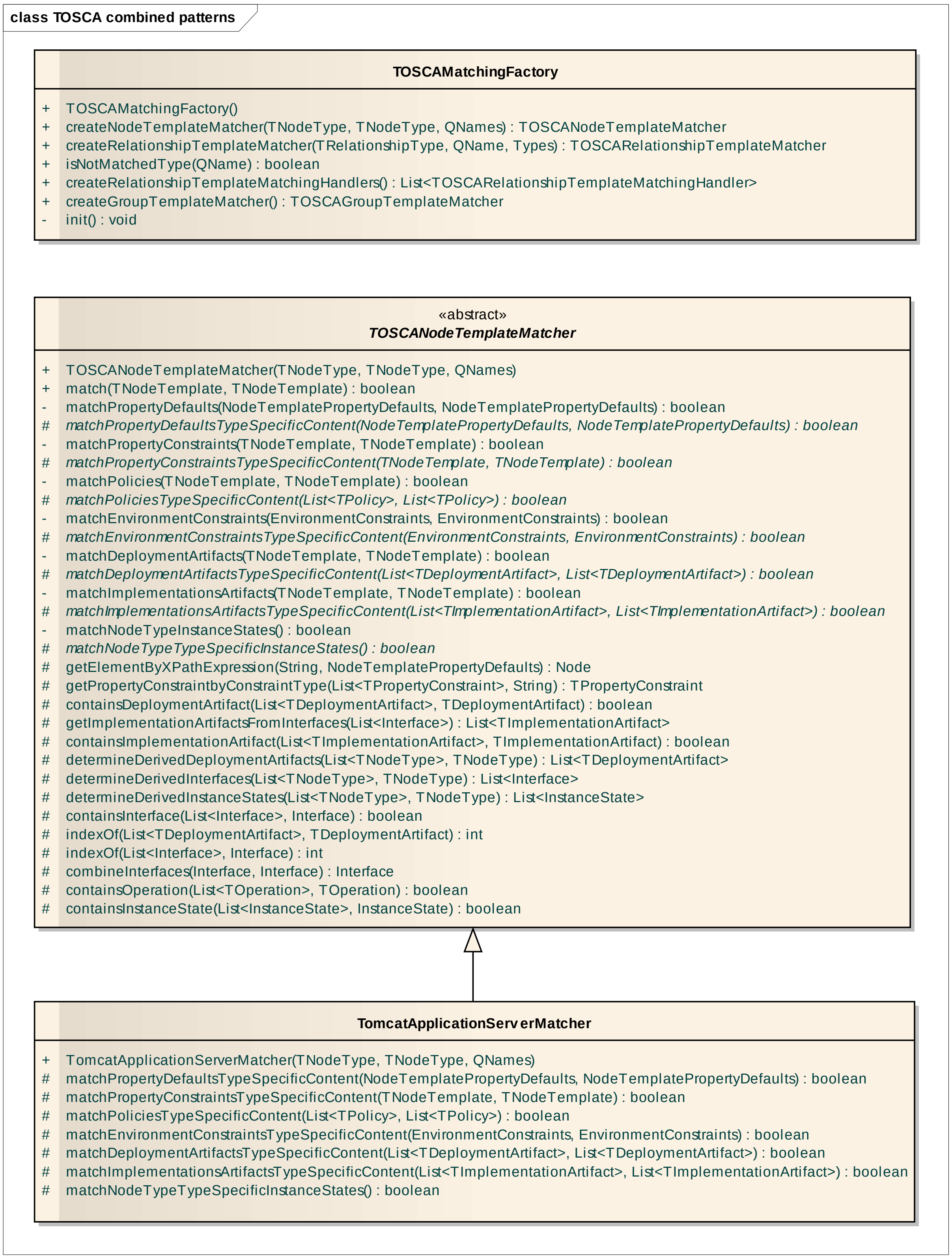

class TOSCA combined patterns
TOSCAMatchingFactory
+ TOSCAMatchingFactory()
+ createNodeTemplateMatcher(TNodeType, TNodeType, QNames) : TOSCANodeTemplateMatcher
+ createRelationshipTemplateMatcher(TRelationshipType, QName, Types) : TOSCARelationshipTemplateMatcher
+ isNotMatchedType(QName) : boolean
+ createRelationshipTemplateMatchingHandlers() : List<TOSCARelationshipTemplateMatchingHandler>
+ createGroupTemplateMatcher() : TOSCAGroupTemplateMatcher
- init() : void
«abstract»
TOSCANodeTemplateMatcher
+ TOSCANodeTemplateMatcher(TNodeType, TNodeType, QNames)
+ match(TNodeTemplate, TNodeTemplate) : boolean
- matchPropertyDefaults(NodeTemplatePropertyDefaults, NodeTemplatePropertyDefaults) : boolean
# matchPropertyDefaultsTypeSpecificContent(NodeTemplatePropertyDefaults, NodeTemplatePropertyDefaults) : boolean
- matchPropertyConstraints(TNodeTemplate, TNodeTemplate) : boolean
# matchPropertyConstraintsTypeSpecificContent(TNodeTemplate, TNodeTemplate) : boolean
- matchPolicies(TNodeTemplate, TNodeTemplate) : boolean
# matchPoliciesTypeSpecificContent(List<TPolicy>, List<TPolicy>) : boolean
- matchEnvironmentConstraints(EnvironmentConstraints, EnvironmentConstraints) : boolean
# matchEnvironmentConstraintsTypeSpecificContent(EnvironmentConstraints, EnvironmentConstraints) : boolean
- matchDeploymentArtifacts(TNodeTemplate, TNodeTemplate) : boolean
# matchDeploymentArtifactsTypeSpecificContent(List<TDeploymentArtifact>, List<TDeploymentArtifact>) : boolean
- matchImplementationsArtifacts(TNodeTemplate, TNodeTemplate) : boolean
# matchImplementationsArtifactsTypeSpecificContent(List<TImplementationArtifact>, List<TImplementationArtifact>) : boolean
- matchNodeTypeInstanceStates() : boolean
# matchNodeTypeTypeSpecificInstanceStates() : boolean
# getElementByXPathExpression(String, NodeTemplatePropertyDefaults) : Node
# getPropertyConstraintbyConstraintType(List<TPropertyConstraint>, String) : TPropertyConstraint
# containsDeploymentArtifact(List<TDeploymentArtifact>, TDeploymentArtifact) : boolean
# getImplementationArtifactsFromInterfaces(List<Interface>) : List<TImplementationArtifact>
# containsImplementationArtifact(List<TImplementationArtifact>, TImplementationArtifact) : boolean
# determineDerivedDeploymentArtifacts(List<TNodeType>, TNodeType) : List<TDeploymentArtifact>
# determineDerivedInterfaces(List<TNodeType>, TNodeType) : List<Interface>
# determineDerivedInstanceStates(List<TNodeType>, TNodeType) : List<InstanceState>
# containsInterface(List<Interface>, Interface) : boolean
# indexOf(List<TDeploymentArtifact>, TDeploymentArtifact) : int
# indexOf(List<Interface>, Interface) : int
# combineInterfaces(Interface, Interface) : Interface
# containsOperation(List<TOperation>, TOperation) : boolean
# containsInstanceState(List<InstanceState>, InstanceState) : boolean
TomcatApplicationServerMatcher
+ TomcatApplicationServerMatcher(TNodeType, TNodeType, QNames)
# matchPropertyDefaultsTypeSpecificContent(NodeTemplatePropertyDefaults, NodeTemplatePropertyDefaults) : boolean
# matchPropertyConstraintsTypeSpecificContent(TNodeTemplate, TNodeTemplate) : boolean
# matchPoliciesTypeSpecificContent(List<TPolicy>, List<TPolicy>) : boolean
# matchEnvironmentConstraintsTypeSpecificContent(EnvironmentConstraints, EnvironmentConstraints) : boolean
# matchDeploymentArtifactsTypeSpecificContent(List<TDeploymentArtifact>, List<TDeploymentArtifact>) : boolean
# matchImplementationsArtifactsTypeSpecificContent(List<TImplementationArtifact>, List<TImplementationArtifact>) : boolean
# matchNodeTypeTypeSpecificInstanceStates() : boolean

Fig. 7.8: Class diagrams of the combined design patterns

Listing 7.1 and Listing 7.2 show the XML schema file that specifies how the framework configurations file for Node Templates respective their Node Types looks like.

```
Schema of a Node-Type-specific configuration part 1
 1  <?xml version="1.0" encoding="UTF-8"?>
 2  <xs:schema
 3    xmlns:xs="http://www.w3.org/2001/XMLSchema">
 4
 5    <xs:element name=”NodeTypeSpecificImplementations"
 6      type="tTypeSpecificNodeTypeConfig">
 7    </xs:element>
 8
 9    <xs:complexType name="tTypeSpecificNodeTypeConfig">
10      <xs:sequence>
11        <xs:element name="nodeTypeMatchers"
12          type="tSpecificNodeTypeMatcher" />
13        <xs:element name="nodeTypeMergers"
14          type="tSpecificNodeTypeMerger" />
15      </xs:sequence>
16    </xs:complexType>
17
18      <xs:complexType name="tSpecificNodeTypeMatcher">
19      <xs:sequence>
20        <xs:element name="specificMatcher" type="tNodeTypeDetail"
21          maxOccurs="unbounded" />
22      </xs:sequence>
23    </xs:complexType>
24        ...
```

Listing 7.1: Schema of Node-Type-specific configuration part 1

The configuration file is modeled the following way: the element *NodeTypeSpecificImplementations* in line 5 of Listing 7.1 forms the root element of the configuration file. The configuration element is divided into two sections *nodeTypeMatchers* and *nodeTypeMergers* (line 9-16), i.e. the configuration file holds the qualified names and implementation classes of Node Types for both matching and merging. In each section an unlimited number of *specificMatcher* respectively *specificMerger* elements is located (line 18-23 in Listing 7.1 and line 25-30 in Listing 7.2). The *specificMatcher* and *specificMerger* elements contain two qualified names (*QName* elements) consisting of a namespace URI and a local part as well as an implementation class element of type string.

Schema of a Node-Type-specific configuration part 2

```
    ...
  <xs:complexType name="tSpecificNodeTypeMerger">
    <xs:sequence>
      <xs:element name="specificMerger" type="tNodeTypeDetail"
        maxOccurs="unbounded" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tNodeTypeQName">
    <xs:sequence>
      <xs:element name="localPart" type="xs:string" />
      <xs:element name="namespaceURI" type="xs:string" />
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="tNodeTypeDetail">
    <xs:sequence>
      <xs:element name="QName" type="tNodeTypeQName" minOccurs="2"
        maxOccurs=″2″ />
      <xs:element name="implementationClass" type="xs:string" />
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

Listing 7.2: Schema of Node-Type-specific configuration part 2

Listing 7.3 shows one corresponding XML configuration file where a Tomcat Application Server Node Type is registered for both matching and merging. The fact that each of the pairwise QName elements is identical indicates that the implementation class is used for matching respectively merging two Node Templates with identical Node Types. Otherwise the QNames would be different. The *namespaceURI* contains the *targetNamespace* and the *localPart* the *id* of the respective Node Type. The *implementationClass* element contains the qualified class name of the corresponding Java class that has to be created by the *TOSCAMatchingFactory* respectively *TOSCAMergingFactory*.

Example of a Node-Type-specific configuration file

```xml
<?xml version="1.0" encoding="UTF-8"?>
<NodeTypeSpecificImplementations>
  <nodeTypeMatchers>
    <specificMatcher>
      <QName>
        <localPart>TomcatApplicationServer</localPart>
        <namespaceURI>http://myTemplate.de/test</namespaceURI>
      </QName>
      <QName>
        <localPart>TomcatApplicationServer</localPart>
        <namespaceURI>http://myTemplate.de/test</namespaceURI>
      </QName>
      <implemenationClass>
        match.typespecific.TomcatApplicationServerMatcher
      </implemenationClass>
    </specificMatcher>
  </nodeTypeMatchers>
  <nodeTypeMergers>
    <specificMerger>
      <QName>
        <localPart>TomcatApplicationServer</localPart>
        <namespaceURI>http://myTemplate.de/test</namespaceURI>
      </QName>
      <QName>
        <localPart>TomcatApplicationServer</localPart>
        <namespaceURI>http://myTemplate.de/test</namespaceURI>
      </QName>
      <implemenationClass>
        merge.typespecific.TomcatApplicationServerMerger
      </implemenationClass>
    </specificMerger>
  </nodeTypeMergers>
</NodeTypeSpecificImplementations>
```

Listing 7.3: Example of a Node-Type-specific configuration file

8 Implementation of the Framework

The following chapter discusses the prototypical implementation of the researched concepts of Chapter 5 and Chapter 6 as well as the architectural decisions of Chapter 7. Section 8.1 gives some general remarks about the implementation and the used technologies. Section 8.2 discusses the implementation of the framework's basic data structures using JAXB while Section 8.3 details the implementation of the extensibility concept already described on the architectural level in the previous chapter.

8.1 General Remarks

The programming language used for the implementation of the TOSCAMerge framework is Java in version Java SE 6. Additional libraries used are Apache log4j 1.2.17 [3] for logging and JAXB 2.2 [31] for building the basic data structures as well as for loading and saving of the TOSCA Service Templates. Apache log4j is used to equip the framework with the possibility to write the processing steps of matching and merging into a log file for later comprehension. Thereby, the granularity of the logging details can be adjusted.

The build management tool used for the framework is Apache Maven [2]. Thus, the framework's workspace is structured as demanded by Maven and its dependencies are specified in an enclosed *pom.xml* file. The paths to the type-specific configuration files are located in a Java properties file [34], denoted by *TOSCAMerge.properties* that must be placed in the framework's classpath in order to conveniently retrieve the values via classloader.

A simple client is also part of the prototype. It invokes the stateless *TOSCAMergeService* and passes two loaded Service Templates loaded by an auxiliary Java class, denoted by *TOSCAFileHandler*, using JAXB. As the prototype is a plain Java implementation and not running in a Servlet container the service is not stateless in the sense of an HTTP connection but the Correspondences can be transferred to another instance of the service for merging. The intermediary result, i.e. a list of Correspondences can be reviewed and changed as required by the requirements of Section 4.2. Thereby, flags indicating an *added* or *deleted* Correspondence can be used to indicate the *TOSCAMergeService* how to update the list of Correspondences and the internal state such as the set of innate ones.

Furthermore, the framework contains a class, denoted by *TOSCAServiceTemplateMerger* that unifies the Types of the two Service Template documents into one schema and adds the Node and Relationship Types that are also part of the Service Templates and not of the Topology Templates into a merged document while avoiding duplicate Types.

8.2 Implementation of the TOSCA Data Structures

The TOSCA specification available for this thesis is *Version 1 Working Draft 05.* A corresponding XML-schema file incorporating the specified data structures forms the basis for the prototypical implementation of the TOSCAMerge framework. In order to facilitate a convenient handling of the Service Templates the Java Architecture for XML Binding

(JAXB) is utilized. In contrast to XML-APIs such as Document Object Model (DOM) or Simple API for XML (SAX) [23] JAXB provides an abstraction from the XML elements. With the support of a provided *binding compiler* Java classes can be generated that correspond to the XML elements respectively complex types of an XML Schema document. These classes are annotated with the information which particular XML Element corresponds to which Java member variable. This is called a *binding*. With JAXB, XML Service Templates can be easily marshalleld, i.e. serialized and stored in an XML file, and unmarshalled, i.e. deserialized and loaded into the corresponding Java data structure.

The challenge for generating the basic data structures of the TOSCA Merge framework was the fact that many XML elements in the TOSCA Schema document are so-called anonymous types. Listing 8.1 depicts such an anonymous type, nested into the definition of Group Templates in this case. Butek and Kendrick [11] point out that anonymous types do not allow reuse and may cause problems when a binding must name them.

```
Example of an anonymous type in the TOSCA XML-schema
```

```
    ...
<xs:element name="Policies" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="Policy" type="tPolicy"
        maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
    ...
```

Listing 8.1: Example of an anonymous type in the TOSCA XML-schema

Additionally, the JAXB compiler transforms these types into static inner Java classes, e.g. a static inner class Policies in a class TGroupTemplate. To avoid this behavior the standard binding must be modified using a *Binding Customizations* specified in an additional binding file. A section of the binding file used in this work is shown in Listing 8.2

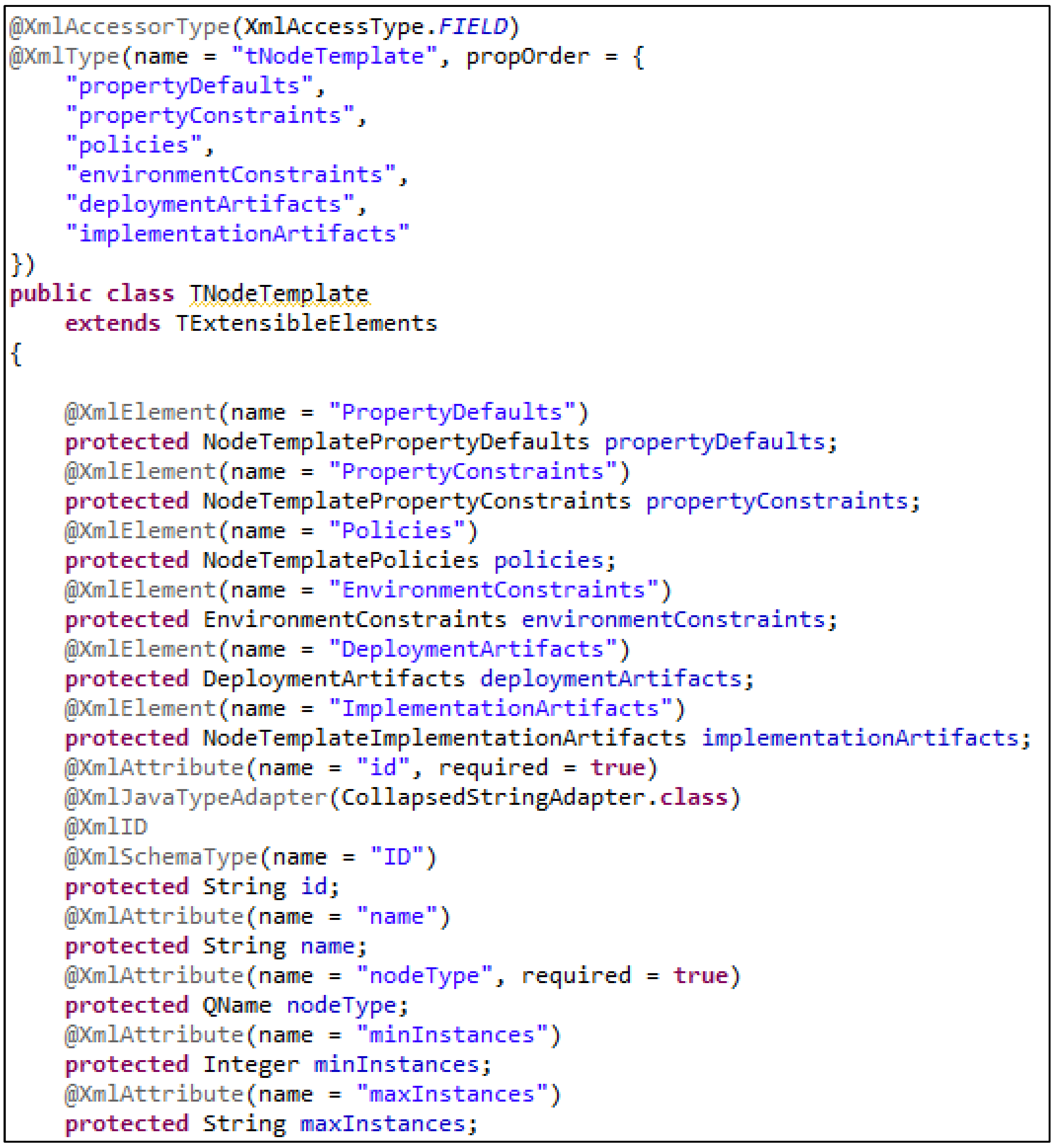
Section of the used JAXB Customization Bindings

```
 1  <jxb:bindings version="1.0"
 2    xmlns:jxb="http://java.sun.com/xml/ns/jaxb"
 3    xmlns:xs="http://www.w3.org/2001/XMLSchema"
 4    xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc">
 5
 6    <jxb:globalBindings localScoping="toplevel">
 7      <xjc:superClass
 8        name="de.toscamerge.generated.extension.BaseClass"/>
 9    </jxb:globalBindings>
10
11    <jxb:bindings schemaLocation="TOSCA-v1.0-wd05.xsd"
12        node="/xs:schema">
13      <jxb:bindings node="//xs:complexType[@name='tGroupTemplate']
14        /xs:complexContent/xs:extension/xs:sequence/xs:element
15          [@name='Policies']//xs:complexType">
16            <jxb:class name="GroupTemplatePolicies"/>
17    </jxb:bindings>
18      ...
```

Listing 8.2: Section of the used JAXB Customization Bindings

The *localScoping="toplevel"* attribute specifies that all complex types of the schema file should be generated as independent Java class instead of nested inner classes (line 6). However, the anonymous type containing the Policies as depicted in Listing 8.1 is used not only inside a Group Template definition but also inside Node Types and Node Templates. Thus, the binding compiler must be explicitly instructed to use different names for each generated Java class representing a Policy in order to make each class name unique. The lines 13-17 show the assignment of the class names by navigating to the affected elements using XPath expressions. Also visible in this listing is the assignment of a superclass which all other generated Java classes have to extend (line 7-8). It corresponds to the abstract *BaseClass*, holding information about the innate and collected Correspondences as well as the number of merges, introduced in Fig. 7.7 in the previous chapter. It is shown below in Fig. 8.2.

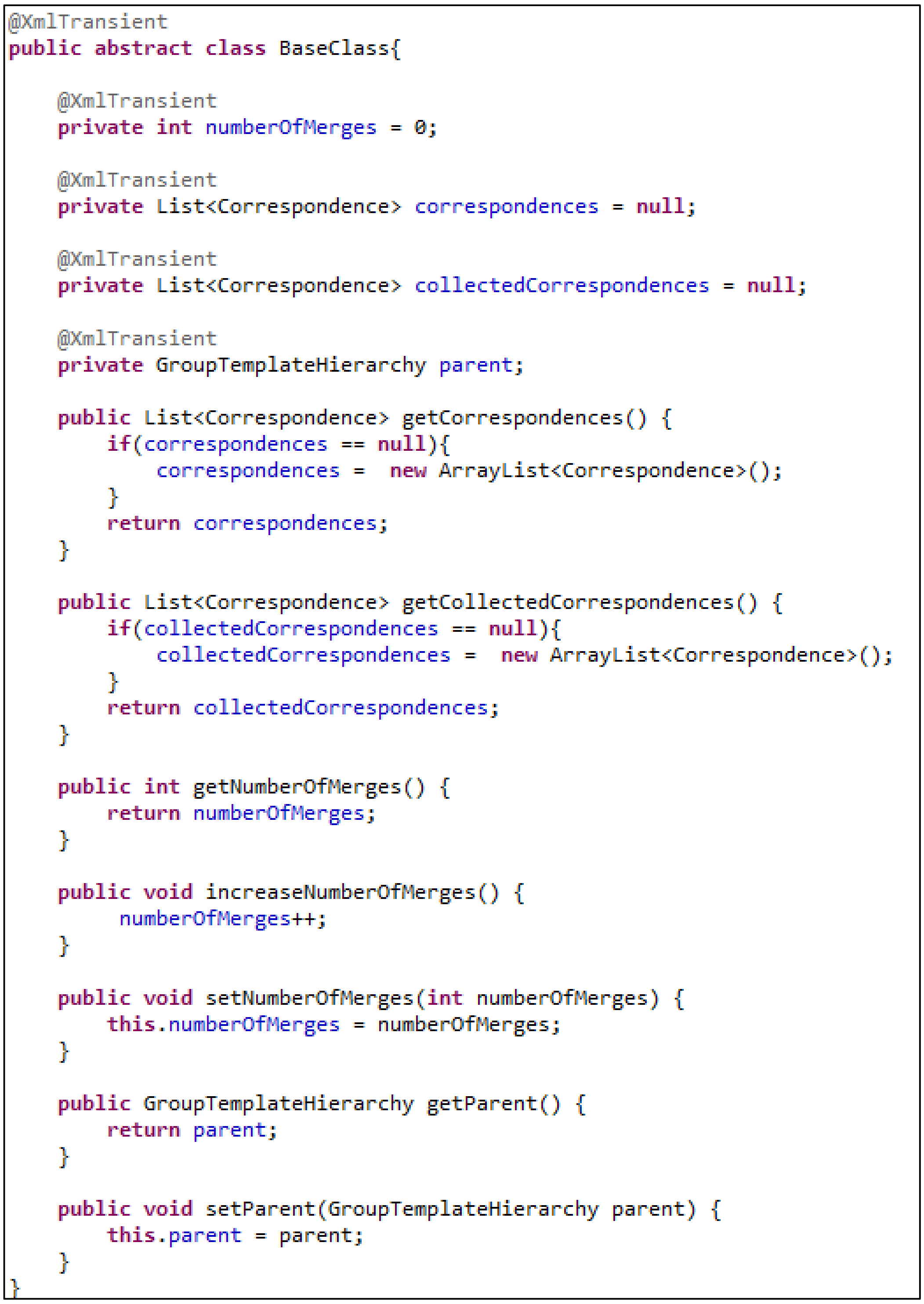
Fig. 8.1 depicts the generated Java class *TNodeTemplate* representing a Node Template and its properties with the corresponding binding annotations exemplarily. The anonymous type Policies had to be renamed into *NodeTemplatePolicies* to avoid naming collisions with other class names. Not visible in the figure are the getter- and setter-methods for the Node Template Properties.



```java
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "tNodeTemplate", propOrder = {
    "propertyDefaults",
    "propertyConstraints",
    "policies",
    "environmentConstraints",
    "deploymentArtifacts",
    "implementationArtifacts"
})
public class TNodeTemplate
    extends TExtensibleElements
{

    @XmlElement(name = "PropertyDefaults")
    protected NodeTemplatePropertyDefaults propertyDefaults;
    @XmlElement(name = "PropertyConstraints")
    protected NodeTemplatePropertyConstraints propertyConstraints;
    @XmlElement(name = "Policies")
    protected NodeTemplatePolicies policies;
    @XmlElement(name = "EnvironmentConstraints")
    protected EnvironmentConstraints environmentConstraints;
    @XmlElement(name = "DeploymentArtifacts")
    protected DeploymentArtifacts deploymentArtifacts;
    @XmlElement(name = "ImplementationArtifacts")
    protected NodeTemplateImplementationArtifacts implementationArtifacts;
    @XmlAttribute(name = "id", required = true)
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
    @XmlID
    @XmlSchemaType(name = "ID")
    protected String id;
    @XmlAttribute(name = "name")
    protected String name;
    @XmlAttribute(name = "nodeType", required = true)
    protected QName nodeType;
    @XmlAttribute(name = "minInstances")
    protected Integer minInstances;
    @XmlAttribute(name = "maxInstances")
    protected String maxInstances;
```

Fig. 8.1: Section of the Java class of a Node Template

The aforementioned *BaseClass* is depicted Fig. 8.2. The class itself and all of its variables are marked with the *@XmlTransient* annotation that indicates that it must be ignored during the marshalling from Java to XML, respectively, unmarshalling from XML to Java. This is necessary as this class is a concept introduced in the context of the TOSCAMerge framework and not in the TOSCA specification. Note that the Java implementation differs from the proposed algorithms regarding the usage of sets. As visible in Fig. 8.2 lists hold elements that occur multiple times instead of sets. This is also true for the generated JAXB classes. The implementation does not need the strict semantics of sets, respectively, is able to enforce them whenever necessary.



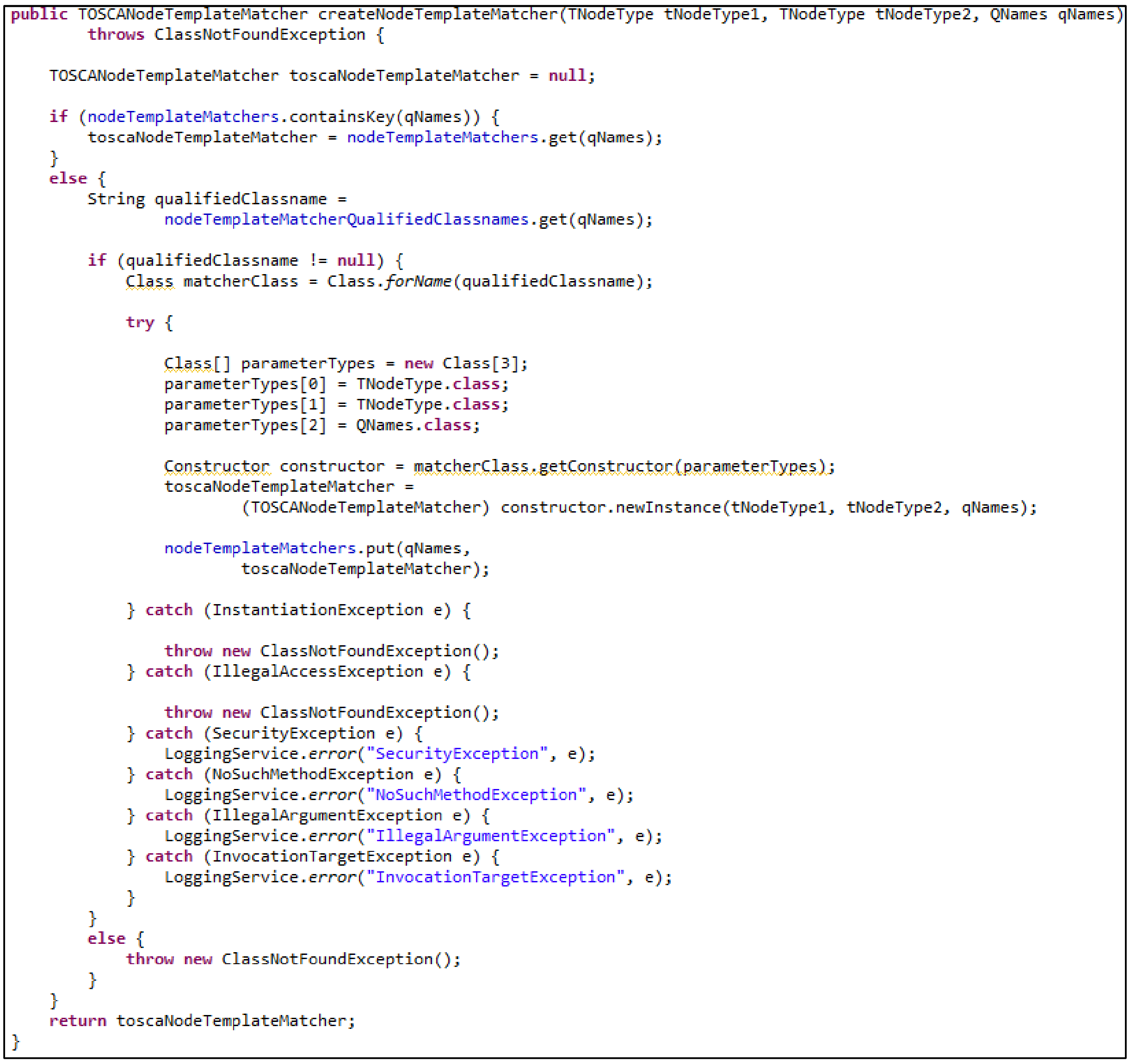
```java
@XmlTransient
public abstract class BaseClass{

    @XmlTransient
    private int numberOfMerges = 0;

    @XmlTransient
    private List<Correspondence> correspondences = null;

    @XmlTransient
    private List<Correspondence> collectedCorrespondences = null;

    @XmlTransient
    private GroupTemplateHierarchy parent;

    public List<Correspondence> getCorrespondences() {
        if(correspondences == null){
            correspondences =  new ArrayList<Correspondence>();
        }
        return correspondences;
    }

    public List<Correspondence> getCollectedCorrespondences() {
        if(collectedCorrespondences == null){
            collectedCorrespondences =  new ArrayList<Correspondence>();
        }
        return collectedCorrespondences;
    }

    public int getNumberOfMerges() {
        return numberOfMerges;
    }

    public void increaseNumberOfMerges() {
         numberOfMerges++;
    }

    public void setNumberOfMerges(int numberOfMerges) {
        this.numberOfMerges = numberOfMerges;
    }

    public GroupTemplateHierarchy getParent() {
        return parent;
    }

    public void setParent(GroupTemplateHierarchy parent) {
        this.parent = parent;
    }
}
```

Fig. 8.2: Class BaseClass

8.3 Implementation of the Extensibility Concept

8.3.1 Extensibility Concept

In Section 7.3 the architecture elements of the TOSCAMerge framework that are responsible for the adding of plugins were introduced. Exemplarily, this chapter will have a closer look at the implementation of the *TOSCAMatchingFactory* that uses a modified version of the Factory Method design pattern [21], [29].

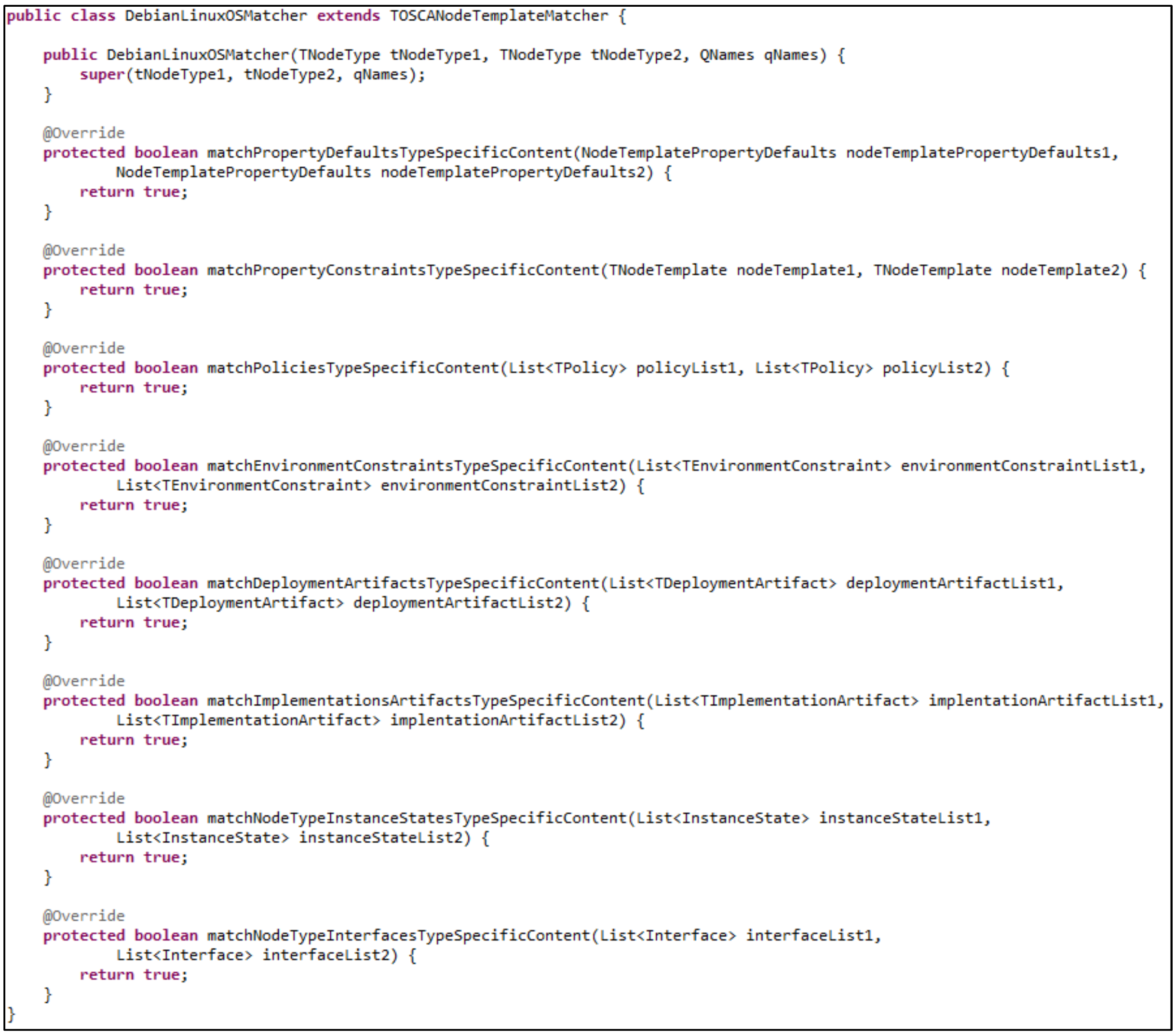


```java
public TOSCANodeTemplateMatcher createNodeTemplateMatcher(TNodeType tNodeType1, TNodeType tNodeType2, QNames qNames)
        throws ClassNotFoundException {

    TOSCANodeTemplateMatcher toscaNodeTemplateMatcher = null;

    if (nodeTemplateMatchers.containsKey(qNames)) {
        toscaNodeTemplateMatcher = nodeTemplateMatchers.get(qNames);
    }
    else {
        String qualifiedClassname =
                nodeTemplateMatcherQualifiedClassnames.get(qNames);

        if (qualifiedClassname != null) {
            Class matcherClass = Class.forName(qualifiedClassname);

            try {

                Class[] parameterTypes = new Class[3];
                parameterTypes[0] = TNodeType.class;
                parameterTypes[1] = TNodeType.class;
                parameterTypes[2] = QNames.class;

                Constructor constructor = matcherClass.getConstructor(parameterTypes);
                toscaNodeTemplateMatcher =
                        (TOSCANodeTemplateMatcher) constructor.newInstance(tNodeType1, tNodeType2, qNames);

                nodeTemplateMatchers.put(qNames,
                        toscaNodeTemplateMatcher);

            } catch (InstantiationException e) {

                throw new ClassNotFoundException();
            } catch (IllegalAccessException e) {

                throw new ClassNotFoundException();
            } catch (SecurityException e) {
                LoggingService.error("SecurityException", e);
            } catch (NoSuchMethodException e) {
                LoggingService.error("NoSuchMethodException", e);
            } catch (IllegalArgumentException e) {
                LoggingService.error("IllegalArgumentException", e);
            } catch (InvocationTargetException e) {
                LoggingService.error("InvocationTargetException", e);
            }
        }
        else {
            throw new ClassNotFoundException();
        }
    }
    return toscaNodeTemplateMatcher;
}
```

Fig. 8.3: Method createNodeTemplateMatcher of class TOSCAMatchingFactory

Fig. 8.3 shows the method *createNodeTemplateMatcher* of the *TOSCAMatchingFactory* that creates instances of the *TOSCANodeTemplateMatcher* subject to the qualified names of two Node Types. First it is checked if the corresponding instance extending the class *TOSCA-NodeTemplateMatcher* has already been created and stored in a hash map denoted by the blue painted *nodeTemplateMatchers*. This step avoids creating a new *TOSCANodeTemplate-Matcher* and, thus, unnecessary resource consumption.

If the instance of *TOSCANodeTemplateMatcher* appropriate for the two Node Types' qualified names has not yet been created, the qualified name of the class extending *TOSCA-NodeTemplateMatcher* and implementing the Node-Type-specific functionality of the passed Node Types is retrieved from yet another hash map built at creation time of the *TOSCAMatchingFactory*. The qualified name, i.e. the name of the class and its package is used to invoke the corresponding instance using the Java Reflection API [23].



```java
public class DebianLinuxOSMatcher extends TOSCANodeTemplateMatcher {

    public DebianLinuxOSMatcher(TNodeType tNodeType1, TNodeType tNodeType2, QNames qNames) {
        super(tNodeType1, tNodeType2, qNames);
    }

    @Override
    protected boolean matchPropertyDefaultsTypeSpecificContent(NodeTemplatePropertyDefaults nodeTemplatePropertyDefaults1,
            NodeTemplatePropertyDefaults nodeTemplatePropertyDefaults2) {
        return true;
    }

    @Override
    protected boolean matchPropertyConstraintsTypeSpecificContent(TNodeTemplate nodeTemplate1, TNodeTemplate nodeTemplate2) {
        return true;
    }

    @Override
    protected boolean matchPoliciesTypeSpecificContent(List<TPolicy> policyList1, List<TPolicy> policyList2) {
        return true;
    }

    @Override
    protected boolean matchEnvironmentConstraintsTypeSpecificContent(List<TEnvironmentConstraint> environmentConstraintList1,
            List<TEnvironmentConstraint> environmentConstraintList2) {
        return true;
    }

    @Override
    protected boolean matchDeploymentArtifactsTypeSpecificContent(List<TDeploymentArtifact> deploymentArtifactList1,
            List<TDeploymentArtifact> deploymentArtifactList2) {
        return true;
    }

    @Override
    protected boolean matchImplementationsArtifactsTypeSpecificContent(List<TImplementationArtifact> implentationArtifactList1,
            List<TImplementationArtifact> implentationArtifactList2) {
        return true;
    }

    @Override
    protected boolean matchNodeTypeInstanceStatesTypeSpecificContent(List<InstanceState> instanceStateList1,
            List<InstanceState> instanceStateList2) {
        return true;
    }

    @Override
    protected boolean matchNodeTypeInterfacesTypeSpecificContent(List<Interface> interfaceList1,
            List<Interface> interfaceList2) {
        return true;
    }
}
```

Fig. 8.4: Plugin-class for matching Node Templates with identical Node Types

Java Reflection is an element of the Java programming language that allows loading and instantiating classes whose code is not available during compile time. This applies in particular to frameworks offering the possibility of adding new plugins. An essential part of Java Reflection is the class *Class* capable of loading arbitrary classes into the Java Virtual Machine (JVM) using the static method *forName*. With the object of class *Class*, in Fig. 8.3 denoted by *matcherClass*, the constructor for the actual class to instantiate can be obtained and configured. The constructor object is then used to create an instance of the designated TOSCANodeTemplateMatcher instance that is stored in the hash map using the qualified names as key.

Fig. 8.4 shows the basic structure of a Node-Type-specific plugin that matches to identical Node Types representing a Debian Linux operating system. The respective matching methods for the Node Template Properties are yet empty in this example but can be used to implement the Node-Type-specific functionality for matching or for the invocation of third-party-functionality such as policy matching engine.

8.3.2 Adding New Plugins to the TOSCAMerge Framework

The following section describes the adding of new plugins to the TOSCAMerge framework and gives guidelines how to proceed.

(1) First a new Java class has to be created extending either *TOSCANodeTemplateMatcher* for matching Node Templates Properties or *TOSCARelationshipTemplateMatcher* for matching Relationship Templates Properties or *TOSCARelationshipTemplateMatchingHandler* for adding a new algorithm to compare Relationship Templates of a particular Relationship Type. A *TOSCAGroupTemplateMatcher* can also be added once. For adding type-specific merging functionality the new class must either extend *TOSCANodeTemplateMerger* or *TOSCARelationshipTemplateMerger or TOSCAGroupTemplateMerger.* Each plugin for matching and merging of Node Templates corresponds to two particular qualified names. If they are identical, the matching respective merging is conducted between two Node Templates having identical Node Types. Otherwise it is done between related Node Types. Relationship Templates and their handlers only correspond to one identical qualified name indicated by the targetNamespace and the id of a Relationship Type.

(2) To announce the new plugin to the framework the implementation class has to be added to the appropriate configuration file as depicted in Table 8.1. The first column *Extended class* indicates the abstract framework class the plugin extends to form a variation point. The second column Configuration file indicates the XML-file the plugin has to be added to and the third column XML element names the particular element the new plugin has to create in the configuration file.

Extended class	**Configuration file**	**XML element**
TOSCANodeTemplateMatcher	nodeTypeSpecificImplementations.xml	specificMatcher
TOSCARelationshipTemplateMatcher	relationshipTypeSpecificImplementations.xml	specificMatcher
TOSCARelationshipTemplateMatchingHandler	relationshipTypeSpecificImplementations.xml	specificMatcher
TOSCANodeTemplateMerger	nodeTypeSpecificImplementations.xml	specificMerger
TOSCARelationshipTemplateMerger	relationshipTypeSpecificImplementations.xml	specificMerger
TOSCAGroupTemplateMatcher	groupTemplateImplementation.xml	specificMatcher
TOSCAGroupTemplateMerger	groupTemplateImplementation.xml	specificMerger

Table 8.1: Necessary elements to create a new plugin

Adding new Node Types to the list of Types that are not considered when matching and merging is similarly easy. The Node Type has to be added to the XML configuration file *notMatchedTypes.xml* by inserting a new *notMatchedType* element containing the namespace and id of the Node Type.

9 Evaluation of the Algorithms and the Implemented Concepts

The following chapter addresses the goals for the concepts made in Section 4.2 and evaluates the proposed algorithms of Chapter 5 and 6. First, an evaluation against the general requirements is conducted in Section 9.1. Then in Section 9.2, the algorithms' properties and their computational complexity are analyzed. Section 9.3 reviews the creation of a set of example TOSCA Service Templates and picks up the motivational scenario from the introductory chapter. Finally, the merging of more than two Topology Templates is discussed briefly in Section 9.4

9.1 Evaluation of the Findings Regarding the General Requirements

The following section evaluates the proposed concepts and algorithms for matching and merging of Topology Templates with regard to the general requirements identified in Section 4.2. The general requirements are *element preservation*, *relationship preservation*, *extraneous item prohibition*, *property preservation*, *value preference, semantically correct results* and the *inclusion of domain-specific knowledge.*

The *element preservation* requirement is heeded as the merging algorithm presented in Listing 6.1 takes the left-hand side Topology Template and adds every merged Node Template to it. Moreover, it makes sure that every Node Template that does not correspond to another one is also added to the left-hand Topology Template in the end. The algorithm for Group Templates Listing 6.16 also fulfills this requirement.

The *relationship preservation* requirement is very similar to *element preservation.* This requirement is fulfilled by the *RelationshipTemplateMergingHandler* Listing 6.10. All the merged Relationship Templates are added to the left-hand side Topology Template either directly on Nesting Level 0 or inside a Group Template.

All the proposed functions adhere to the *extraneous item prohibition* requirement. No additional elements are created that were not part of the input Topology Templates.

The *match* and *merge* functions for the Properties of Node, Relationship, and Group Templates consider all properties defined in the TOSCA specification. If the matching or merging of a particular property cannot be conducted generically, it is delegated to a type-specific plugin. Thus, the *property preservation* requirement that the properties of each element must be preserved in the merged result and a merged element must not have unified properties that contradict its original semantics is fulfilled.

The value preference requirement that demands the definition of a preferred model is used throughout all the concepts and algorithms. Whenever two property values are equipollent the left-hand side value is chosen.

To fulfill the *semantically correct results* requirement the matching concepts and algorithms consider cases when two elements must not be matched in order to avoid invalid semantics.

Examples of this are the handling of Relationship Templates with HostedOn and Dependency semantics.

The last general requirement was the *inclusion of domain-specific knowledge.* This requirement is fulfilled by the introduction the extendable TOSCAMerge framework that uses type-specific plugins that implement domain-specific knowledge how to handle the Properties of Node, Relationship and Group Templates.

9.2 Discussion of the Proposed Algorithms

This section discusses the developed algorithms and their implementation with regard to the requirements identified in Section 4.2. The requirements were the following: *termination of the algorithms*, *deterministic result*, *practicable computational complexity* and the *assessability of the intermediary results.* Furthermore, the properties of the algorithms are discussed.

9.2.1 General Properties of the Proposed Algorithms

The proposed algorithms always terminate after having matched all elements of two Topology Templates and merged all identified Correspondences. Thus, *the termination of the algorithms* requirement is fulfilled. The same is true for the requirement of having a *deterministic result* provided the input is identical. The computational complexity is evaluated and discussed in the next sections.

Both proposed algorithms for matching and merging are no classical graph algorithms such as the Dijkstra's algorithm [22] for finding the shortest ways from a node to all other nodes. The reason for this lies in the nature of the TOSCA specification that defines a XML grammar to describe the graph-like structure of an IT environment. The limitation of modeling a graph with an XML grammar is that there is no viable way of directly specifying all incident edges and, thus, all adjacent nodes such as in an adjacency list [40]. Of course, one could define a start node XML element that nests all incident edges and adjacent nodes inside. However, even a small number of nodes and edges would lead to an overly complex XML document. Instead the TOSCA specification models the edges of a TOSCA graph, i.e. the Relationship Templates, as entities that know their source and target node (see Section 2.3.1). Therefore, the algorithms proposed in the Chapters 5 and 6 operate on separate, unordered sets that contain the nodes and edges. To find the edges incident to a particular node essentially the whole set has to be searched linearly and every source and target element compared to the particular node. The implementation in Java does not need the restrictions of sets and uses lists holding the TOSCA elements.

Another general property of the proposed algorithms that has to be discussed is that they belong to the class of the so-called *greedy* algorithms [48], [40]. These algorithms try to find a globally optimal solution by iteratively extending the current solution by the next best local solution without assessing the global context. Solutions that are found are not revised even if better ones arise in a future step. The advantage of this class of algorithms is that they are relatively simple to design and efficient to implement. The disadvantage is that often only a local optimum, albeit a correct one, is found rather than the desired global optimum. With regard to the matching of Topology Template in this thesis this becomes apparent when looking at the proposed matching cases of Relationship Templates with Communication and Dependency semantics depicted in Fig. 5.8 and Fig. 5.9 in Section 5.2.1. These cases assume that some Node Template Correspondences already exist and that it is subject to them if another Correspondence can be created or not. This always leads to a correct matching and avoids invalid semantics in the Topology Template, however, the possibility that the new Correspondence would contribute to a better solution than the already existing ones cannot be eliminated. That means the relative position of Node Templates in the Topology Template has an influence on the quality of the solution.

A correct but not necessarily globally optimal solution is also the result of the proposed merging algorithm. The merging algorithm proposed in Listing 6.1 processes the Node Template Correspondences in the encountered order they are stored in the mapping. It uses the subroutine *hasCorrectNumberOfCorrespondences* depicted in Listing 6.4 and Listing 6.5 to evaluate if two Node Templates may still be merged with regard to other Node Templates they have already been merged with. This approach may prevent two Node Templates from being merged even if the new merging would yield a better solution than an already conducted one. This is due to the fact that one or both of the current Node Templates have already been unified with a Node Template that has no Correspondence to one of the current Node Templates under consideration. This applies to the merging of Relationship and Group Templates as well. Thus, the relative position of the elements to be merged in the Topology Template influences the quality of the solution again.

With regard to the *greedy* characteristics of the proposed algorithms the identification of the *assessability of the intermediary results* requirement of Section 4.2 gets a further justification. Although a result of the matching and merging can be done automatically a human inspection of the set of created Correspondences that possesses only local optimality can be improved by adding or deleting of correspondences or rearrange their order. This could improve the result to a more optimal solution. The design of the TOSCAMerge framework allows for the review of the generated Correspondences inside and between Topology Templates and, thus, fulfills the *assessability of the intermediary results* requirement.

9.2.2 Complexity Considerations of the Matching Algorithm

The following section analyzes the computational complexity of the proposed matching algorithm and its subroutines. It is assumed in each case that both the inside matching as well as the matching between the different Topology Templates is conducted. m denotes the number of elements in the left-hand side Topology Template and n the number of elements in the right-hand side Topology Template.

Preparation Steps Prior to the Actual Matching

In the function *buildRelationshipTemplateSet* introduced in Listing 5.24 all Relationship Templates from both Topology Templates are stored in an additional set for more convenient processing. Additionally, the parent GroupTemplateHierarchy elements are assigned. The computational complexity including the search through all Group Templates is the following:

$$\mathrm{O}(n+m+(n+m)) = O(2n+2m) \tag{1}$$

That means two iterations over all elements of both Topology Templates.

High-level Matching Algorithm

The high level algorithm incorporating Group Templates has two main loops that compare every Node Template of left-hand side with every Node Template in right-hand side Topology Template. The exclusion of 1:n, n:1 and m:n mapping characteristics avoids the necessity to compare every possible subset of Node Templates of one Topology Template with every possible subset of Node Templates of another Topology Template. Instead of $2^m \times 2^n$ comparisons and therefore $O(2^m \times 2^n)$ time complexity at most $m \times n$ comparisons are needed yielding a time complexity of $O(m \times n) \cong O(n^2)$ with m being the number of Node Templates in the first Topology Template and n the number of Node Templates in the second Topology Template. However, the inside matching must also be accounted for. A comparison of every element with every other element inside a Topology Template means a time complexity of $O(n^2 - n)$ where n is the number of Node Templates. With regard to both Topology Templates the time complexity is $O((n^2 - n) + (m^2 - m))$. Adding the complexity for the actual comparison between the two Topology Templates we yield a complexity of

$$O((n^2 - n) + (m^2 - m) + (m \times n)) \cong O(3n^2) \cong O(n^2) \tag{2}$$

That means the algorithm still has quadratic time complexity.

Auxiliary Subroutines and Matching of Group Templates

A number of subroutines for checking if Group Templates can be accessed, to find Group Template Correspondences, or if Node Templates may be matched across different Nesting Levels were proposed in Chapter 5. However, none of them requires iterating over the whole set of elements, instead only the set with the already found Node Template respectively Group Template Correspondences is traversed. So this multiplies only a constant factor to the overall time complexity and can be neglected. The same is true for the traversal of the Group Template Hierarchy.

Matching of Relationship Templates

The matching of Relationship Templates can add a significant amount of complexity to the overall processing. Let p be the number of Relationship Templates in the set 1 and q the number of Relationship Templates in set 2. If the Relationship Templates with HostedOn semantics are analyzed (see Listing 5.9), then each of the two sets has to be fully traversed in the worst case. Therefore the time complexity is the following one:

$$O(q+p) \tag{3}$$

As p and q must be smaller than the overall number of elements in the corresponding Topology Templates this means that only a constant factor, albeit a possibly large one, is multiplied with the previous complexity.

However, if Relationship Templates with Communication or Dependency semantics are matched, the computational complexity is higher. Similar to the matching of Node Templates every element of set 1 has to be matched with every element of set 2. This yields a complexity of

$$O(q \times p) \tag{4}$$

for each of the two Relationship Types.

So the overall computational complexity considering (1), (2), (3) and (4) is the following one:

$$O((2n + 2m) + (3n^2 \times (q + p) \times 2(q \times p)) \ \cong 4n + (3n^2 \times 2p^3)$$

Assuming p is smaller than the cardinality of the Topology Templates the following expression holds true:

$$4n + (3n^2 \times 2p^3) \cong n^2 \tag{5}$$

That means the overall matching has quadratic computational complexity.

9.2.3 Complexity Considerations of the Merging Algorithm

The following section will analyze the computational complexity of the proposed merging algorithms.

Preparation Steps prior to the Actual Merging

The merging algorithm does not need any additional preparation steps. The Relationship Template sets built the matching step are used.

High Level Merging Algorithm

The basic complexity of the merging algorithm proposed in Listing 6.1 is bound to the number of found Node Template Correspondences that have to be processed. Therefore, the complexity for merging will be $O(k)$ with k the number of found Node Template Correspondences. In the worst case when matching two Topology Templates where all Node Templates inside and between the Topology Templates match, $k = \frac{n^2-n}{2} + \frac{m^2-m}{2} + 1$, where n is the number of Node Templates in the first Topology Template and m is the number of Node Templates in the second Topology Template. The additional 1 Correspondence is the one between the two Topology Templates already merged inside. With such a constellation the complexity would be

$$O(k) = O\left(\frac{n^2 - n}{2} + \frac{m^2 - m}{2} + 1\right) \cong O(n^2 + m^2 + 1) \cong O(2n^2) \cong O(n^2) \tag{6}$$

In a real life case k is expected to be much smaller than n^2.

In the subroutine *hasCorrectNumberOfCorrespondences* four sequential iterations over the innate and collected Node Template Correspondences are conducted. The two sets of innate outgoing Node Template Correspondences as well as the two sets of collected ones can only have a combined cardinality of k. Thus, the overall complexity for the function *hasCorrectNumberOfCorrespondences* is:

$$O(k+k) = O(2k) \tag{7}$$

The subroutine *reconnectEdges* proposed in Listing 6.2 iterates over the whole set of Relationship Templates with cardinality q build before. Thus it has the complexity of

$$O(q) \tag{8}$$

The subroutine *relocateEdges* only processes two Relationship Templates and, therefore, has a constant complexity of $O(1)$[19]

The subroutine *reconnectCorrespondences* iterates over all found Node Template Correspondences and, thus, has a complexity of

$$O(k) \tag{9}$$

Merging of Relationship Templates

The algorithm for merging of Relationship Templates was introduced in Listing 6.10. It iterates over the set of all Relationship Templates attached to each Node Template Correspondence in the function *handleRelationshipTemplates*. Let the cardinality of the set Relationship Template Correspondences be l, then the complexity is

$$O(l) \tag{10}$$

For the subroutine *hasCorrectNumberOfCorrespondences* the same assumptions as in the case of Node Template Correspondences are true, thus, the complexity is

$$O(l+l) = O(2l) \tag{11}$$

In the subroutine *reconnectCorrespondences* for Relationship Templates all Node Template Correspondences and the attached Relationship Template Correspondences have to be processed, thus, the complexity is:

$$O(k \times l) \tag{12}$$

Merging of Group Templates

The algorithm for merging of Group Templates was proposed in Listing 6.14 and Listing 6.15. It iterates over a set of Group Template Correspondences with cardinality g. Thus, the complexity for the main algorithm is

[19] However, the implementation in Java has a larger complexity as the evaluation of containment in a set must possibly iterate over the whole content set of the Group Template.

$$O(g) \tag{13}$$

Similar to the merging of Node Templates the two subroutines *hasCorrectNumberOfCorrespondences* and *reconnectCorrespondences* have the complexities

$$O(g+g) = O(2g) \tag{14}$$

and respectively

$$O(g) \tag{15}$$

The overall complexity of one iteration of the functions *performNodeTemplateMerging* and *performGroupTemplateMerging*, considering all equations from (6) to (15) is the following one , regardless of merging inside one or between two Topology Templates:

$$O\left(k \times \left(2k + q + k + l \times (2l + (\mathrm{k} \times l))\right) + g \times (2g + g)\right) \tag{16}$$

$$= O(k \times (3k + q + 2l^2 + kl^2) + 3g^2)$$

$$= O(3k^2 + qk + 2l^2k + k^2l^2 + 3g^2)$$

$$\cong k^2 + \ g^2$$

The other variables can be disregarded as they are assumed to be smaller than k^2. From the overall equation can be safely concluded that the merging has quadratic complexity for Node Template and Group Template Correspondences. In order to fully merge two Topology Templates three merge iterations have to be conducted, however, this does not change the overall estimated complexity. If the worst case scenario described with equation (6) occurs with $k = n^2$ the complexity could be $O(n^2 \times n^2 = O(n^4)$ with n the number of Node Templates. But as already pointed out this case seems very unlikely.

9.2.4 Overall Complexity

After having estimated both worst case computational complexities in equation (5) and (16), i.e. $O(n^2 + k^2 + g^2)$, with n the number of TOSCA elements, k the number of found Node Template Correspondences and g the number of found Group Template Correspondences and disregarding all constant factors, it can be concluded the merging of two Topology Templates can always be conducted with polynomial computation complexity and most often even with quadratic computational complexity. According to Saake and Sattler [40] these are complexity classes that allow the practical solving of problems if the input size n does not exceed a certain size, i.e. 2^{16}. Thus, the *practical computational complexity* requirement from Section 4.2 is also fulfilled. However, it must be noted that the Java implementation of the algorithms is slightly less efficient than the theoretical value that was derived here. The main reason lays in the necessity to possibly iterate over all elements of a list in order to evaluate if a given element is contained. For example the evaluation if a Policy element is already in a Policy list and needs to be replaced by a derived Policy element. The pseudo code algorithms conduct these steps in one pass, the real implementation is more complex. Since these lists of properties are much smaller than the overall list of Node, Relationship and Group Templates the impact on complexity can be disregarded. Another reason is the implementation as a stateless service. The state is passed between the client and the service as a list of Correspondences and the two Service Templates. Thus, when invoking another instance of the service after matching it has to calculate the two sets of overall Relationship Templates anew increasing the computational complexity. But again it is only increased by a constant factor that can be disregarded.

9.3 Creation of Sample TOSCA Service Templates

Goal (4) of the problem statement in Section 1.1 was the creation of a set of example TOSCA Service Templates to evaluate the prototypically implemented framework and the researched concepts. This has been done collaterally to development of the matching and merging concepts and the implementation of the prototype.

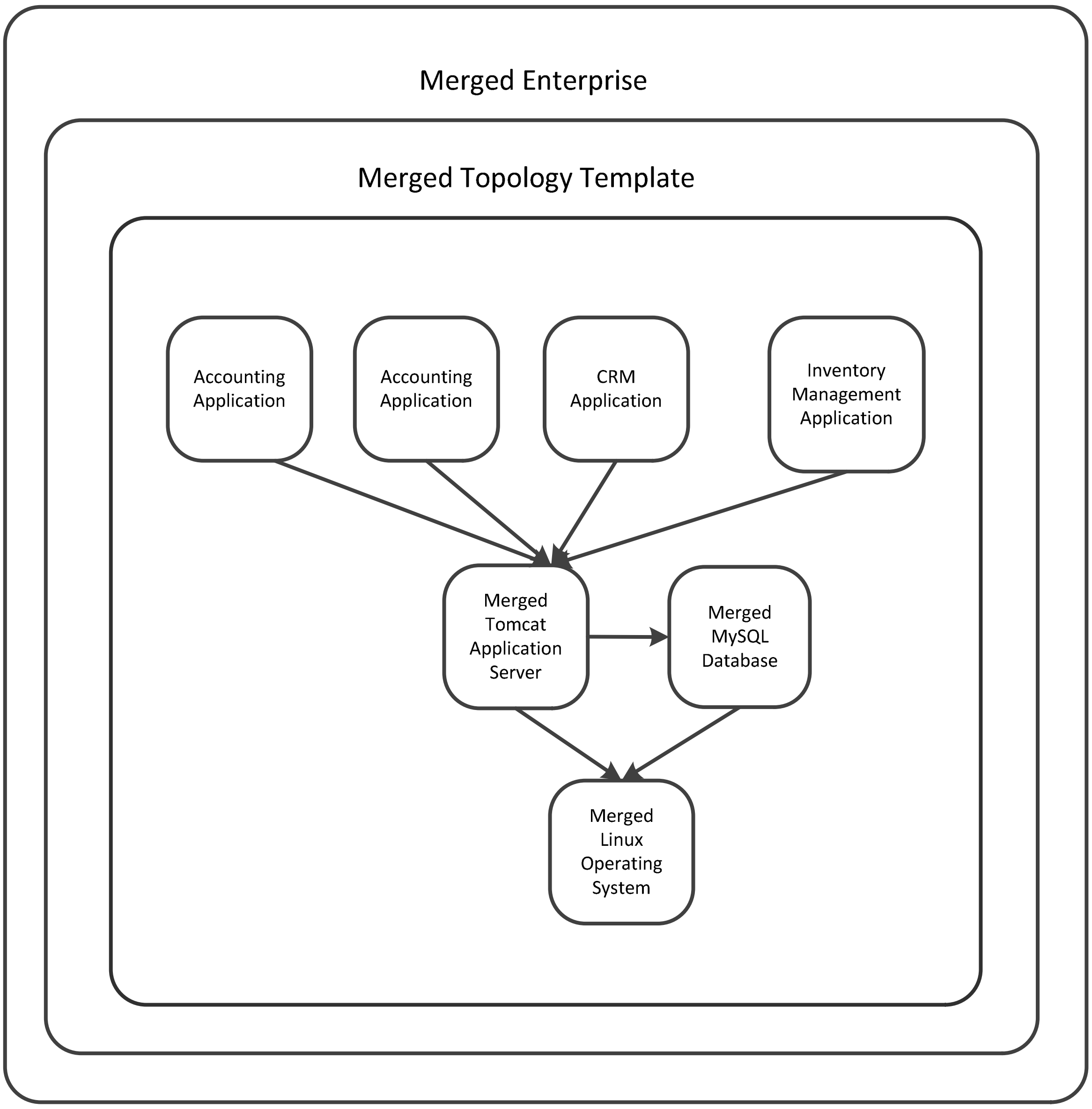


Fig. 9.1: The two merged Topology Template of the motivating scenario

For every identified case a corresponding left-hand and right-hand side Topology Template has been created to test the implemented algorithms.

In Section 1.2 a motivating scenario has been given to illustrate the necessity of developing concepts and algorithms for automatically finding similar elements inside and between two Topology Templates and for merging these elements and ultimately the two Topology Templates as a whole. To complete this thesis the motivating scenario is picked up at this point and the researched concepts are applied to it. The scenario assumed two equal enterprises to be merged in order to achieve economic synergies. The merged Topology Template in Fig. 9.1 shows a possible result of the two separate enterprise application environments modeled as Topology Templates after being merged automatically by using the TOSCAMerge framework. By definition the different applications cannot be merged because of their complex, individual business logic. Even in the case of the *Accounting Application* this is true. Thus, a domain expert has to decide which of the application to keep for future use. The *Tomcat Application Servers* and the *MySQL Databases,* however, have been merged automatically by using the type-specific plugins to merge the properties. In case of the *Operating Systems* that had different but related Node Types the corresponding type-specific plugin to the TOSCAMerge framework decided to use the *Linux Operating System* Node Type instead of the *Windows Operating System* Node Type.

9.4 Discussion of Merging of More Than two Topology Templates

The matching and merging of more than two Topology Templates was not a goal for this master's thesis. Nevertheless, at this point this aspect is discussed briefly. The proposed concepts and algorithms in this thesis can also be used for more than two Topology Templates by successively matching and merging the result of one complete step with yet another Topology Template. Leser and Naumann [26] describe this approach for the related area of schema integration and call it a *binary approach.* The advantages are the relatively easy implementation and the possibility to merge Topology Templates that are more important than others in an earlier step to assure a greater impact on the overall result. However, the question remains how to identify the most important ones and, thereby, the order of the matching and merging.

A second approach would be what Leser and Naumann call an n-ary approach that matches and merges all Topology Templates at once. The advantage is that no local decisions have to be made that prohibit the merging with Topology Templates that are considered at a later point in time. However, the computational complexity increases very fast with every additional Topology Template and leaves the region of practicable computational complexity. Furthermore, the proposed algorithms in this work are greedy algorithms so the consideration of more than two Topology Templates at once would only lead to a local optimum anyway.

10 Conclusion and Future Work

This master's thesis developed a concept for finding similar elements inside and between two Topology Templates by systematically exploring all different constellations TOSCA elements can take. After an extensive study of related work in Chapter 3 in the area of graph, process and schema matching and merging, the notion of a Correspondence between Node, Relationship and Group Templates was introduced to indicate elements that correspond to each other and can be merged. Moreover, the matching concept is automated by developing appropriate algorithms. Thereby, the incorporation of domain-specific knowledge is allowed for by the invocation of type-specific plugins that handle the matching of properties that cannot be conducted generically.

The second contribution of this thesis is the development of a concept and algorithms that use the determined Correspondences to merge the corresponding TOSCA elements. Both matching and merging adhere to a set of identified requirements such as preservation of all elements and properties found in the original Topology Templates or the design of algorithms with practical computational complexity. A third goal of this work was the design and prototypical implementation of the extendable TOSCAMerge framework that allows for a convenient integration of type-specific plugins covering the domain-specific handling of the TOSCA elements' properties. An important requirement at this juncture was an architecture that enables the assessability of the intermediary results, i.e. a human domain expert can evaluate the determined Correspondences and manipulate them if desired. The implementation was accompanied by the creation of a set of example TOSCA Service Templates testing the different matching and merging cases.

Finally, an extensive evaluation of the concepts and algorithms against the identified requirements completed this thesis in Chapter 9. Notably, the evaluation of the matching and merging algorithms revealed that they feature the properties of greedy algorithms. That means they always produce a semantically correct result as required but rather a local optimum is found than a globally optimal solution. However, the main advantage is the efficient implementation in the TOSCAMerge framework that has quadratic computational complexity in most cases.

Future work has to be conducted by the inclusion of the TOSCA management plans [32] in the merging concepts. The plans have to be adapted to the resulting topology of an IT service after being merged. Another topic is the integration and display of the determined Correspondences into Vino4TOSCA, a visual notation for TOSCA proposed by Breitenbücher et al. [9]. This would allow for the convenient manipulation of the intermediary results. In the same way, the integration in the existing web-based modeling tool for TOSCA, the Visual Editor for TOSCA (Valesca) [12] is conceivable.

Moreover, as Turau notes [48], the proposed greedy algorithms can be seen as starting point for the development of algorithms that find the globally optimal solution for matching and merging of Topology Templates at the expense of a higher computational complexity.

Likewise, the merging of more than two Topology Templates in an n-ary approach as briefly discussed in Section 9.4 is a topic for future research.

Bibliography

[1] Amazon Web Services LLC, "Amazon Elastic Compute Cloud (Amazon EC2)," [Online]. Available: http://aws.amazon.com/ec2/.

[2] Apache Software Foundation, "Apache Maven Project," [Online]. Available: http://maven.apache.org/

[3] Apache Software Foundation, "Apache log4j™ 1.2," [Online]. Available: http://logging.apache.org/log4j/1.2/.

[4] Apache Software Foundation, "Apache Tomcat," [Online]. Available: http://tomcat.apache.org/.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," 2009.

[6] C. Baun, M. Kunze, J. Nimis and S. Tai, Cloud Computing - Web-basierte dynamische IT-Services, 2. Aufl. ed., Heidelberg, Dordrecht, London, New York: Springer-Verlag, 2011.

[7] T. Binz, G. Breiter, F. Leymann and T. Spatzier, "Portable Cloud Services Using TOSCA," *IEEE Internet Computing,* vol. 16, no. 3, pp. 80 - 85, May - June 2012.

[8] J. A. Bondy and U. S. R. Murty, Graph Theory, New York: Springer, 2005.

[9] U. Breitenbücher, T. Binz, O. Kopp and F. Leymann, "Vino4TOSCA: A Visual Notation for Application Topologies based on TOSCA," in *Proceedings of the 20th International Conference on Cooperative Information Systems (CoopIS 2012),* Rome, 2012.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, Pattern-orientierte Software-Architektur - Ein Pattern-System, Bonn, Paris: Addison-Wesley-Longman, 1998.

[11] R. Butek and S. Kendrick, "IBM developerWorks - Web services hints and tips: avoid anonymous types," 30 August 2011. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-avoid-anonymous-types/index.html.

[12] CloudCyle Project, "Valesca – Visual Editor for TOSCA," [Online]. Available: http://www.cloudcycle.org/Valesca/

[13] R. Dijkman, M. Dumas and L. García-Bañuelos, "Graph Matching Algorithms for Business Process Model Similarity Search," *Business Process Management, LNCS,* vol. 5701, pp. 48-63, 2009.

[14] Distributed Management Task Force, Inc, "Open Virtualization Format (OVF)," [Online]. Available: http://www.dmtf.org/standards/ovf.

[15] J. Dunkel and A. Holitschke, Softwarearchitektur für die Praxis, Berlin, Heidelberg: Springer-Verlag, 2003

[16] M. M. Gala, E. Quintarelli and L. Tanca, "Graph Transformation for Merging User Navigation Histories," in *Applications of Graph Transformations with Industrial Relevance AGTIVE 2003, LNCS*, Berlin, Heidelberg, New York, 2004.

[17] Google Inc, "Google App Engine - Google Developers," 4 7 2012. [Online]. Available: https://developers.google.com/appengine/.

[18] Google Inc., "Google Apps for Business," [Online]. Available: http://www.google.com/intl/en/enterprise/apps/business/products.html#docs.

[19] F. Gottschalk, W. M. P. van der Aalst and M. H. Jansen-Vullers, "Merging Event-Driven Process Chains," *On the Move to Meaningful Internet Systems: OTM 2008, LNCS,* vol. 5331/2008, pp. 418-426, 2008.

[20] Q. Hardy, "The New York Times - Active in Cloud, Amazon Reshapes Computing," 27 August 2012. [Online]. Available: http://www.nytimes.com/2012/08/28/technology/active-in-cloud-amazon-reshapes-computing.html.

[21] C. Kinzel Filho, "Code Project - Factory Method + Reflection: Achieving Better Extensibility in Applications," 2008 August 31. [Online]. Available: http://www.codeproject.com/Articles/28977/Factory-Method-Reflection-Achieving-Better-Extensi.

[22] S. O. Krumke and N. Noltemaier, Graphentheoretische Konzepte und Algorithmen, 2. aktualisierte Aufl. ed., Wiesbaden: Vieweg+Teubner, 2009.

[23] G. Krüger, Handbuch der Java-Programmierung, 4., aktualisierte Aufl. ed., München: Addison-Wesley Verlag, 2006.

[24] J. M. Küster, C. Gerth, A. Förster and G. Engels, "Detecting and Resolving Process Model Differences in the Absence of a Change Log," in *Proceedings of the 6th International Conference on Business Process Management (BPM '08)*, Berlin, Heidelberg, 2008.

[25] M. La Rosa, M. Dumas, R. Käärik and R. Dijkman, "Merging business process models," in *Proceedings of the 2010 international conference on On the move to meaningful internet systems - Volume Part I (OTM'10)*, 2010.

[26] U. Leser and F. Naumann, Informationsintegration, Heidelberg: dpunkt Verlag, 2007.

[27] P. Mell and Grance, T., "The NIST Definition of Cloud Computing," National Institute of Standards and Technology, Gaithersburg, 2011.

[28] S. Melnik, H. Garcia-Molina and E. Rahm, "Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching," in *Proceedings of the 18th International Conference on Data Engineering (ICDE '02)*, Washington, DC, USA, 2002.

[29] N.N., "OODesign.com - Factory Pattern," [Online]. Available: http://www.oodesign.com/factory-pattern.html.

[30] N.N., "Open Cloud Manifesto," [Online]. Available: http://www.opencloudmanifesto.org/Open%20Cloud%20Manifesto.pdf.

[31] N.N., "Project JAXB," [Online]. Available: http://jaxb.java.net/.

[32] OASIS, "Topology and Orchestration Specification for Cloud Applications Version 1.0, Working Draft 05," 30 March 2012. [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca.

[33] OMG, "Unified Modeling Language: Infrastructure," [Online]. Available: http://www.omg.org/spec/UML/2.0/Infrastructure/PDF/.

[34] Oracle Corporation, "Java™ Platform Standard Ed. 6 - Class java.util.Properties," [Online]. Available: http://docs.oracle.com/javase/6/docs/api/java/util/Properties.html.

[35] Oracle Corporation, "MySQL Enterprise Edition," [Online]. Available: http://www.oracle.com/us/products/mysql/mysqlenterprise/index.html?ssSourceSiteId=ocomde.

[36] D. F. Parkhill, The Challenge of the Computer Utility, New Jersey: Addison-Wesley Educational Publishers Inc.,, 1966.

[37] T. Posch, K. Birken and M. Gerdom, Basiswissen Softwarearchitektur- Verstehen, entwerfen, wiederverwenden, 3. aktualisierte und erweiterte Auflage ed., Heidelberg: dpunkt.verlag, 2011.

[38] R. Pottinger and P. A. Bernstein, "Merging models based on given correspondences," in *Proceedings of the 29th international conference on Very large data bases - Volume 29 (VLDB '2003)*, Berlin, 2003.

[39] R. Reiter, "On closed world data bases," in *Readings in nonmonotonic reasoning*, San Francisco, Morgan Kaufmann Publishers Inc., 1987, pp. 300 - 310.

[40] G. Saake and K.-U. Sattler, Algorithmen und Datenstrukturen, Eine Einführung mit Java, Heidelberg: dpunkt Verlag, 2010.

[41] Salesforce.com Inc., "What is Force.com?," [Online]. Available: http://www.force.com/why-force.jsp.

[42] A. Scherp and S. Boll, "Framework-Entwurf," in *Handbuch der Software-Architectur*, 2. überbeitete und erweiterte Aufl. ed., R. Reussner and W. Hasselbring, Eds., Heidelberg, dpunkt.verlag, 2009, pp. 383-405.

[43] L. Schubert, "The future of Cloud Computing, Opportunities for European Cloud Computing beyond 2010," Commission of the European Communities, 2010.

[44] J. A. Schumpeter, Capitalism, Socialism and Democracy, New York: Harper, 1942.

[45] S. Segura, D. Benavides, A. Ruiz-Cortés and P. Trinidad, "Automated Merging of Feature Models Using Graph Transformations," in *Generative and Transformational Techniques in Software Engineering II, LNCS*, Berlin, Heidelberg, 2007.

[46] S. Sun, A. Kumar and J. Yen, "Merging workflows: a new perspective on connecting business processes," *Decision Support Systems,* vol. 42, no. 2, pp. 844 - 858, November 2006.

[47] P. Tittmann, Graphentheorie, Eine anwendungsorientierte Einführung, München: Carl Hanser Verlag, 2011.

[48] V. Turau, Algorithmische Graphentheorie, München: Oldenbourg Verlag, 2004.

[49] E. van der Vlist, XML Schema, Sebastopol: O'Reilly Media, Inc., 2002.

[50] W3C, "XML Schema Part 0: Primer Second Edition," 28 October 2008. [Online]. Available: http://www.w3.org/TR/xmlschema-0/#any.

[51] S. Weerawarana, F. Curbera, F. Leymann, T. Storey and D. F. Ferguson, Web Services Platform Architecture - SOAP, WSDL, WS-Policy, WS-Adressing, WS-BPEL, WS-Reliable Messaging, and More, Upper Saddle River: Person Education, Inc., 2005.

All links were last followed on October 16, 2012.

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 anderweitigen 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 17. Oktober 2012

(Andreas Weiß)