UNIVERSITY OF PISA

Department of Computer Science

Master Degree Program in Computer Science

MASTER THESIS

Matching cloud services with TOSCA

SUPERVISOR

Prof. Antonio BROGI

CANDIDATE

Jacopo SOLDANI

Academic Year 2012-2013

To those who made this possible,
both who can be here and understand
and who cannot.

Abstract

The OASIS TOSCA specification aims at enhancing the portability of cloud-based applications by defining a language to describe and manage service orchestrations across heterogeneous clouds. A service template is defined as an orchestration of typed nodes, which can be instantiated by matching other service templates. In this thesis, after defining the notion of *exact matching* between TOSCA service templates and node types, we define three other types of matching (*plug-in*, *flexible* and *white-box*), each permitting to ignore larger sets of non-relevant syntactic differences when type-checking service templates with respect to node types. We also describe how service templates that plug-in, flexibly or white-box match node types can be suitably adapted so as to exactly match them.

Contents

List of Figures

List of Listings

Chapter 1

Introduction

How to deploy and manage, in an efficient and adaptive way, complex multi-service applications across heterogeneous cloud environments is one of the problems that have emerged with the cloud revolution [30]. Currently, migrating (parts of) an application from one cloud to another is a costly and error-prone process that has to be performed manually. Part of (if not the whole) application must be stopped in order to migrate services (possibly along with data) to a different cloud and to restart them, taking care of synchronizing and maintaining the interoperability with the rest of the application. As a result, cloud users tend to end up locked into the cloud platform they are using since it is practically unfeasible for them to migrate (parts of) their application across different clouds platforms [29].

In this scenario, OASIS recently created a Technical Committee on *Topology and Orchestration Specification for Cloud Application* (TOSCA), whose goal is to ease the portability of cloud-based applications by defining a language to describe and manage service orchestrations across heterogeneous clouds. The first specification of TOSCA [25] defines a XML-based lanugage that permits to specify (in a vendor-agnostic way) topology and behaviour of complex multi-cloud applications as service

templates that orchestrate typed nodes.

The expected impact of TOSCA on cloud service portability is well explained in [18]. Let us think of similar attempts to deal with the lifecycle of complex man-made structures (like skyscrapers or bridges). Such structures are constructed, modified, maintained, and even dismantled using industry standard descriptions and manifests. The focus of these documents is primarily at the conceptual level of the building itself, its principal components, construction and maintenance, and not on how those components (like a furnace or elevator) are actually built themselves. Similarly, TOSCA is designed to support the definition of a common, machine-readable language for service description, maintenance, and operational management (which are the best practices needed to support cloud services across their lifetime). In other words, TOSCA enables the creation of an eco-system in which cloud service developers can describe (and model) the principal components, characteristics and requirements of a service in a standardized fashion so that the service can be understood, installed (deployed) or removed (undeployed) by different types of TOSCA-compliant cloud providers with very little additional effort.

As stated in the TOSCA primer [26], node types can be made concrete by substituting them by a service template. However, while the matching between service templates and node types is mentioned with reference to an example ([26], page 35):

> «*Service template ST may substitute node type N because the boundary of ST matches all defining elements of N: all properties, operations, requirements and capabilities of ST match exactly those of N.*»,

no formal definition of *matching* is given either in [25] or in [26]. A definition of matching is employed in [33] to merge TOSCA services by matching entire portions of their topology templates. The definition of

matching employed in [33] is however very strict, as two service components are considered to match only if they expose the same qualified name.

The objective of this work is to contribute to the TOSCA specification by first providing a formal definition of the notion of *exact matching* between TOSCA *ServiceTemplate* and *NodeType* elements, and by then extending such definition in order to provide three other types of matching (*plug-in*, *flexible* and *white-box*), each permitting to ignore larger sets of non-relevant syntactic differences when type-checking service templates with respect to node types. More precisely:

- the *plug-in* matching extends the *exact* one by considering a service that "require less" and "offers more" than a node type compatible with the latter;
- the *flexible* matching in turn extends the *plug-in* one by employing ontologies to check whether differently named features are semantically equivalent (so as to ignore non-relevant syntactic differences);
- the *white-box* matching in turn extends the *flexible* one by searching missing (equivalent) features inside the service topology. It still employs ontologies to check whether differently named features can be considered semantically equivalent. Furthermore, it employs a recursive algorithm to detect available compositions of operations which are semantically equivalent to needed (missing) operations.

To allow exploiting the new notions of matching not only during type-checking but also for node instantiation, we describe how a service template that *plug-in*, *flexibly* or *white-box* matches a typed node can be suitably adapted so as to exactly match it.

The results presented in this thesis intend to contribute to the formal definition of TOSCA. The different types of matching defined in this thesis can be fruitfully integrated in the TOSCA implementations that are

currently under development (such as [27] and [32]) in order to enhance their type-checking capabilities. More in general, the presented definitions of matching can be exploited to implement type-checking mechanism over service descriptions by taking into account, beyond functional features, also requirements, capabilities, policies, and properties.

It is worth pointing that implementing our matching notions (e.g., as a plug-in of TOSCA IDEs) will contribute to cloud service portability and multi-cloud service development. Indeed, with the availability of such an implementation, a cloud service developer will have the possibility to:

- employ more available (adapted) cloud services instead of developing her application's encompassed components,
- migrating more application's components across heterogeneous clouds by changing the used available (adapted) cloud services, and
- choose between more different cloud service providers the one which provides the compatible service with the best quality-price ratio.

Outline

The rest of the document is organized as follows:

Chapter 2 provides an overview of the TOSCA specification [25].

Chapter 3 starts by formalizing the TOSCA [25] notion of (black-box) *exact* matching. It then proceeds showing two other ways to match services from a black-box viewpoint (*plug-in*, *flexible*). Along with these matching notions, a way to adapt available service templates (in order to make them exactly match the desired node types) is provided.

Chapter 4 extends the matching notion of Chapter 3 by moving the viewpoint to a white-box one. A way to adapt service templates which *white-box* match the desired node types is given too.

Chapter 5 shows a partial implementation of the matching procedure in order to demonstrate the feasibility of the notions proposed in this thesis.

Chapter 6 summarizes the contributions of this thesis, discusses the work in related research fields and provides some concluding remarks.

Chapter 2

Background: TOSCA

The Topology and Orchestration Specification for Cloud Application (TOSCA) [25] is an XML-based language and metamodel which can be used to describe IT services. The main goal of the TOSCA specification is to allow a description of composite cloud-based applications and of their management in a modular and portable fashion.

The creator of a cloud service defines it in a so-called *service template* (Figure 2.1). This template is composed by:

- a *topology template*, a graph in which typed nodes represent service's components and typed relationships connect and structure nodes into the topology, and by
- *plans*, workflows used to describe managing concerns.

The following sections describe the most important elements of the TOSCA specification [25]. Before describing the TOSCA syntax (Section 2.2), we proceed by showing the major use cases supported by this specification (Section 2.1). Once all the basic concepts are given, we provide a complete TOSCA cloud service example (Section 2.3).

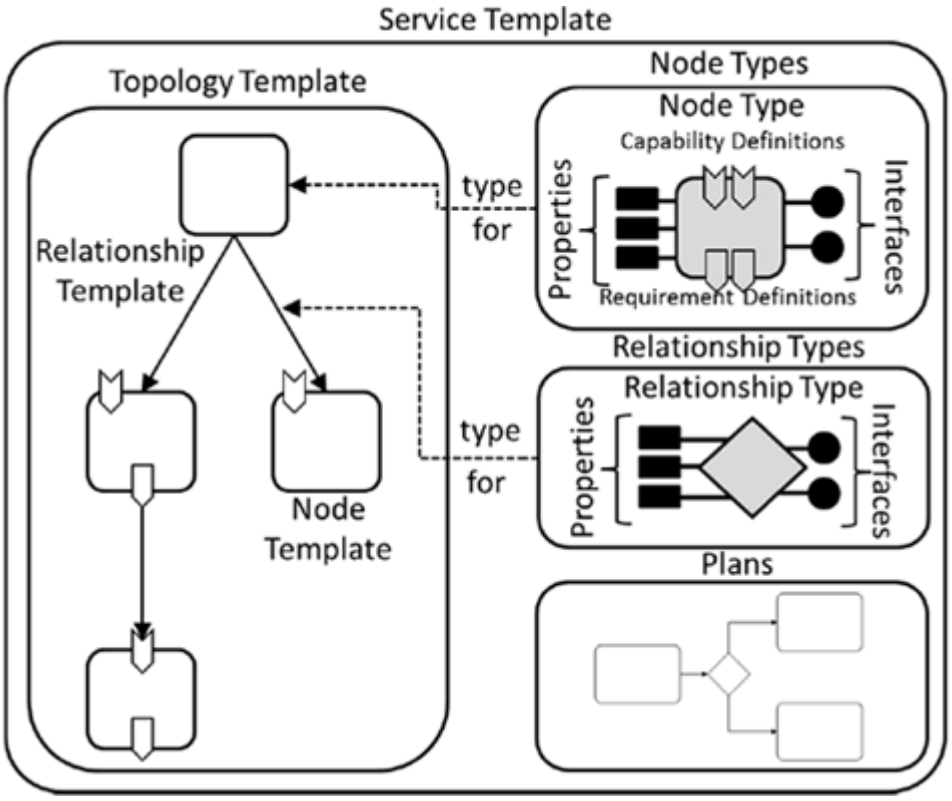


Figure 2.1: TOSCA Service template [25].

2.1 Use cases of TOSCA

In the previous paragraph we have seen which is the main use case of TOSCA: to provide a standard support in specifying topological and management aspects of cloud service applications. Despite this, in the TOSCA specification [25] several other supported use cases are proposed.

Service as marketable entities

According to the authors, the service template standardization will cause the creation and spread of a market for hosted IT services.

Having a standard for topology template definition enables interoperable specification of services' structure. Especially, that standard will let

cloud service development experts define service topology models. Those models could then be published in repositories of one or more service providers. Each provider would map the specified service topology to its available concrete infrastructure in order to support concrete instances of the service and adapt the management plans accordingly.

Furthermore, making a concrete instance of a topology template can be done by running a corresponding plan (also known as *build plan*). So, the service developer who creates the service template could provide this plan too. The build plan can be adapted to the concrete environment of a particular service provider. This is not only the case of build plans: other management plans of a service could then be specified as part of a service template (and adapted to the providers' infrastructure).

Thus, not only the structure of a service can be defined in an interoperable manner, but also its management plans. Such a configuration will let consumers easily search, select and use predefined IT services (hosted on cloud service providers).

Portability of service templates

TOSCA service templates standardization will support the portability of IT service definitions. Observe that, as specified by the TOSCA authors, portability denotes the ability of one cloud provider to understand the structure and behaviour of a service template created by another party (e.g., another cloud provider, enterprise IT department, service developer).

Furthermore, if a service template is portable, this does not mean that its encompassed components are portable too. Portability of a service only implies that its definition is comprehensible in an interoperable manner (i.e., the topology model and corresponding plans are understood by TOSCA-compliant providers). Individual components portability has to be ensured (if needed) via other mechanisms.

Service composition

Another use case specified by TOSCA authors is the composition of service templates. Since a service template provides an abstraction that does not make assumption about the hosting environment, the deployed service could be hosted on more than one cloud provider. This enables an important feature: multi-cloud service applications deployment (e.g., large organization could use automation products from different suppliers for different data centers).

2.2 TOSCA syntax

In this section we will show how cloud services can be defined using TOSCA. To do it, we will follow the same notation of the TOSCA specification to define the serialization of resources:

- characters are appended to items to indicate cardinality:
  - "?" (0 or 1);
  - "*" (0 or more);
  - "+" (1 or more);
- vertical bars, "|", denote choice;
- parentheses, "(" and ")", are used to indicate the scope of the operators "?", "*", "+" and "|";
- ellipses (i.e., "...") indicate points of extensibility.

Definitions

All elements needed to define a cloud service are provided in the TOSCA *Definitions* element (Listing 2.1). This element is the root of a TOSCA

XML document. It has a set of properties; the most important ones will be discussed next.

```
<Definitions id="xs:ID"
             name="xs:string"?
             targetNamespace="xs:anyURI">

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

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

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

    (
      <ServiceTemplate> ... </ServiceTemplate>
    |
      <NodeType> ... </NodeType>
    |
      <NodeTypeImplementation> ...
      </NodeTypeImplementation>
    |
      <RelationshipType> ... </RelationshipType>
    |
      <RelationshipTypeImplementation> ...
      </RelationshipTypeImplementation>
    |
      <RequirementType> ... </RequirementType>
    |
      <CapabilityType> ... </CapabilityType>
    |
      <ArtifactType> ... </ArtifactType>
    |
```

```
37        <ArtifactTemplate> ... </ArtifactTemplate>
38      |
39        <PolicyType> ... </PolicyType>
40      |
41        <PolicyTemplate> ... </PolicyTemplate>
42      ) +
43
44  </Definitions>
```

Listing 2.1: TOSCA *Definitions* element high level syntax

Each *Definitions* element has a unique *id* regarding its namespace and (possibly) a descriptive, human-readable *name*. In addition, the *targetNamespace* attribute declares the namespace of the *Definitions*. This is an important feature because it lets other *Definitions* elements reference this one.

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

With the optional *Types* element the application developer can specify additional XML definitions to use throughout the *Definitions* document (e.g., as attributes in other elements). In this way, the developer does not need to define them in separate documents and import them via *Import* elements. The types are XML Schema elements by default but they could also be of arbitrary types.

The explanation of the other elements reported in Listing 2.1 is given in the following sections.

Service templates

The *ServiceTemplate* element (Listing 2.2) specifies each topological and management aspect of a cloud application by means of *TopologyTemplate* elements and *Plans*, respectively.

```
<ServiceTemplate id="xs:ID"
                 name="xs:string"?
                 targetNamespace="xs:anyURI"
                 substitutableNodeType="xs:QName"?>

  <BoundaryDefinitions>
    <Properties> ... </Properties> ?

    <PropertyConstraints> ... </PropertyConstraints> ?

    <Requirements> ... </Requirements> ?

    <Capabilities> ... </Capabilities> ?

    <Policies> ... </Policies> ?

    <Interfaces> ... </Interfaces> ?
  </BoundaryDefinitions> ?

  <TopologyTemplate>
  (
    <NodeTemplate id="xs:ID" name="xs:string"?
                  type="xs:QName"
                  minInstances="xs:integer"?
                  maxInstances="xs:integer|xs:string"?>
      <Properties> ... </Properties> ?

      <PropertyConstraints> ... </PropertyConstraints> ?

      <Requirements> ... </Requirements> ?

      <Capabilities> ... </Capabilities> ?

      <Policies> ... </Policies> ?

      <DeploymentArtifacts> ... </DeploymentArtifacts> ?
    </NodeTemplate>
    |
```

```
39      <RelationshipTemplate id="xs:ID" name="xs:string"?
40                            type="xs:QName">
41        <Properties> ... </Properties> ?
42
43        <PropertyConstraints> ... </PropertyConstraints> ?
44
45        <SourceElement ref="xs:IDREF"/>
46
47        <TargetElement ref="xs:IDREF"/>
48
49        <RelationshipConstraints> ...
50        </RelationshipConstraints> ?
51      </RelationshipTemplate>
52      ) +
53    </TopologyTemplate>
54
55    <Plans> ... </Plans> ?
56  </ServiceTemplate>
```

Listing 2.2: TOSCA *ServiceTemplate* element high level syntax

As for the *Definitions* element, each *ServiceTemplate* element requires an unique *id* in its own *targetNamespace*. An important additional attribute is *substitutableNodeType*. This attribute specifies the *NodeType* that can be substituted by this *ServiceTemplate*. So, if another *ServiceTemplate* contains a *NodeTemplate* of the specified *NodeType* (or any *NodeType* which is derived from the specified one), then such *NodeTemplate* can be substituted by an instance of the *ServiceTemplate* under definition.

Another way a node type can be substituted by a service template is by matching what it exposes (*Properties*, *Requirements*, *Capabilities*, *Policies* and *Interfaces*) with what a *ServiceTemplate* exposes. The latter comes with the *BoundaryDefinitions* element.

When the cloud application developer has specified all the global aspects, she could proceed in defining the topology and management concerns.

Topology template 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[1]. Each edge expresses the semantics of the relationship between nodes.

Plans The *Plans* element contains *Plan* elements specifying how to manage the *ServiceTemplate* under definition during its life cycle (e.g., how to deploy, start and stop it).

Node types

With the *NodeType* element (Listing 2.3) it is possible to specify a reusable entity that defines the type of one or more node templates in a *ServiceTemplate* element.

```
1  <NodeType name="xs:NCName"
2            targetNamespace="xs:anyURI"?
3            abstract="yes|no"?
4            final="yes|no"?>
5    <DerivedFrom typeRef="xs:QName"/> ?
6
7    <PropertiesDefinition element="xs:QName"?
8                          type="xs:QName"?/> ?
9
10   <RequirementDefinitions> ... </RequirementDefinitions> ?
11
12   <CapabilityDefinitions> ... </CapabilityDefinitions> ?
13
14   <InstanceStates> ... </InstanceStates> ?
15
16   <Interfaces> ... </Interfaces> ?
17 </NodeType>
```

Listing 2.3: TOSCA *NodeType* element high level syntax.

[1]Both *NodeTemplate* elements and *RelationshipTemplate* elements are typed by referring *NodeType* elements and *RelationshipType* elements, respectively.

Each *NodeType* requires a unique *name* to be identified in its own *targetNamespace*. The *NodeType* under definition could be also specified as *abstract* or *final* (if needed). The former means that no instances can be created from *NodeTemplate* elements that use this *NodeType* as their type. The latter says that other *NodeType* elements must not be derived from the under definition one.

With the optional *DerivedFrom* element, the node type developer can implement inheritance. Conflicting definitions are resolved by the rule that local new definition always override derived definitions.

The optional *RequirementDefinitions* and *CapabilityDefinitions* elements are used to specify which properties and capabilities the *NodeType* under consideration exposes.

The *InstanceStates* element is used to model the life cycle of an instance of this *NodeType*. Indeed, this element specifies the set of states an instance of this *NodeType* can occupy.

Finally, with the optional element *Interfaces* the developer could define the operations which can be performed on (instances of) this *NodeType*. Such operations definition is given in the form of nested *Interface* elements (each of which is characterized by a *name* and some *Operation* elements).

Once the *NodeType* has been defined, it should be linked with the executables by which is implemented. *NodeTypeImplementation* elements provide a collection of implementation artifacts (executables implementing the interface operations) and deployment artifacts (executables needed to materialize instances of *NodeTemplate* elements referring the *NodeType* under consideration)[2].

[2]The respective executables are defined as separate *ArtifactTemplate* elements and are referenced from the implementation artifacts and deployment artifacts of a *NodeTypeImplementation*.

Relationship types

As mentioned earlier, pairs of *NodeTemplate* elements are connected by *RelationshipTemplate* elements. Each of these *RelationshipTemplate* elements is typed by a *RelationshipType* element (Listing 2.4).

```
<RelationshipType name="xs:NCName"
                  targetNamespace="xs:anyURI"?
                  abstract="yes|no"?
                  final="yes|no"?>
  <DerivedFrom typeRef="xs:QName"/> ?

  <PropertiesDefinition element="xs:QName"?
                        type="xs:QName"?/> ?

  <InstanceStates> ... </InstanceStates> ?

  <SourceInterfaces> ... </SourceInterfaces> ?

  <TargetInterfaces> ... </TargetInterfaces> ?

  <ValidSource> ... </ValidSource> ?

  <ValidTarget> ... </ValidTarget> ?
</RelationshipType>
```

Listing 2.4: TOSCA *RelationshipType* element high level syntax.

Every *RelationshipType* requires a unique *name* to be identified in its own *targetNamespace*. If needed, the *RelationshipType* under definition could be specified as *abstract* or *final*. These two attributes have the same meaning of those appearing in *NodeType* elements. Furthermore, also the meaning (and usage) of *DerivedFrom*, *PropertiesDefinition* and *InstanceStates* elements is the same of those contained in *NodeType* elements.

The optional *SourceInterfaces* and *TargetInterfaces* elements contain definitions of manageability interfaces of a relationship of this *Relation-*

shipType (in order to actually establish the relationship between the source and the target in the deployed service). Those interface definitions are contained in nested *Interface* elements, whose content is the same as for *NodeType* interfaces.

With the optional *ValidSource* and *ValidTarget* elements, the service developer can specify the type of object that is allowed as a valid origin or target for relationships of the *RelationshipType* under definition.

Finally, the *RelationshipType* should be linked with the objects by which its interfaces are implemented. This comes with a separated *RelationshipTypeImplementation* element. Indeed, such an element provides the collection of executables implementing the interface operations (also known as *implementation artifacts*)[3].

Requirements and capabilities

Each *NodeType* element can declare to expose some requirements and capabilities. As shown in Figure 2.2, they can be expressed instantiating *RequirementType* and *CapabilityType* elements (Listing 2.5) via *RequirementDefinition* and *CapabilityDefinition* elements.

```
1  <RequirementType name="xs:NCName"
2                   targetNamespace="xs:anyURI"?
3                   abstract="yes|no"?
4                   final="yes|no"?
5                   requiredCapabilityType="xs:QName"?>
6    <DerivedFrom typeRef="xs:QName"/> ?
7
8    <PropertiesDefinition element="xs:QName"?
9                          type="xs:QName"?/> ?
10 </RequirementType>
11
12 <CapabilityType name="xs:NCName"
```

[3]The particular executables are defined as separate *ArtifactTemplate* elements and referenced from the implementation artifacts of a *RelationshipTypeImplementation*.

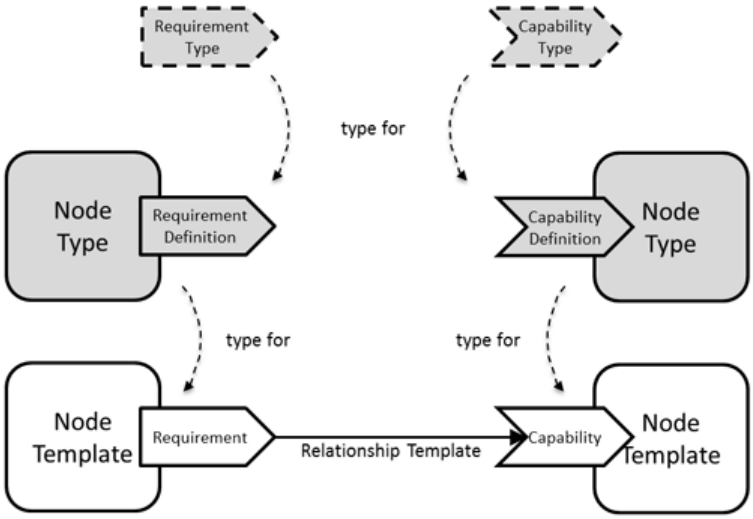


Figure 2.2: Requirement and capabilities [25]

```
13                  targetNamespace="xs:anyURI"?
14                  abstract="yes|no"?
15                  final="yes|no"?>
16   <DerivedFrom typeRef="xs:QName"/> ?
17
18   <PropertiesDefinition element="xs:QName"?
19                         type="xs:QName"?/> ?
20 </CapabilityType>
```

Listing 2.5: TOSCA *RequirementType* element and *CapabilityType* element high level syntax.

The meaning of all the *RequirementType* and *CapabilityType* elements common properties is the same of previous elements. The only addition is the optional *requiredCapabilityType* attribute. It could be used to specify which *CapabilityType* elements[4] satisfy the requirement expressed by the *RequirementType* element under definition.

[4]Since TOSCA supports inheritance, both the specified capability type and those types derived from it satisfy the considered requirement.

Policies

Non-functional behaviour or QoS (Quality of Service) are defined in TOSCA by means of policies. A *Policy* can express such diverse things like monitoring behaviour, payment conditions, scalability, or continuous availability, for example.

As reported in Listing 2.1, a *NodeTemplate* can be associated with a set of *Policies* collectively expressing the non-functional behaviour or QoS that each instance of the *NodeTemplate* will expose. Each *Policy* specifies the actual properties of the non-functional behaviour. These properties are defined by means of a *PolicyType* (Listing 2.6).

```
1  <PolicyType name="xs:NCName"
2              policyLanguage="xs:anyURI"?
3              abstract="yes|no"? final="yes|no"?
4              targetNamespace="xs:anyURI"?>
5    <DerivedFrom typeRef="xs:QName"/> ?
6
7    <PropertiesDefinition element="xs:QName"?
8                          type="xs:QName"?/>
9
10   <AppliesTo>
11     <NodeTypeReference typeRef="xs:QName"/> +
12   </AppliesTo> ?
13
14   policy type specific content ?
15 </PolicyType>
```

Listing 2.6: TOSCA *PolicyType* element high level syntax

Observe that a *PolicyType* could specify the non-functional behaviour it represents via both *PropertiesDefinition* and *policy type specific content*[5]. Furthermore, (via the *AppliesTo* element) a policy type can declare the set of *NodeType* elements it specifies non-functional behaviour for. Note

[5]The latter uses an arbitrary language which can be specified in the optional *PolicyLanguage* attribute.

that being "applicable to" does not enforce implementation (e.g., in case a *PolicyType* expressing high availability is associated with a *WebserverNodeType*, an instance of the *WebserverNodeType* is not necessarily high available). Whether or not an instance of a *NodeType* to which a *PolicyType* is applicable will show the specified non-functional behaviour, is determined by a *NodeTemplate* of the corresponding *NodeType*.

Once the general properties have been defined in a *PolicyType*, their actual values are provided by one or more *PolicyTemplate* elements (Listing 2.7).

```
1 <PolicyTemplate id="xs:ID"
2                 name="xs:string"?
3                 type="xs:QName">
4   <Properties> ... </Properties>
5
6   <PropertyCostraints> ... </PropertyCostraints>
7
8   policy type specific content ?
9 </PolicyTemplate>
```

Listing 2.7: TOSCA *PolicyTemplate* element high level syntax.

A *PolicyTemplate* defines the invariant properties of the non-functional behaviour to be represented. Its variant properties are setted in a *Policy* element of a *NodeTemplate*. This is because those properties result from the actual usage of a *PolicyTemplate* in the *NodeTemplate* under consideration (e.g., a *PolicyTemplate* for italian customers yearly payments will set the *paymentPeriod* property to “yearly” and the *currency* property to “EUR”, leaving the *amount* property open; the *amount* property will be set when the corresponding *PolicyTemplate* is used for a *Policy* within a *NodeTemplate*).

2.3 A TOSCA example

Suppose that a cloud application developer wants to build the weather forecast cloud application in Figure 2.3. Her work starts with the creation

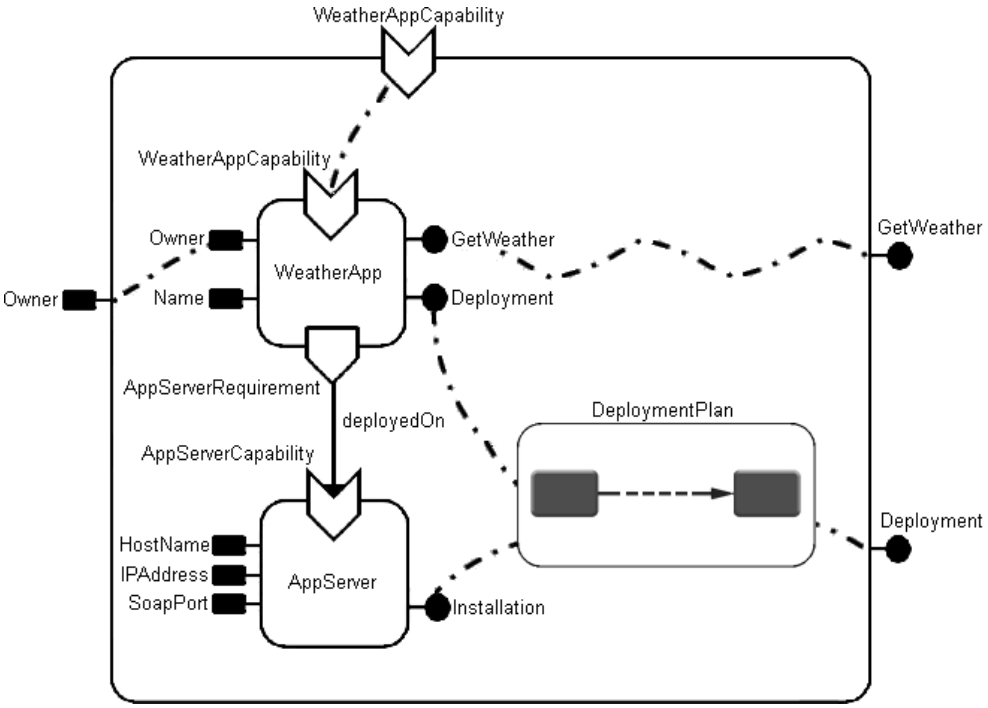


Figure 2.3: TOSCA cloud application example.

of a new *Definitions* document (Listing 2.8) in which to put all required elements. To enhance readability we are going to show each needed definition in a separate section.

```
1 <Definitions name = "SampleSTDefinitions"
2              targetNamespace =
3                    "http://www.example.com/weatherSample">
4   ...
5 </Definitions>
```

Listing 2.8: TOSCA example *Definitions* document.

Requirement and capability types

Before writing the *NodeType* and *RelationshipType* elements needed by the desired application, the developer must ensure she has all the necessary requirement and capability types. In other words, she has to write the *AppServerRequirement*, *AppServerCapability* and *WeatherAppCapability* definitions (Listing 2.9) and to include them in the *Definitions* element.

```
<RequirementType
        name = "AppServerRequirement"
        targetNamespace =
              "http://www.example.com/weatherSample"
/>
<CapabilityType
        name = "AppServerCapability"
        targetNamespace =
              "http://www.example.com/weatherSample"
/>
<CapabilityType
        name = "WeatherAppCapability"
        targetNamespace =
              "http://www.example.com/weatherSample"
/>
```

Listing 2.9: TOSCA example requirement and capability types definitions.

Please note that the exposed requirement and capabilities definitions are only composed by their name. This is because they will be used only in establishing the meaning of desired relationships.

Node types

Once requirement and capability types are defined, the application developer needs only one other ingredient to proceed in *NodeType* definitions:

the properties structures definition (Listing 2.10).

```
<Types>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
             elementFormDefault="qualified"
             attributeFormDefault="unqualified">
    <xs:element name="ApplicationProperties">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="Owner"
                      type="xs:string"/>
          <xs:element name="Name"
                      type="xs:string"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    <xs:element name="AppServerProperties">
      <xs:complexType>
        <xs:sequence>
          <xs:element name="HostName"
                      type="xs:string"/>
          <xs:element name="IPAddress"
                      type="xs:string"/>
          <xs:element name="SoapPort"
                      type="xs:positiveInteger"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
  </xs:schema>
</Types>
```

Listing 2.10: Properties structure definitions required in our TOSCA example.

As the *Types* element is included in the *Definitions* document, she can proceed defining the required *NodeType* elements (Listing 2.11).

```
<NodeType name = "WheaterApplicationType"
          targetNamespace =
                "www.example.com/weatherSample">
```

```
4
5   <PropertiesDefinition element = "ApplicationProperties"/>
6
7   <RequirementDefinitions>
8     <RequirementDefinition name = "ServerRequired"
9                            type = "AppServerRequirement"/>
10  </RequirementDefinitions>
11
12  <CapabilityDefinitions>
13    <CapabilityDefinition name = "WeatherApplication"
14                          type = "WeatherAppCapability"/>
15  </CapabilityDefinitions>
16
17  <InstanceStates>
18    <InstanceState state = "www.example.com/started"/>
19    <InstanceState state = "www.example.com/stopped"/>
20  </InstanceStates>
21
22  <Interfaces>
23    <Interface name = "Deployment">
24      <Operation name = "DeployApplication">
25        <InputParameters>
26          <InputParameter name = "ServerIPAddress"
27                          type = "xs:string"/>
28        </InputParameters>
29      </Operation>
30    </Interface>
31    <Interface name = "GetWeather">
32      <Operation name = "GetWeather">
33        <InputParameters>
34          <InputParameter name = "City"
35                          type = "xs:string"/>
36        </InputParameters>
37        <OutputParameters>
38          <OutputParameter name = "Weather"
39                           type = "xs:string"/>
40          <OutputParameter name = "Temperature"
41                           type = "xs:positiveInteger"/>
```

```
42          </OutputParameters>
43        </Operation>
44      </Interface>
45
46 </NodeType>
47
48 <NodeType name = "AppServerType"
49           targetNamespace =
50                 "www.example.com/weatherSample">
51
52   <PropertiesDefinition element = "AppServerProperties"/>
53
54   <CapabilityDefinitions>
55     <CapabilityDefinition name = "ServerCapability"
56                           type = "AppServerCapability"/>
57   </CapabilityDefinitions>
58
59   <Interfaces>
60     <Interface name = "Installation">
61       <Operation name = "AcquireNetworkAddress">
62       <Operation name = "DeployApplicationServer">
63     </Interface>
64   </Interfaces>
65 </NodeType>
```

Listing 2.11: TOSCA example *NodeType* definitions.

Relationship types

Looking at Figure 2.3, we observe that the desired service topology contains a relationship between the *WeatherApp*'s requirement and the *AppServer*'s capability. So, the desired relationship's type must be defined by the application developer (Listing 2.12).

```
1 <RelationsipType name = "deployedOnType">
2   <ValidSource typeRef = "sample:AppServerRequirement"/>
3
4   <TargetSource typeRef = "sample:AppServerCapability"/>
```

```
</RelationshipType>
```

Listing 2.12: TOSCA example *RelationshipType* definition.

Topology template

In the previous sections we have shown the type definitions made by the weather forecast application developer. Now, she can proceed in instantiating those types in order to build up the desired service topological aspects (Listing 2.13).

```
<ServiceTemplate id = "SampleST">
  <TopologyTemplate id = "SampleTopology">

    <NodeTemplate id = "SampleWeatherApp"
                  name = "WeatherApp"
                  nodeType =
                      "sample:WheaterApplicationType">
      <Properties>
        <ApplicationProperties>
          <Owner>Minnie<Owner>
          <Name>Wheater Forecast Application<Name>
        </ApplicationProperties>
      </Properties>

      <Requirements>
        <Requirement id = "SampleServerRequirement"
                     name = "ServerRequired"
                     type =
                         "sample:AppServerRequirement"/>
      </Requirements>

      <Capabilities>
        <Capability id = "SampleWeatherAppCapability"
                    name = "WeatherApplication"
                    type =
                        "sample:WeatherAppCapability"/>
      </Capabilities>
```

```
28      </NodeTemplate>
29
30      <NodeTemplate id = "SampleAppServer"
31                    name = "AppServer"
32                    nodeType = "sample:AppServerType">
33
34        <Capabilities>
35          <Capability id = "SampleServerCapability"
36                      name = "ServerCapability"
37                      type = "sample:AppServerCapability"/>
38        </Capabilities>
39      </NodeTemplate>
40
41      <RelationshipTemplate
42             id = "SampleDeployedOn"
43             relationshipType = "deployedOnType">
44        <SourceElement id = "SampleWeatherApp"/>
45        <TargetElement id = "SampleAppServer"/>
46      </RelationshipTemplate>
47
48    </TopologyTemplate>
49    ...
50  </ServiceTemplate>
```

Listing 2.13: TOSCA example *TopologyTemplate* definition.

Plans

Once the service topology has been defined, the application developer must define all desired management aspects. A possible solution to the exposed issue is to use URI references to already existing plans (Listing 2.14).

```
1  <ServiceTemplate id = "SampleST">
2    ...
3    <Plans>
4      <Plan id = "SampleDeploymentPlan"
5            name = "DeploymentPlan"
```

```
6           planType =
7               "www.example.com/Plan/Types/BuildPlan"
8           planLanguage =
9               "www.example.com/Plan/Languages/BPEL">
10        <PlanModelReference reference = "prj:DeployApp"
11      </Plan>
12    </Plans>
13    ...
14 </ServiceTemplate>
```

Listing 2.14: TOSCA example *Plans* definition.

Service template and boundaries

Last but not least, the application developer must ensure that her service only exposes what she wants. This is done via the *BoundaryDefinitions* element (Listing 2.15).

```
1  <ServiceTemplate id = "SampleST">
2    ...
3    <BoundaryDefinitions>
4      <Properties>
5        <Owner/>
6        <PropertyMappings>
7          <PropertyMapping serviceTemplatePropertyRef =
8                                          "$doc/Owner"
9                           targetObjectRef =
10                                         "SampleWeatherApp"
11                          targetObjectPropertyRef =
12                                         "$doc//Owner"/>
13       </PropertyMappings>
14     </Properties>
15
16     <Capabilities>
17       <Capability name = "WeatherApplicationCapability"
18                   ref = "SampleWeatherAppCapability" />
19     </Capabilities>
20
```

```
21      <Interfaces>
22        <Interface name = "GetWeather">
23          <Operation name = "GetWeather">
24            <NodeOperation nodeRef = "SampleWeatherApp"
25                           interfaceName = "GetWeather"
26                           operationName = "GetWeather"/>
27          </Operation>
28        </Interface>
29        <Interface name = "Deployment">
30          <Operation name = "DeployApplication">
31            <Plan planRef = "SampleDeploymentPlan"/>
32          </Operation>
33        </Interface>
34      </Interfaces>
35
36    </BoundaryDefinitions>
37  </ServiceTemplate>
```

Listing 2.15: TOSCA example boundary definitions.

Chapter 3

NodeType black-box matching (and adaptation)

The objective of this chapter is to give a notion of black-box matching between *NodeType* and *ServiceTemplate* elements. Once this notion is available, it can be used as a first step in understanding whether a TOSCA service component could be replaced with an available cloud service.

To provide such a compatibility notion we start by defining an *exact matching* between the two TOSCA elements under consideration (Section 3.1). Then, observing that there is no need to exactly match the elements, we define the notion of *plug-in matching* (Section 3.2). Finally, using ontologies to ignore non-relevant syntactic differences, we derive the *flexible matching* notion (Section 3.3).

We also describe how a *ServiceTemplate* that *plug-in* or *flexibly* matches the desired *NodeType* can be suitably adapted so as to exactly match it.

3.1 Exact matching

We consider the problem of matching a *NodeType* N with a *ServiceTemplate* ST. The TOSCA Primer [26] gives an informal notion of matching between these two elements. Referring to Figure 3.1, the Primer authors

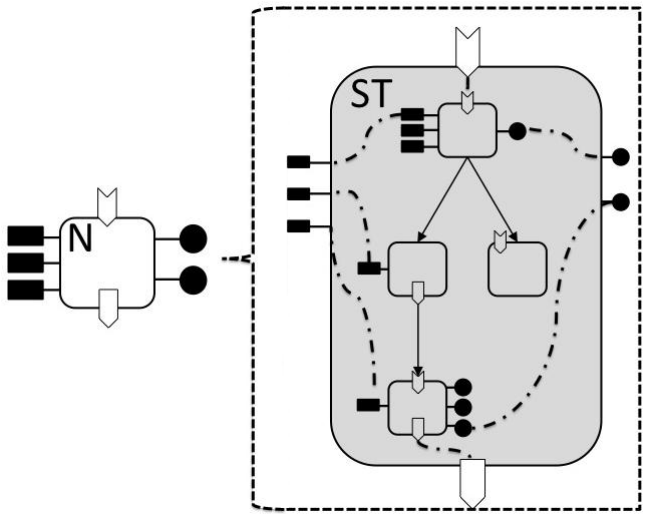


Figure 3.1: *ServiceTemplate* substituting a *NodeType* [26].

say:

> «*Service template ST may substitute node type N because the boundary of ST matches all defining elements of N: all properties, operations, requirements and capabilities of ST match exactly those of N.*»

No additional matching information is given. So, we want to formalize this notion.

3.1.1 Definition of *exact matching*

What we want to do is to define an operator "$\equiv$" which takes a pair $\langle$*NodeType*, *ServiceTemplate*$\rangle$ and returns a truth value (which represents

whether the two elements exactly match or not).

Definition 3.1. A *NodeType* N exactly matches a *ServiceTemplate* ST ($N \equiv ST$) if and only if:

$$N.RequirementDefinitions \equiv_R ST.Requirements \wedge$$

$$N.CapabilityDefinitions \equiv_C ST.Capabilities \wedge$$

$$PolicyType \text{ applicable to } N \equiv_{PO} ST.Policies \wedge$$

$$N.PropertiesDefinition \equiv_{PR} ST.Properties \wedge$$

$$N.Interfaces \equiv_I ST.Interfaces$$

$\circ$

To understand what Definition 3.1 means, we need to analyse each of the given conditions.

Exact matching of requirements

Looking at the syntax of the elements interested in this kind of matching (Listing 3.1), we can define the required condition.

```
1  <NodeType ...>
2    ...
3    <RequirementDefinitions>
4      <RequirementDefintion name = ...
5               requirementType = ...
6               lowerBound = ...
7               upperBound = ...> +
8    </RequirementDefinitions> ?
9    ...
10 </NodeType>
11
12 <ServiceTemplate>
13   ...
```

```
14   <BoundaryDefinitions>
15     ...
16     <Requirements>
17       <Requirement name = ... ref = ...> +
18     </Requirements> ?
19     ...
20   </BoundaryDefinitions> ?
21   <TopologyTemplate>
22     <NodeTemplate ...>
23       <Requirements>
24         <Requirement id = ...
25                name = ...
26                type = ...> +
27       </Requirements> ?
28     </NodeTemplate ...> ?
29   </TopologyTemplate>
30   ...
31 </ServiceTemplate>
```

Listing 3.1: High level syntax of TOSCA *NodeType* and *ServiceTemplate* elements interested in requirements matching.

The Topology and Orchestration Specification for Cloud Applications [25] says that:

> «*The name and type of the Requirement MUST match the name and type of a RequirementDefinition in the NodeType specified in the type attribute of the NodeTemplate.*»

Extending this concept to our problem, we can state that each *RequirementDefinition* in the *NodeType* must have the same name and type of exactly one of the *ServiceTemplate*'s *Requirements* (referred by the *ref* attribute in the *BoundaryDefinitions* element).

Furthermore, looking at *NodeType* specification [25], the authors says that the *NodeType* under consideration exposes also the requirements (if not overridden) of the *NodeType* it is derived from. This means that its

requirements consist of the (set theoretic) union of the requirements it defines and the requirements it inherits from the parent *NodeType*.

The above reasoning let us define the condition needed in Definition 3.1.

Definition 3.2. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.RequirementDefinitions \equiv_R ST.Requirements$$

if and only if[1]:

1. $\forall$ *RequirementDefinition* $x \in N.RequirementDefinitions$
   $\exists!$ *Requirement* $y \in ST.Requirements$:
   $x.name = y.name \wedge x.requirementType = y.ref.type$

2. $\forall$ *Requirement* $y \in ST.Requirements$
   $\exists!$ *RequirementDefinition* $x \in N.RequirementDefinitions$:
   $x.name = y.name \wedge x.requirementType = y.ref.type$

$\circ$

Exact matching of capabilities

The high similarity between the capabilities syntax and the requirements one leads to the following definition.

Definition 3.3. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.CapabilityDefinitions \equiv_C ST.Capabilities$$

[1]Using both the conditions we ensure the one-to-one correspondence between the requirements of the two elements.

if and only if:

1. $\forall$ *CapabilityDefinition* $x \in N.CapabilityDefinitions$
   $\exists!$ *Capability* $y \in ST.Capabilities$:
   $x.name = y.name \wedge x.capabilityType = y.ref.type$

2. $\forall$ *Capability* $y \in ST.Capabilities$
   $\exists!$ *CapabilityDefinition* $x \in N.CapabilityDefinitions$:
   $x.name = y.name \wedge x.capabilityType = y.ref.type$

$\circ$

Policies compatibility

As before, we need to look at the syntax of interested elements to define the policies matching condition.

```
1  <PolicyType name = ...
2        ...>
3    ...
4    <AppliesTo>
5      <NodeTypeReference typeRef = .../> +
6    </AppliesTo> ?
7    ...
8  </PolicyType>
9
10 <PolicyTemplate id = ...
11       name = ...
12       type = ...>
13   ...
14 </PolicyTemplate>
15
16 <ServiceTemplate>
17   ...
18   <BoundaryDefinitions>
19     ...
20     <Policies>
```

```
21        <Policy name = ...
22            policyType = ...
23            policyRef = ...>
24        ...
25        </Policy> +
26      </Policies> ?
27      ...
28    </BoundaryDefinitions> ?
29    ...
30 </ServiceTemplate>
```

Listing 3.2: High level syntax of TOSCA elements interested in policies matching.

Looking at Listing 3.2, we immediately observe the lack of *NodeType* element. In this case we have to consider *PolicyType*, *PolicyTemplate* and *ServiceTemplate* elements. This is because a *NodeType* cannot expose any kind of policy, but a policy can specify which set of *NodeType* elements it is applicable to.

Assumption 3.1. If a *PolicyType* P_{TY} does not contain any *AppliesTo* element, then P_{TY} will be applicable to any *NodeType*.

The above consideration suggests that if we want to substitute a *NodeType* N with a *ServiceTemplate* ST, then the latter must expose policies applicable to the former. So, let us define the infix operator $\succ$ to indicate that the *PolicyType* of the considered policy (on the left of $\succ$) is applicable to the *NodeType* (on the right of $\succ$).

Definition 3.4. The infix operator

$$\succ: (\{PolicyType\} \cup \{PolicyTemplate\}) \times \{NodeType\} \rightarrow \{\texttt{true},\texttt{false}\}$$

is defined as follows:

- let P_{TY} be a *PolicyType* and N a *NodeType*. $P_{TY} \succ N$ is `true`

if and only if the $P_{TY}.AppliesTo$ element (is empty or) contains a *NodeTypeReference* to N or to a *NodeType* from which N is derived;

- let P_{TMP} be a *PolicyTemplate* and N a *NodeType*. $P_{TMP} \succ N$ is `true` if and only if $P_{TMP}.type \succ N$ is `true`.

∘

Now we can define the desired condition.

Definition 3.5. Let N be a *NodeType* and ST a *ServiceTemplate*. The set of *PolicyTypes* applicable to N exactly matches ($\equiv_{PO}$) $ST.Policies$ if and only if:

$$\forall ST.Policies.Policy\ x, (x.policyType \succ N \vee x.policyRef \succ N)$$

∘

Exact matching of properties

Understanding how to match *NodeType.PropertiesDefinition* and *ServiceTemplate* properties requires to look at their syntax (Listing 3.3).

```
1  <NodeType ...>
2    ...
3    <PropertiesDefinition
4      element = ... ?
5      type = ... ?
6    />
7    ...
8  </NodeType>
9
10 <ServiceTemplate>
11   ...
12   <BoundaryDefinitions>
```

```
13      ...
14      <Properties>
15        XML fragment
16        <PropertyMappings>
17          <PropertyMapping
18            serviceTemplatePropertyRef = ...
19            targetObjectRef = ...
20            targetPropertyRef = ...
21          /> +
22        </PropertyMappings> ?
23      </Properties> ?
24      <PropertyConstraints>
25        ...
26      </PropertyConstraints> ?
27      ...
28    </BoundaryDefinitions> ?
29    <TopologyTemplate>
30      ...
31      <NodeTemplate ...>
32        ...
33        <Properties> ... </Properties>
34        ...
35      </NodeTemplate>
36      ...
37      <RelationshipTemplate ... >
38        ...
39        <Properties> ... </Properties>
40        ...
41      </RelationshipTemplate>
42      ...
43    </TopologyTemplate>
44    ...
45  </ServiceTemplate>
```

Listing 3.3: High level syntax of TOSCA *NodeType* and *ServiceTemplate* elements interested in properties matching.

Observe that there is a different properties definition between *NodeType* and *ServiceTemplate* elements:

- the former specifies (via one of the *element* and *type* attributes) the XML schema of the *NodeType* observable properties;
- the latter specifies *ServiceTemplate* actual property values (and constraints) using a XML fragment and mapping the exposed properties to the ones of nested elements (using the *serviceTemplatePropertyRef* attribute - to refer the property defined in the XML fragment - and *targetObjectRef* and *targetObjectPropertyRef* attributes - to identify the property of the nested element).

This means that, because of lack of values (and constraints) in *NodeType* elements, we can only match property types.

Furthermore, in the TOSCA specification [25], the authors assume that the XML type representing the *NodeType* properties extends the XML type of the properties of the *NodeType* referenced in the *DerivedFrom* element. This implies that we do not have to worry about what the *NodeType* element inherits from its parent.

The above considerations let us define the desired condition:

Definition 3.6. Let N be a *NodeType* and ST a *ServiceTemplate*. We say that

$$N.PropertiesDefinition \equiv_{PR} ST.Properties$$

if and only if the XML type of $ST.Properties$ is the same as the one defined with $N.PropertiesDefinition$

∘

Exact matching of interfaces

Finally, we want to analyze *Interfaces* matching. So, as done before, we need to take a look to the elements interested in this matching (Listing 3.4).

```
<NodeType ...>
  ...
  <DerivedFrom typeRef = ... /> ?
  ...
  <Interfaces>
    <Interface name = ... >
      <Operation name = ... >
        <InputParameters>
          <InputParameter
            name = ...
            type = ...
            required = ... ?
          /> +
        </InputParameters> ?
        <OutputParameters>
          <OutputParameter
            name = ...
            type = ...
            required = ... ?
          /> +
        </OutputParameters> ?
      </Operation> +
    </Interface> +
  </Interfaces> ?
  ...
</NodeType>

<ServiceTemplate>
  ...
  <BoundaryDefinitions>
    ...
    <Interfaces>
      <Interface name = ... >
```

```
34          <Operation name = ... >
35          (
36            <NodeOperation
37              nodeRef = ...
38              interfaceName = ...
39              operationName = ...
40            />
41          |
42            <RelationshipOperation
43              relationshipRef = ...
44              interfaceName = ...
45              operationName = ...
46            />
47          |
48            <Plan planRef = ... />
49          )
50          </Operation> +
51        </Interface> +
52      </Interfaces> ?
53      ...
54    </BoundaryDefinitions> ?
55    <TopologyTemplate>
56      ...
57      <NodeTemplate
58        id = ...
59        name = ...
60        type = ...
61        ...
62      >
63        ...
64      </NodeTemplate>
65      ...
66      <RelationshipTemplate
67        id = ...
68        name = ...
69        type = ...
70        ...
71      >
```

```
72        ...
73      </RelationshipTemplate>
74      ...
75    </TopologyTemplate>
76    <Plans>
77      <Plan
78        id = ...
79        name = ...
80        ...
81      >
82        ...
83        <InputParameters>
84          <InputParameter
85            name = ...
86            type = ...
87            required = ... ?
88          /> +
89        </InputParameters> ?
90        <OutputParameters>
91          <OutputParameter
92            name = ...
93            type = ...
94            required = ... ?
95          /> +
96        </OutputParameters> ?
97        ...
98      </Plan> +
99    </Plans> ?
100 </ServiceTemplate>
```

Listing 3.4: High level syntax of TOSCA *NodeType* and *ServiceTemplate* elements interested in interfaces matching.

To obtain the desired exact matching condition, we need to ensure that both the *NodeType* element and the *ServiceTemplate* element expose the same interfaces. In other words, we want that each *NodeType* interface contains the same operations of exactly one of the *ServiceTemplate* interfaces (and vice versa). Formally:

Definition 3.7. Let N be a *NodeType* and ST a *ServiceTemplate*. We say that

$$N.Interfaces \equiv_I ST.Interfaces$$

if and only if:

1. $\forall$ *Interface* $x \in N.Interfaces$ $\exists!$ *Interface* $y \in ST.Interfaces$:
   $x.name = y.name \wedge$
   $\forall$ *Operation* $i \in x$ $\exists!$*Operation* $j \in y$:
   $i \equiv_O j$

2. $\forall$ *Interface* $y \in ST.Interfaces$ $\exists!$ *Interface* $x \in N.Interfaces$:
   $y.name = x.name \wedge$
   $\forall$*Operation* $j \in y$ $\exists!$*Operation* $i \in x$:
   $j \equiv_O i$

$\circ$

The above definition requires to specify how two *Operation* elements can be considered in the $\equiv_O$ relationship.

Definition 3.8. Consider two *Operation* elements O_1 and O_2. We say that

$$O_1 \equiv_O O_2$$

if and only if

1. $O_1.name = O_2.name$

2. $\forall$ *InputParameter* $a \in O_1.InputParameters$
   $\exists!$ *InputParameter* $b \in O_2.InputParameters$:
   $a.name = b.name \wedge a.type = b.type \wedge a.required = b.required$

3. $\forall$ *InputParameter* $b \in O_2.InputParameters$
   $\exists!$ *InputParameter* $a \in O_1.InputParameters$
   $b.name = a.name \wedge b.type = a.type \wedge b.required = a.required$

4. $\forall$ *OutputParameter* $a \in O_1.OutputParameters$
   $\exists!$ *OutputParameter* $b \in O_2.OutputParameters$:
   $a.name = b.name \wedge a.type = b.type \wedge a.required = b.required$

5. $\forall$ *OutputParameter* $b \in O_2.OutputParameters$
   $\exists!$ *OutputParameter* $a \in O_1.OutputParameters$:
   $b.name = a.name \wedge b.type = a.type \wedge b.required = a.required$

$\circ$

3.1.2 Exact matching examples

Suppose that our TOSCA cloud application requires a node whose type is the one in Figure 3.2. What we want to do is to check whether existing

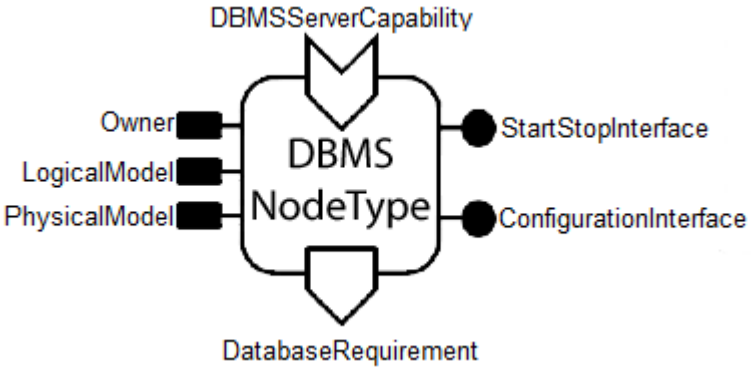


Figure 3.2: TOSCA *DBMSNodeType* example.

ServiceTemplate elements can be used as substitute for *NodeTemplate* elements of the specified *NodeType*. In the following paragraphs we will show two examples of existing *ServiceTemplate* to be matched[2]: the first

[2]For the sake of simplicity, we will consider *ServiceTemplate* elements not exposing any kind of *Policy*.

one will be fully compatible with the desired *NodeType*; the second one will not.

Suppose that the *ServiceTemplate ST* in Figure 3.3 is available and suppose that its *Interface* elements contains the same operations of the ones in the *DBMSNodeType*. It is clear that the following condition is

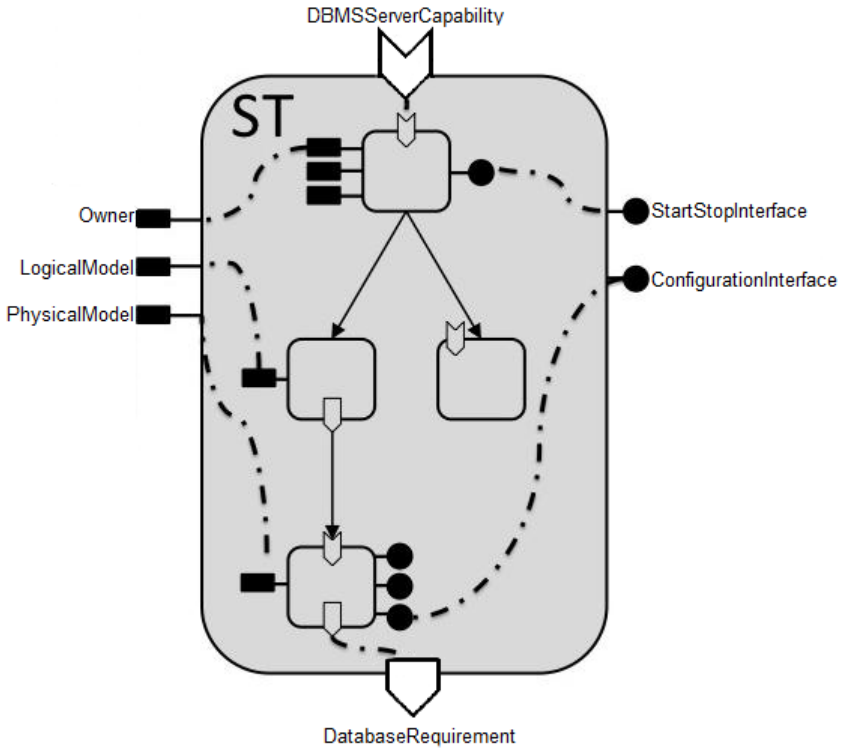


Figure 3.3: TOSCA *ServiceTemplate* exact matching example.

true:

$$ST \equiv DBMSNodeType.$$

Let us now modify the previous *ServiceTemplate ST* by exposing just another property of the nested nodes (Figure 3.4). Because of the introduction of the new *Property* in ST' the exact matching condition becomes `false`. Nevertheless, it is clear that the *ServiceTemplate* under

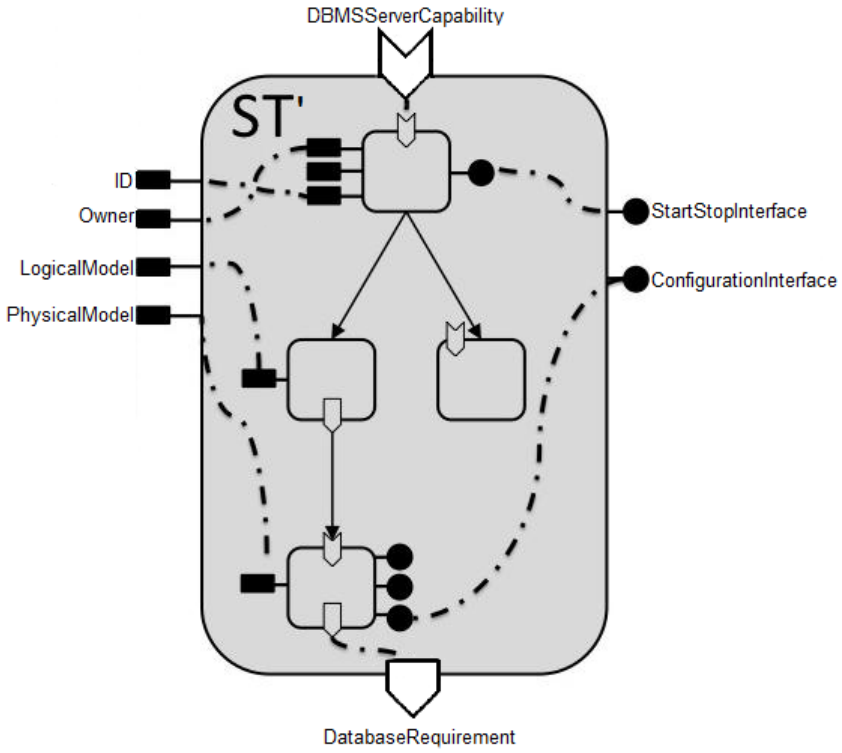


Figure 3.4: TOSCA *ServiceTemplate* exact matching example (modified).

consideration (with some simply adaptation) can be used as a substitute for the *DBMSNodeType*.

The problem stays in the too much strict matching condition given in Definition 3.1. So, we need to be more flexible (by relaxing some of the constraints).

3.2 Plug-in matching

In the previous section we have provided a notion of full compatibility between *NodeType* and *ServiceTemplate* elements. Then, that notion has been applied to some examples. In the last one of these we have observed that, despite the *ServiceTemplate* element under consideration could be simply adapted to be compatible with the *NodeType* we want to match, the Definition 3.1 cannot be used. As previously reported, the problem stays in the too much strict conditions provided by the definition considered.

In this section we will relax some of the Definition 3.1 boundaries in order to obtain a formal definition of *plug-in matching*.

3.2.1 Definition of *plug-in matching*

The objective of this section is to define an operator "$\subseteq$" analogue to the "$\equiv$" introduced in Section 3.1: it takes a pair $\langle$*NodeType*,*ServiceTemplate*$\rangle$ and returns a truth value (which is `true` if the *NodeType* and *ServiceTemplate* are in *plug-in* matching, `false` otherwise).

To better understand how this operator works, we have to clarify what *plug-in* matching means. Consider the *NodeType* N and the *ServiceTemplate* ST. Intuitively speaking, we say that $N \subseteq ST$ if:

- ST's policies are applicable to N,

- N exposes more requirements than ST, and
- N offer less capabilities, properties and operations than ST.

If this is the case, then we can easily adapt ST (without looking inside of it) in order to obtain a *ServiceTemplate* ST' such that $N \equiv ST'$.

Observation 3.1. Please note that if $N \equiv ST$ then there is no need of adaptation on ST to obtain the exact matching condition. This means that

$$N \equiv ST \Longrightarrow N \subseteq ST.$$

Now we have all the fundamentals needed to proceed defining formally what *plug-in* matching means.

Definition 3.9. A *NodeType* N plug-in matches a *ServiceTemplate* ST ($N \subseteq ST$) if and only if:

$$N.RequirementDefinitions \subseteq_R ST.Requirements \wedge$$

$$N.CapabilityDefinitions \subseteq_C ST.Capabilities \wedge$$

$$PolicyType \text{ applicable to } N \equiv_{PO} ST.Policies \wedge$$

$$N.PropertiesDefinition \subseteq_{PR} ST.Properties \wedge$$

$$N.Interfaces \subseteq_I ST.Interfaces$$

$\circ$

As before, to better understand what the above definition means, we have to focus on each of the given conditions[3].

[3]Please note that, since a *NodeType* N elements cannot expose policies, we still have to check whether *ServiceTemplate*'s policies are applicable to N (as it was in Definition 3.1).

Plug-in matching of requirements

A *ServiceTemplate* ST can be used as a substitute for *NodeType* N if it exposes less requirements than N. In other words, if $ST \subseteq N$ then the set of ST requirements is a subset of the set of N requirements.

Looking at TOSCA specification [25] we observe that

- the semantics of the requirement only depends on its type. It follows that during requirements matching there is no need to consider the *name* attribute;
- a *RequirementType* can inherit semantics from another *RequirementType* by means of the *DerivedFrom* element. This force us to consider inheritance during our matching.

  Observe that, if a *RequirementType* RT is derived from a *RequirementType* RT' then RT extends the semantics of RT' (e.g. the *WebServerRequirementType* can be extended with the *ApacheWebServerRequirementType*; the former only requires a web server to be satisfied; the latter requires a web server with Apache software to be satisfied). It follows that, if we want to substitute the *NodeType* N with the *ServiceTemplate* ST, then the requirement types of ST must be the same or super-types of the one of N.

So, before giving the desired formal condition, we have to define an operator to indicate that a TOSCA element is derived from another one.

Definition 3.10. The infix operator $\vdash$ takes a pair of TOSCA elements T_1 and T_2 and returns a truth value according to the following rules:

- $T_1 \vdash T_2 =$ `true`, if T_1.*DerivedFrom* contains a reference to T_2;
- $T_1 \vdash T_2 =$ `true`, if T_1.*DerivedFrom* contains a reference to a TOSCA element T_3 such that $T_3 \vdash T_2$;

- $T_1 \vdash T_2 =$ `false`, otherwise.

$\circ$

Definition 3.11. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.RequirementDefinitions \subseteq_R ST.Requirements$$

if and only if:

$$\forall\ Requirement\ y \in ST.Requirements$$

$$\exists\ RequirementDefinition\ x \in N.RequirementDefinitions:$$

$$x.ref.type = y.requirementType \vee$$

$$x.requirementType \vdash y.ref.type.$$

$\circ$

Plug-in matching of capabilities

As before, the high correlation between capabilities and requirements TOSCA syntax lets us do the same reasoning and taking the same conclusions. There is only one difference between the two cases: this time we talk about what ST offers (and not what it requires). So, this time the inclusion relationship is between the set of N capabilities and the set of ST capabilities. Formally:

Definition 3.12. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.CapabilityDefinitions \subseteq_C ST.Capabilities$$

if and only if:

$$\forall\ CapabilityDefinition\ x \in N.CapabilityDefinitions$$

$$\exists\ Capability\ y \in ST.Capabilities:$$

$$y.ref.type = x.requirementType\ \vee$$

$$y.ref.type \vdash x.requirementType.$$

$\circ$

Plug-in matching of properties

Talking about exact matching, we sad that the properties XML type of *NodeType* N and *ServiceTemplate* ST must be the same. Observe that this a too strict condition: we only need that ST offers *at least* the same properties as N. Indeed, if ST offers more properties than N, we can hide the exceeding ones (in order to obtain the exact matching).

Definition 3.13. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.PropertiesDefinition \subseteq_{PR} ST.Properties$$

if and only if the XML type of $ST.Properties$ extends the one defined via $N.PropertiesDefinition$.

$\circ$

Plug-in matching of interfaces

In the exact matching reasoning, we want that the *ServiceTemplate* ST offers the same interfaces as *NodeType* N. This is a too coarse grain reasoning: we only need that for each operation O exposed by N exists *at least* one of the ST operations which is equal to O. If this is the case, then we can group operations in order to obtain the desired interfaces.

This finer grain reasoning let us give the formal interfaces plug-in matching condition.

Definition 3.14. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.Interfaces \subseteq_I ST.Interfaces$$

if and only if:

$$\forall\ Operation\ x \in N.Interfaces.Interface,$$

$$\exists\ Operation\ y \in ST.Interfaces.Interface:$$

$$x \equiv_O y$$

$\circ$

Observation 3.2. Please note that the operation exposed by N and the one exposed by ST sill must be in a exact match ($\equiv_O$) relationship.

3.2.2 Adaptation: the *oblivion boundaries* approach

In the previous section we have seen a plug-in manner to check whether a *ServiceTemplate* can be a substitute for a *NodeType*. It is worth noting that such a flexibility requires the introduction of a kind of service adaptation to let the *ServiceTemplate* be used in place of the *NodeType*.

Remember that we are facing the problem with a black-box approach. So, we cannot modify the *ServiceTemplate* internally: we can only operate out of its boundaries. To show how this external adaptation is done, we will start looking at what the *NodeType* offers (*CapabilityDefinitions*, *PropertiesDefinition* and *Interfaces*). Once the offerings adaptation is

done we will consider also the adaptation of what the *NodeType* needs (*RequirementDefinitions*)[4].

Capabilities, properties and interfaces adaptation

The idea of *plug-in matching* is to check whether a *ServiceTemplate ST* offers *at least* the same capabilities, properties and operations as the *NodeType N* we need to substitute. If this is the case, the next step is to restrict the set of TOSCA elements offered by *ST* to only those offered by *N*.

The desired adaptation can be done by constructing a new *ServiceTemplate ST'* via the *oblivion boundaries* approach (Figure 3.5). The

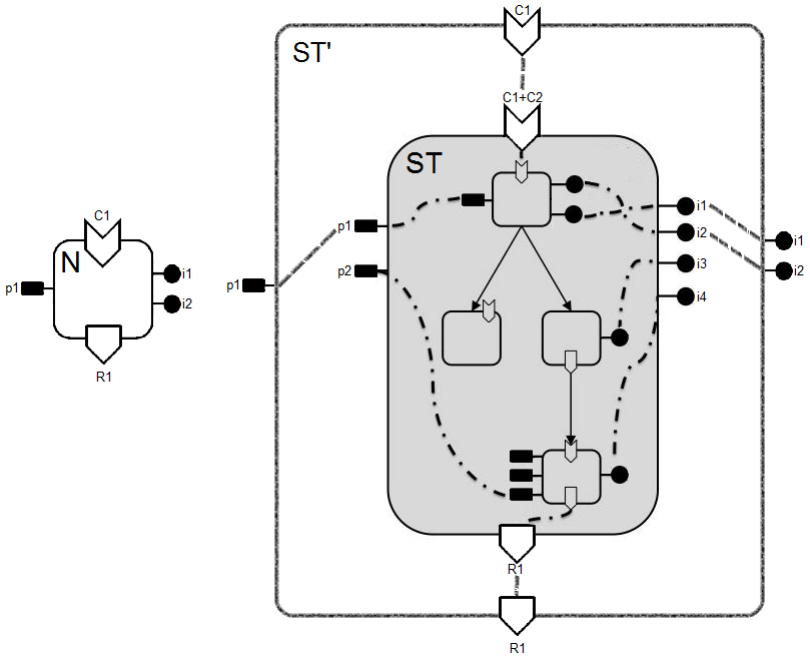


Figure 3.5: *Oblivion boundaries* adaptation approach.

new service *TopologyTemplate* will be (by now) composed by only one

[4]Since we still check whether policies are exactly matched, we do not need to adapt them.

node: the *ServiceTemplate* ST we want to adapt. Once the topology is defined, we have to decide what to offer externally (via the *Boundary-Definitions* element):

- ST' will expose only the properties defined in $N.PropertiesDefinition$ element;
- the set of capabilities exposed by ST' will be the subset of ST capabilities in plug-in matching with the ones of N. Because of possible name mismatchings, the capabilities are renamed (if needed) in order to be the same as the ones exposed by N;
- for each interface of N, the required operations are detected and grouped to form the relative interface of ST'.

Full adaptation

Let us also consider requirements in our adaptation procedure. Remember that a *ServiceTemplate* ST is compatible with a *NodeType* N only if the latter exposes more requirements than the former. This clearly means that this time we cannot restrict what ST exposes to what N. Indeed, we have to transform ST in ST' making such a kind of requirements addition. How can this be done?

In the previous paragraph, while we were talking about the *oblivion boundaries* adaptation approach (without requirements adaptation consideration), we said that we have to create a new service ST' in which ST is the only *NodeTemplate*. To compensate missing requirements we can introduce another node in our topology: the *echo* node (see Figure 3.6). This node has no functional meaning: its only purpose is to replicate[5] ST requirements and add missing requirements. With the *echo* node, our new service ST' can expose all the requirements declared by N. This let us say $ST' \equiv N$ and then ST' can be a candidate for N substitution.

[5]This replication justifies the name *echo*.

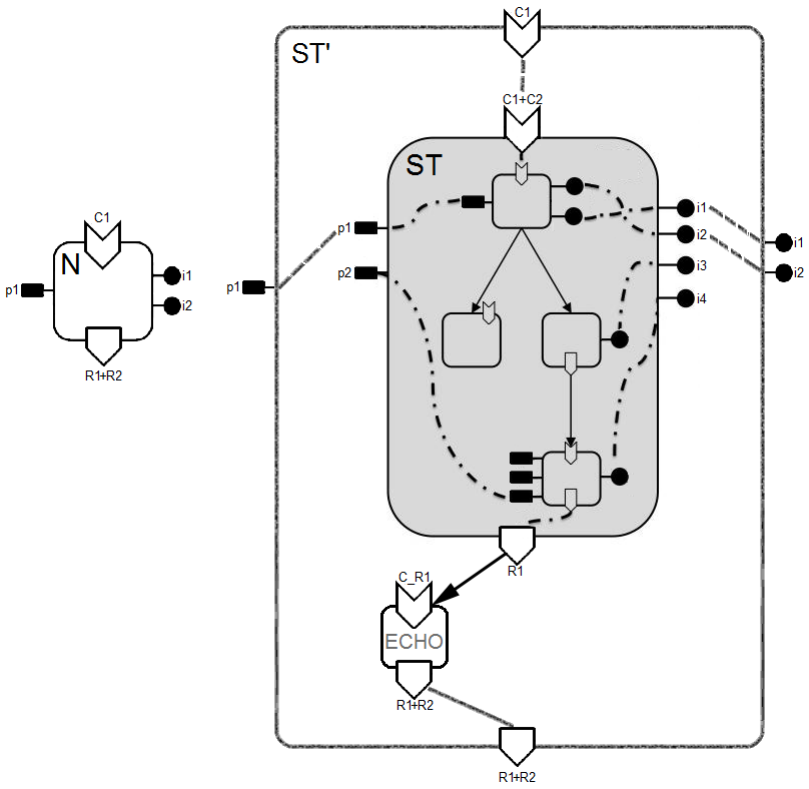


Figure 3.6: *Echo* node usage in the *oblivion boundaries* adaptation approach.

Concluding remark

Before concluding the discussion of the *oblivion boundaries* approach, we have to make some important practical observation.

Because of our adaptation approach only requires to restrict the available service boundaries and (eventually) add some new requirements, one could think of simplifying the ST boundaries so as to make them equal to those desired. Is it a right approach? Let us think about what to transform the ST specification in the ST' one implies. If we change that specification, then we have to develop a new cloud service application (which is clearly an extremely expensive approach). So, it is important to maintain the ST' specification as an adaptation of ST since this is a valuable information for deploying the needed adaptation.

Consider now the situation in which N is a *NodeType* required by a client and ST is a *ServiceTemplate* offered by a provider. Once ST' specification has been defined, the client and the provider could behave as follows:

1. the client still interacts with ST without considering the oversupplied features. If this is the case, no adaptation is implemented (and the provider believes that the client does not use what she has not required);

2. the client interacts with a cloud service application implementing ST' (offered by the provider). Typically, this application is obtained by generating an adaptor for the available application implementing ST.

3.2.3 Plug-in matching (and adaptation) examples

Let us reconsider the matching example of Figure 3.4. With the too strict exact matching conditions the compatibility checking fails. Our

new plug-in matching (and *oblivion boundaries* adaptation) approach instead makes that checking be satisfied.

A more complete example of plug-in matching can be done with the situation in Figure 3.7. In this case we want to match an available *ServiceTemplate* ST with a *NodeType* N. The *NodeType* under consideration has the following characteristics:

- it offers less properties and interfaces[6] than ST;
- it offers the same capabilities as ST;
- it needs a requirement (*ApacheApplicationContainerRequirement*) which type is derived from the one of the requirement exposed by ST (*ApplicationContainerRequirement*).

With this configuration we can easily verify the truth of the following condition:

$$N.CapabilityDefinitions \equiv_C ST.Capabilities \wedge$$

$$N.RequirementDefinitions \subseteq_R ST.Requirements \wedge$$

$$N.PropertiesDefinition \subseteq_{PR} ST.Properties \wedge$$

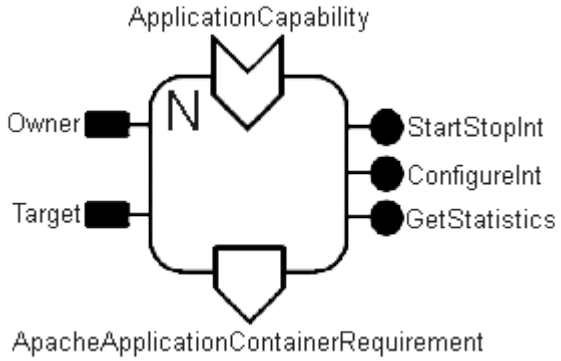
$$N.Interfaces \subseteq_I ST.Interfaces.$$

In other words, we can easily check that $ST \subseteq N$. This means that we can adapt the service ST in order to obtain a service ST' which exact matches N (see Figure 3.8).

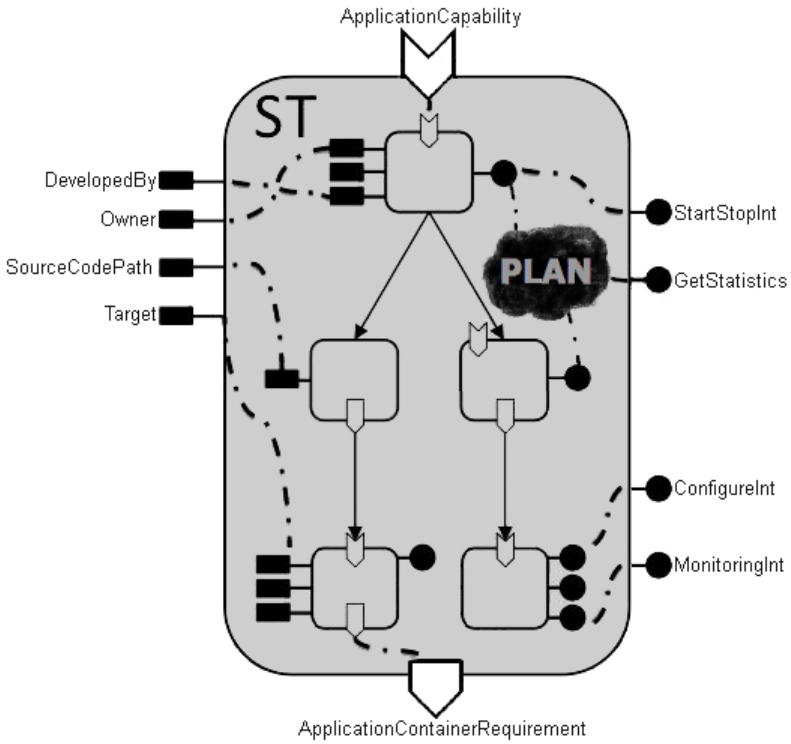
Let us now modify the *ServiceTemplate* of Figure 3.7.(b) as reported in Figure 3.9. Suppose that:

- the *GetStatistics* interface of N (see Figure 3.7.(a)) contains the only *Get* operation;

[6]For the sake of simplicity, we are assuming that *Interface* elements with the same *name* are offering the same *Operation* elements.



(a)



(b)

Figure 3.7: TOSCA *ServiceTemplate* plug-in matching example.

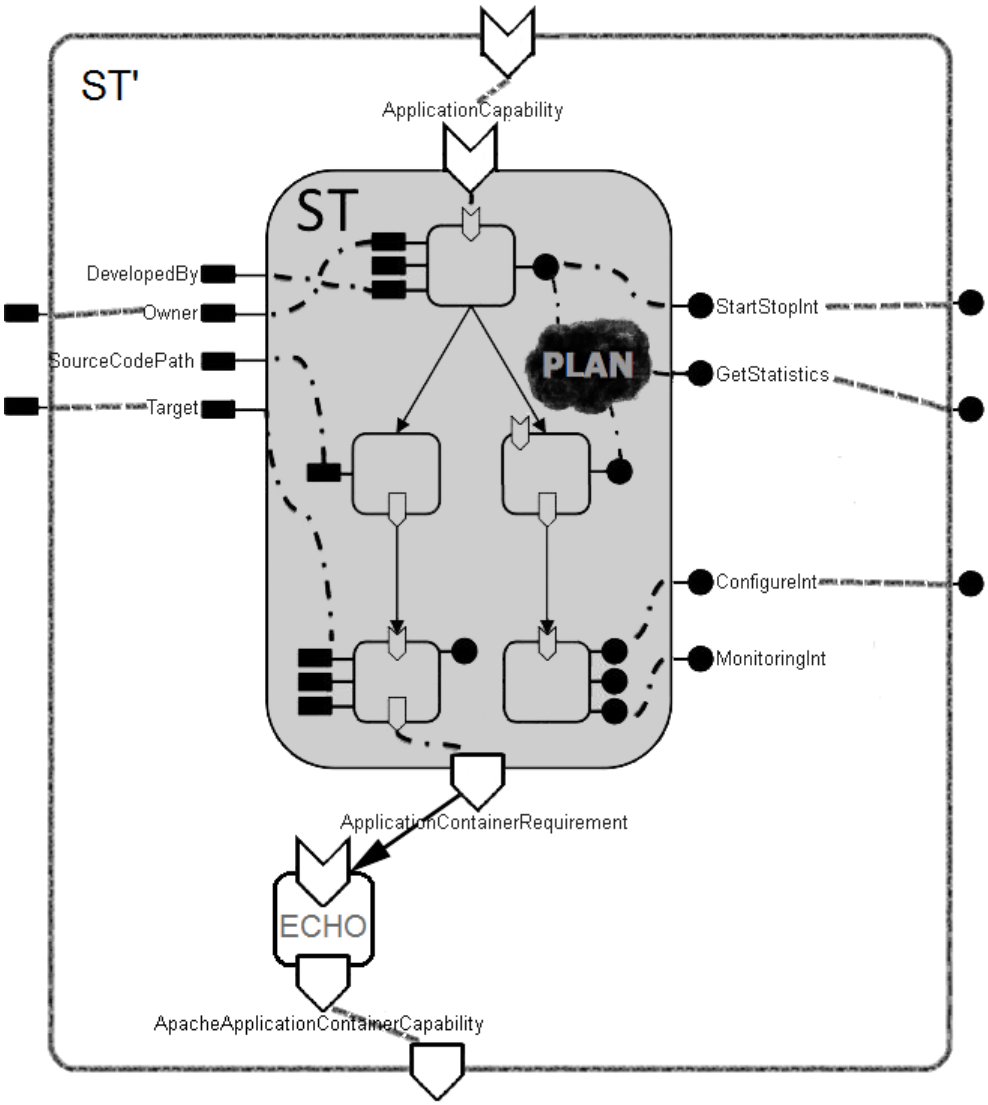


Figure 3.8: TOSCA *ServiceTemplate oblivion boundaries* adaptation example.

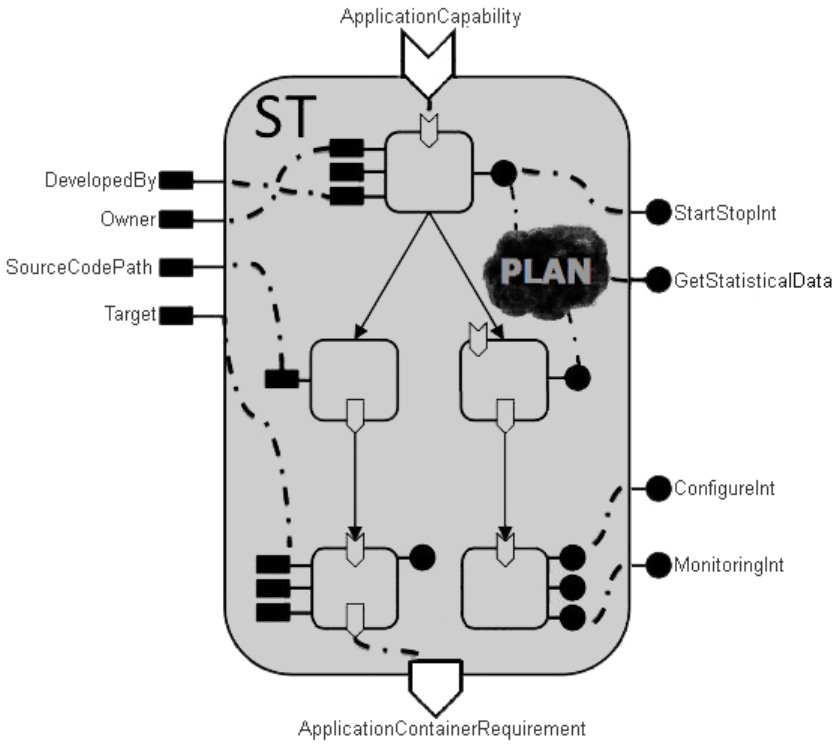


Figure 3.9: TOSCA *ServiceTemplate* plug-in matching example (modified).

- the *GetStatisticalData* interface of ST contains the only *GetData* operation (which is semantically equivalent[7] to *Get*).

Clearly, with the $\subseteq$ operator, N cannot be considered compatible with ST. So, despite the two operation are semantically equivalent, the compatibility check fails.

How can we deal with this problem? So far we have worked at a syntactic level. Introducing such a kind of semantics will let us overcome problems such as the one previously indicated.

3.3 Flexible matching

With the examples given in Section 3.2 we have shown how the purely syntactical matching (defined by the operator "$\subseteq$") could not be enough to check whether a *NodeType* element can be substituted by a *ServiceTemplate* element.

The objective of this section is to overcome the exposed problem using some kind of semantic checking. Please note that, since:

- *NodeType* elements cannot specify any kind of policy, and
- requirements (capabilities) semantics is specified[8] via their *RequirementType* (*CapabilityType*),

the semantic check only affects properties and interface operations.

[7] With *semantically equivalent* we mean that it requires the same input parameters and produces the same output parameters.

[8] As TOSCA [25] authors says.

3.3.1 Definition of *flexible matching*

What we are now going to do is to define a new operator "$\cong$" analogue to "$\subseteq$" and "$\equiv$" previously introduced: it takes a pair $\langle$*NodeType*,*ServiceTemplate*$\rangle$ and returns a truth value (which is `true` if the two input elements are in *flexible* matching, `false` otherwise).

Understanding how the new operator works requires to clarify what *flexible* matching means. Consider the *NodeType* N and the *ServiceTemplate* ST; we say that $N \cong ST$ if the *plug-in* matching fails only because of the presence of operations and/or properties which are syntactically different but semantically equivalent. So, formally, we only have to modify the "$\subseteq$" conditions about interfaces and properties.

Definition 3.15. A *NodeType* N flexibly matches a *ServiceTemplate* ST ($N \cong ST$) if and only if:

$$N.RequirementDefinitions \subseteq_R ST.Requirements \wedge$$

$$N.CapabilityDefinitions \subseteq_C ST.Capabilities \wedge$$

$$PolicyType \text{ applicable to } N \equiv_{PO} ST.Policies \wedge$$

$$N.PropertiesDefinition \cong_{PR} ST.Properties \wedge$$

$$N.Interfaces \cong_I ST.Interfaces$$

$\circ$

Observation 3.3. Please note that, if two properties (or interfaces) are syntactically (and semantically) equivalent, then they are both plug-in and flexibly matched. This intuitively means that the following property holds:

$$N \subseteq ST \Longrightarrow N \cong ST.$$

Let us clarify what the new properties and interfaces matching conditions means.

Required assumptions

Before giving the desired *flexible* matching conditions, we have to introduce some concerns about semantics.

In the following, we will use ontologies to associate semantic meaning to TOSCA elements under consideration. This force us to make the following assumption.

Assumption 3.2. All cloud service applications are equipped with ontologies (which associate semantic meaning to all TOSCA element names used in their definitions).

This let us assume that the *NodeType* and the *ServiceTemplate* we want to match have semantics associated to their names. So, we can proceed in checking whether their elements' semantic meaning is the same. But, how can we perform such a checking?

Assumption 3.3. Suppose that a cross-ontology matchmaker COM is available. This matchmaker let us verify whether two different ontologies concepts have equivalent semantic meanings.

In the following we will use the notation $C_a \vdash\dashv C_b$ to indicate the semantic checking (done by COM) between different ontologies concepts C_a and C_b, where $C_a \vdash\dashv C_b$ if and only if they are semantically equivalent.

Flexible matching of properties

The Definition 3.13 says that $N.PropertiesDefinition$ plug-in matches $ST.Properties$ if and only if the XML type of the latter extends the one of the former. In other words, each property P_N defined in the $N.PropertiesDefinition$ element must correspond to a property P_{ST} of the *ServiceTemplate* such that:

- $P_{ST}.type$ is (the same as or) a sub-type of $P_N.type$, and

- $P_{ST}.name$ is equal to $P_N.name$.

If the first condition is not satisfied, then there are no chances to adapt the *ServiceTemplate* ST to make it be compatible with N. Vice versa, if two properties do not have the same name, then we can check if they are semantically equivalent.

Definition 3.16. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.PropertiesDefinition \cong_{PR} ST.Properties$$

if and only if

$$\forall\ propertyDefinition\ x \in N.PropertiesDefinition,$$

$$\exists\ property\ y \in ST.Properties:$$

$$\text{the XML type of } y \text{ extends the one defined by } x \wedge$$

$$y.name \vdash\dashv x.name$$

$\circ$

Flexible matching of interfaces

Let us now consider the operations (and interfaces) flexible matching problem. What we want to do is to let an operation O_1 be substituted by another operation O_2 which can be considered semantically equivalent. This semantic equivalence can be explained as follows:

- the O_1 input (output) parameters number is the same as the one of O_2;
- for each O_1 input (output) parameter exists a O_2 input (output) parameter of the same type which name is semantically equivalent to the one of O_1;

Definition 3.17. Consider two *Operation* elements O_1 and O_2. We say that

$$O_1 \cong_O O_2$$

if and only if the following conditions are satisfied:

1. $|\{O_1.InputParameters\}| = |\{O_2.InputParameters\}|$
2. $\forall$ *InputParameter* $b_{in} \in O_2.InputParameters$
   $\exists$ *InputParameter* $a_{in} \in O_1.InputParameters$:
   $a_{in}.name \vdash\dashv b_{in}.name \wedge a_{in}.type \vdash b_{in}.type$
3. $|\{O_1.OutputParameters\}| = |\{O_2.OutputParameters\}|$
4. $\forall$ *OutputParameter* $a_{out} \in O_1.OutputParameters$
   $\exists$ *OutputParameter* $b_{out} \in O_2.OutputParameters$:
   $b_{out}.name \vdash\dashv a_{out}.name \wedge b_{out}.type \vdash a_{out}.type$

$\circ$

So, we have all the fundamentals needed to define the flexible matching condition between *NodeType* and *ServiceTemplate* interfaces.

Definition 3.18. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.Interfaces \cong_I ST.Interfaces$$

if and only if:

$$\forall \textit{ Operation } x \in N.Interfaces.Interface$$

$$\exists \textit{ Operation } y \in ST.Interfaces.Interface:$$

$$x \cong_O y$$

$\circ$

Concluding remark

Note that (by using the "$\vdash\dashv$" notation) we abstract from how the semantic checking is done. To do it we can use one of the already available cross-ontology matchmakers. In other words, according to the purpose of this thesis, in the following we will not consider the problem of *realizing a cross-ontology matchmaker* but we will consider the usage of one already available (such as the one proposed by Martinez-Gil et al. [20]).

3.3.2 Adaptation: extended *oblivion boundaries* approach

In the previous subsection we have seen how it is possible to use ontological semantics in order to obtain the *flexible* matching. What we have done is simply modify the properties and interfaces *plug-in* matching (done by the "$\subseteq$" operator) introducing the semantic checking. Similarly, modifying properties and interfaces adaptation, we can reuse the *oblivion boundaries* adaptation approach.

Observe that the only (but fundamental) checking modification is in the possibility for properties and operations to have different names. So, simply introducing the *ServiceTemplate* properties and operation renaming (in order to match what the *NodeType* exposes), the *oblivion boundaries* can be reused.

3.3.3 Flexible matching (and adaptation) examples

Reconsider the failed example of Section 3.2 (Figure 3.9). With the *plug-in* matching (and adaptation) approach our checking fails because of the presence of operations which names are different. Assume that:

- the operations *Get* and *GetData* do not have any input parameter,

and

- *Get* and *GetData* expose the same number of output parameters (and those parameters name are semantically equivalent).

Under the above assumptions the condition

$$Get \cong_O GetData$$

is `true`. This means that we can easily verify that $N \cong ST$. So, we can adapt the *ServiceTemplate* ST in order to match N with the (extended) *oblivion boundaries* approach.

3.4 Concluding remarks

In this chapter we have seen three different ways to black-box match a *NodeType* N with a *ServiceTemplate* ST. Before going on with our discussion, we have to make some observations.

3.4.1 Matching and adaptation

As reported in Figure 3.10, the proposed black-box matching (and adaptation) approach can be fully automated in order to generate the adapted ST' *ServiceTemplate*. In other words, we can:

- check whether (and how) the two TOSCA elements under consideration match, and
- (possibly) generate the desired ST adaptation.

Looking at Figure 3.10, we observe that, if one of the proposed black-box matching conditions is satisfied, then we can generate an adapted service

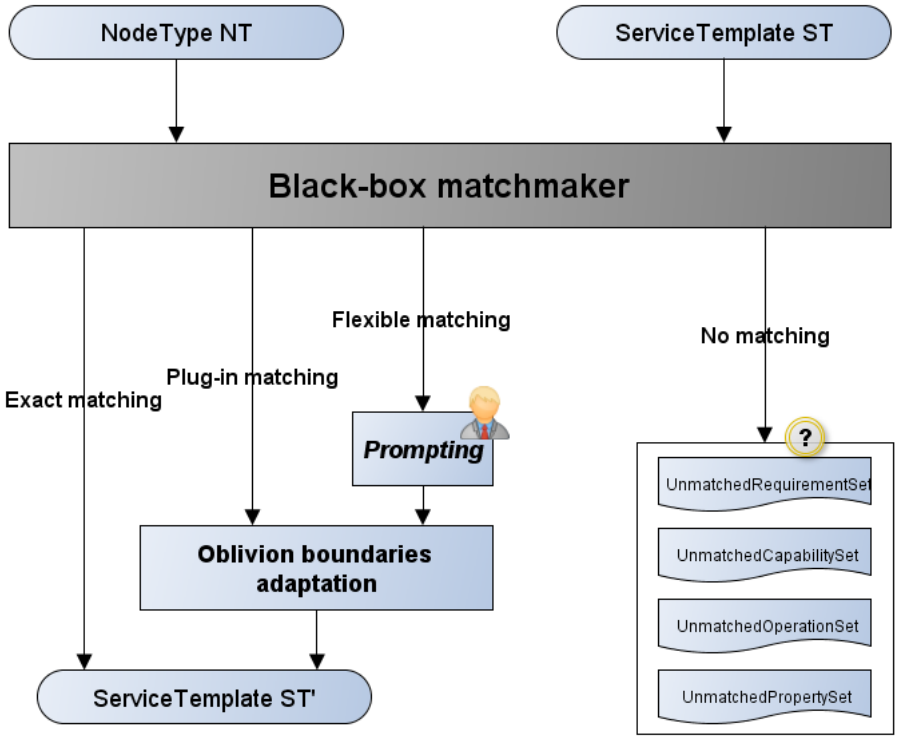


Figure 3.10: Black-box matching (and adaptation) procedure.

ST'. This means that we always follow the approach of Figure 3.11: instead of developing a new service, we use a (automatically generated) adaptor A to interact with the available service ST as if it were the desired ST'.

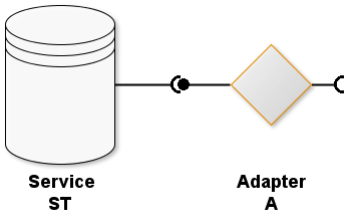


Figure 3.11: Adaptation approach.

The above considerations suggest us a question: which kind of adaptation does A have to implement? To answer it we have to distinguish the possible matching cases. Consider, as always, the *ServiceTemplate* ST and the *NodeType* N.

Exact matching If $N \equiv ST$, then ST exposes exactly what N needs (and so there is no need of adapting ST). This means that, in order to let ST' be the same service as ST, the generated adaptor A must implement the *identity* function.

Plug-in matching If $N \subseteq ST$, then ST exhibits all the features needed by N and some features that N does not need. If this is the case, then we have to restrict what ST exposes to only what N needs. In other words, the function that A has to implement is a *filtering* one.

Flexible matching The last matching case ($N \cong ST$) means that ST exposes

- features that are equivalent to all those required by N (possibly with different names), and

- some undesired features.

This implies that the adaptor A must implement a function which *filters* and (possibly) *renames* those features.

3.4.2 Taking semantics into account

When we talked about the *flexible* matching approach, we stated that ontologies should be used to check whether two features could be considered (semantically) equivalent. So, the generated pairings between (semantically) equivalent features strictly depend on the given ontologies and on the cross-ontology matchmaker. It follows that, if the input ontologies are not so accurate as required, then some generated pairings could not be significant and should not be used in our matching (and adaptation). How can we deal with this problem?

Looking at Figure 3.11 we can observe that when the considered *NodeType* N and *ServiceTemplate* ST are in flexible matching, we have to prompt to the user the generated pairings. If she accepts the matchmaker decisions, then we can proceed in developing the adapted service ST' (via the *oblivion boundaries* adaptation approach).

Please note that the way in which this prompting is performed strictly depends on the matchmaker implementation. We recommend to start giving the most probable pairings and to let the user mark and/or adjust the wrong pairings.

3.4.3 Treatment of mismatchings

We are studying the matching problem between a *NodeType* N and a *ServiceTemplate* ST. So far, we have introduced a methodology to check whether those two TOSCA elements are in (one of the possible) black-box matching. Clearly, there are cases in which N and ST do not black-box

match. So, what can we do when none of the exposed conditions is satisfied? We will see how to proceed in the following chapter.

Chapter 4

NodeType white-box matching (and adaptation)

The previous chapter has provided an (automatable) approach which let us check whether a *NodeType N* matches a *ServiceTemplate ST* (without looking "inside" *ST*). In this chapter we are going to see whether, using a white-box viewpoint, we can extend our matching procedure.

The chapter will start illustrating some examples which make the black-box matching procedures fail (Section 4.1). Then, it will proceed providing the white-box matching (and adaptation) approach (Sections 4.2 and 4.3). Finally, with Section 4.4, some concluding remarks are given.

4.1 Motivating example

Let us consider the *ServiceTemplate ST* in Figure 4.1. Suppose now we want to match it with the *NodeType* elements in Figure 4.2. Using the black-box matching conditions our checking fails because both $WAppType1$ and $WAppType2$ expose more TOSCA elements than *ST*.

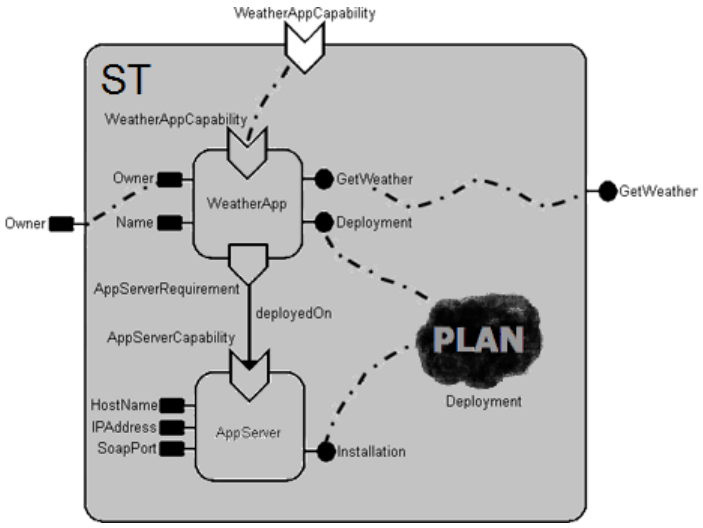
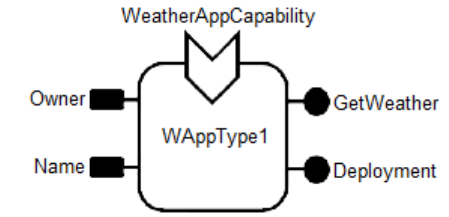
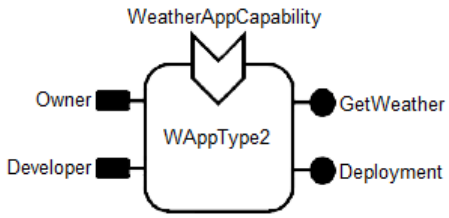


Figure 4.1: Example of available *ServiceTemplate*.



(a)



(b)

Figure 4.2: Example of desired *NodeType* elements.

Let us now try to use a white-box approach. Looking inside ST we can easily check that:

- the $WAppType2$ *NodeType* element cannot be matched with ST (because of the lack of *Developer* property presence);
- ST contains the *Name* (equivalent) property and the *Deployment* operation required by $WAppType1$. More precisely, the *Deployment* operation can be obtained by generating a plan which combines available operations.

So, if we could find the missing features inside the available *ServiceTemplate*, then we could extract them in order to make our service black-box match the desired *NodeType*.

4.2 White-box matching (and adaptation)

Please reconsider Figure 3.10. Looking at the "no matching" case, we observe that our matching procedure could provide us the set of unmatched features (more precisely, those features could be given as a multiset which contains the *UnmatchedRequirementSet*, the *UnmatchedCapabilitySet*, the *UnmatchedOperationSet* and the *UnmatchedPropertySet*).

As observed in the previous section, we should find a way to exhibit all the unmatched required features via the boundaries of the available *ServiceTemplate* (if possible). This means that for each unmatched feature under consideration we should perform two steps:

1. **Matching** - an equivalent feature (if present) must be detected inside ST;
2. **Adaptation** - the boundaries of ST must be modified in order to expose the desired feature.

If this can be done, then we can reuse the black-box flexible matching approach in order to obtain the desired adaptation of the available *ServiceTemplate*.

Observation 4.1. Please note that (as done for flexible matching and adaptation), we have to use some kind of semantics in order to couple desired and available features. This means that we have to reconsider the following (ontology-about) assumptions.

Assumption 4.1. All cloud service applications are equipped with ontologies (which associate semantic meaning to all TOSCA element names used in their definitions).

Assumption 4.2. Suppose that a cross-ontology matcher COM is available. This matcher let us verify whether two different ontologies concepts have equivalent semantic meanings.

The above considerations give us an informal (though incomplete[1]) way of matching in a white-box viewpoint. In the rest of this section we will try to formalize this notion.

4.2.1 White-box matching condition

The objective of this subsection is to define a new operator "$\boxed{\cong}$" analogue to those introduced for the black-box matching (Chapter 3). This new operator takes a pair $\langle NodeType, ServiceTemplate \rangle$ and returns a truth value. More precisely, it returns `true` if the two input elements are in *white-box* matching, `false` otherwise.

As done before, we can define the new operator meaning in a step-wise way.

Definition 4.1. A *NodeType* N white-box matches a *ServiceTemplate* ST ($N \boxed{\cong} ST$) if and only if:

[1] As we will see, dealing with unmatched operations treatment is more complex than simply searching (and exposing) the feature.

$$N.RequirementDefinitions \subseteq_R ST.Requirements \wedge$$

$$N.CapabilityDefinitions \boxed{\cong_C} ST.Capabilities \wedge$$

$$\text{PolicyType applicable to } N \equiv_{PO} ST.Policies \wedge$$

$$N.PropertiesDefinition \boxed{\cong_{PR}} ST.Properties \wedge$$

$$N.Interfaces \boxed{\cong_I} ST.Interfaces$$

$\circ$

Observation 4.2. Please note that we still use the policies (black-box) exact matching condition. As before, this is because we only have to check whether the *ServiceTemplate*'s policies are applicable to the considered *NodeType*.

Matching of requirements

Consider the *UnmatchedRequirementSet* obtained when the black-box matching procedures fail. How can we treat such set? It is worth pointing that there is no need to search such requirements inside ST. In fact, if the *ServiceTemplate* ST exposes less requirements than the *NodeType* N, then the requirement plug-in matching condition will be satisfied. This explains why, in Definition 4.1, we still employ the

$$N.RequirementDefinitions \subseteq_R ST.Requirements$$

condition.

Observation 4.3. As we will see, (since we employ the requirement plug-in matching condition) the adaptation of the white-box matched *ServiceTemplate* ST will generate a new *ServiceTemplate* ST'' such that

$$N \subseteq ST''.$$

Matching of capabilities

Consider a *ServiceTemplate* ST and a *NodeType* N. To understand what

$$N.CapabilityDefinitions \boxed{\cong_C} ST.Capabilities$$

means we have to think about when this kind of matching occurs. The *white-box* approach will be used when the black-box *flexible* one fails. This means that N exhibits some capabilities that are not visible in ST.

What can we do to overcome the above exposed problem? We have to search inside ST the desired features. In particular, we have to look at in-nested *NodeTemplate* elements. To do it we will use the notation

$$ST \rightarrow NodeTemplate$$

to indicate that we are navigating[2] the XML TOSCA *Definitions* tree in order to reach in-nested *NodeTemplate* elements.

Definition 4.2. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.CapabilityDefinitions \boxed{\cong_C} ST.Capabilities$$

if and only if:

$\forall$ *CapabilityDefinition* $x \in N.CapabilityDefinitions$
($\exists$ *Capability* $y \in ST.Capabilities$:
$y.ref.type = x.capabilityType \vee y.ref.type \vdash x.capabilityType$)
$\vee$
($\exists$ *Capability* $y \in ST \rightarrow NodeTemplate.Capabilities$:
$y.ref.type = x.capabilityType \vee y.ref.type \vdash x.capabilityType$).

$\circ$

[2]The "$\rightarrow$" notation's meaning is analogue to that of the "//" (XPath [36]) operator
.

Matching of properties

What we want to do now is to clarify the meaning of the following condition:

$$N.PropertiesDefinition \boxed{\cong_{PR}} ST.Properties$$

As before, we have to look inside ST to find the desired (unmatched) properties. To understand which in-nested elements should be considered in properties white-box matching we have to answer to the following question: by which kind of TOSCA elements could properties be exhibited? The answer is: *RelationshipTemplate* and *NodeTemplate*.

Definition 4.3. Let N be a *NodeType* and ST be a *ServiceTemplate*. We say that

$$N.PropertiesDefinition \boxed{\cong_{PR}} ST.Properties$$

if and only if:

$\forall$ *propertyDefinition* $x \in N.PropertiesDefinition$
($\exists$ *property* $y \in ST.Properties$:
the XML type of y extends the one defined by x
$\wedge$
$y.name \vdash\dashv x.name$)
$\vee$
($\exists$ *property* $y \in ST \rightarrow NodeTemplate.Properties$:
the XML type of y extends the one defined by x
$\wedge$
$y.name \vdash\dashv x.name$)
$\vee$
($\exists$ *property* $y \in ST \rightarrow RelationshipTemplate.Properties$:
the XML type of y extends the one defined by x

$\wedge$

$y.name \vdash\dashv x.name$).

$\circ$

Matching of interface operations

So far, to check whether a *NodeType* N white-box matches a *ServiceTemplate* ST, we only search for (black-box) missing features inside ST. If we use the same approach to explain the meaning of:

$$N.Interfaces \boxed{\cong_I} ST.Interfaces,$$

then we do not consider its whole meaning; indeed, to overcome the missing interface operation problem, we should also check whether combining (some of) the available operations is possible to obtain the desired one.

Definition 4.4. Let N be a *NodeType*, ST be a *ServiceTemplate* and OC_{ST} the set of all possible plans combining ST's operations. We say that

$$N.Interfaces \boxed{\cong_I} ST.Interfaces$$

if and only if:

$\forall$ *Operation* $x \in N.Interfaces.Interface$

$\exists$ *Operation* $y \in ST.Interfaces.Interface$:

$x \cong_O y$

$\vee$

$\exists$ *Plan* $p \in OC_{ST}$:

$x \cong_O$ (*Operation*) p.

$\circ$

Observation 4.4. Please note that:

- *Operation* elements are compared by the black-box flexible matching operator ($\cong_O$). This is because we can look at those TOSCA elements only from a black-box viewpoint (since they are always shown in terms of their name and their input/output parameters);
- since the operator $\cong_O$ checks whether two operations expose equivalent input/output parameters, it can be used also to check if an *Operation* flexibly matches a *Plan* (provided that the latter is treated as an *Operation*).

Concluding remark

Looking at the step-wise definition of the white-box matching operator $\boxed{\cong}$ we can intuitively derive the following implication. Let N be a *NodeType* and ST be a *ServiceTemplate*

$$N \cong ST \implies N \boxed{\cong} ST.$$

This is because the white-box matching definition looks for feature both on $ST.BoundaryDefinitions$ and inside ST. So, if all required (equivalent) features are on the boundaries of ST, then both *flexible* and *white-box* matching condition are satisfied.

4.2.2 Adaptation

In the previous subsection we have introduced a way to *white-box* match a *NodeType* N and a *ServiceTemplate* ST. Clearly, if $ST \boxed{\cong} N$, this does not mean that the former can be (immediately) used as a substitute for the latter. Indeed, we have to adapt ST in order to be used in place of N.

Adaptation of capabilities and properties

If we omit interface operations from consideration, then the adaptation process simply consist in modifying the ST boundaries in order to exhibit those features that cannot be seen from a black-box viewpoint.

Adaptation of interface operations

The above adaptation approach is not enough if used to exhibit missing operations. This is because, despite it can be used to exhibit operations simply hidden inside ST, it cannot solve the problem of operations taken from OC_{ST} (viz., the set of all possible plans combining ST's operations). Indeed, to solve the latter problem we have to:

1. look at (all) possible plans combining ST's operations and check whether some of them flexibly match the unavailable operations;
2. if this is the case, store those plans inside ST and modify its boundaries in order to exhibit those plans as available operations.

Full adaptation

Once the *white-box* matching is done and the available *ServiceTemplate* (ST) boundaries have been modified, we can look at ST as a (new) *ServiceTemplate* ST''. Since the objective of *white-box* matching is to modify ST in order to black-box match *NodeType* N, the condition

$$ST'' \cong N$$

holds. It follows that we can simply reuse the (extended) *oblivion boundaries* adaptation approach in order to let ST'' be included in the adapted service ST' (which is the one with the client interacts).

Concluding remark

So far we have implicitly assumed to have OC_{ST} (the set of all possible plans combining ST's operations) to be somehow available.

In the following section we will focus on how to:

- generate (all) possible plans combining ST's operations, and
- check whether some of them flexibly match the missing operations.

4.3 Generating plans

In this section we will provide a solution that overcomes the plan generation problem. This solution will follows the two steps approach proposed in [5] by Brogi et al.

Functional dependency synthesis We know that each operation is equipped with functional information (i.e., its input and output parameters). This functional information defines functional dependencies within and among operations. Hence, we need to represent the relationships which state "which set of input parameters an operation requires in order to produce a set of output parameters" (intra-operation dependencies), as well as "which set of output parameters produced by an operation is required as input by another one" (inter-operation dependencies).

Furthermore, in both Sections 3.3 and 4.2, we made our checking using some kind of semantics[3]. Hence, it is also necessary to represent those relationships that state "which parameters are semantically equivalent".

[3]We used ontologies to describe (and compare) concepts by which our features are annotated.

So, we have to collect all those dependencies into a suitable data structure. As we will see (Subsection 4.3.1), the first step builds a dependency hypergraph, whose nodes represent functional attributes of operations (i.e., their input/output parameters), and whose hyperedges represent relationships among them.

Operation sequence detection Once the dependency hypergraph is built, we can proceed with the second step. Given a missing operation O, we can explore the hypergraph to detect which operation sequence:

- takes as input O's input parameters, and
- produces as output its output parameters.

More precisely, the parameters taken as input and produced as output by the detected sequence are semantically equivalent to those of O.

4.3.1 Functional dependency synthesis

Before describing the dependency hypergraph and how to build it, we include hereafter the definitions of hypergraph, directed hypergraph and directed hyperedge (as described in [15]).

Definition 4.5. A *hypergraph* is a pair $H = \langle V, E \rangle$, where

- $V = \{v_1, v_2, ..., v_n\}$ is a set of vertices (or nodes), and
- $E = \{E_1, E_2, ..., E_m\}$, with $E_i \subseteq V$ for $i = 1, ..., m$, is a set of hyperedges.

Note that when $|E_i| = 2$, $i = 1, ..., m$, the hypergraph is a standard graph.

∘

Definition 4.6. A *directed hypergraph* is a hypergraph with directed hyperedges. A *directed hyperedge* is an ordered pair, $E = \langle X, Y \rangle$, of (possibly empty) subsets of vertices. X is the tail of E, denoted by $T(E)$, while Y is its head, denoted by $H(E)$.

∘

Figure 4.3 is illustrates an example of directed hyperedge with tail $\{x_1, x_2\}$ and head $\{y_1, y_2, y_3, y_4\}$.

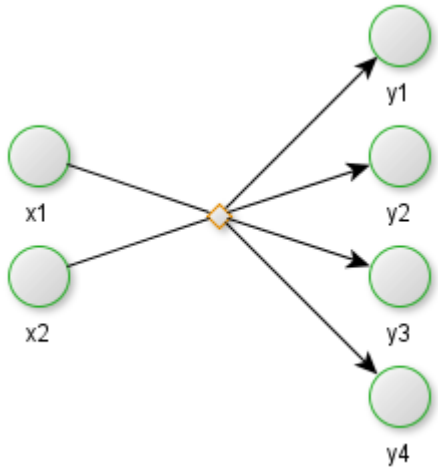


Figure 4.3: A directed hyperedge.

The dependency hypergraph

A hypergraph is a suitable notation to represent parameters (as nodes) and dependencies among them (as hyperedges). As mentioned earlier, we have to distinguish between intra-operation and inter-operation dependencies.

Definition 4.7. A *labelled directed hypergraph* (E, V, l) is a directed hypergraph $\langle E, V \rangle$ with a labelling function $l : E \to A$ assigning to each hyperedge a label from a given alphabet A.

A *labelled directed hyperedge* is denoted by a triple $E = \langle X, Y, a \rangle$, where X, Y, a denote the tail, the head and the label of E, respectively.

∘

The hyperedge labelling availability let us obtain the desired discrimination.

Intra-operation dependency Let O be an *Operation*. The output parameters of O *intra-operation* depend on its input parameters. It follows that there must exists a (E_{intra}) hyperedge from $O.InputParameters$ to O.OutputParameter labelled with O:

$$\langle O.InputParameters, O.OutputParameters, O \rangle \in E_{intra}.$$

Inter-operation dependency To understand this kind of dependency we have to look at parameter semantics. Let p_1 be an output parameter of an operation O_1 and p_2 be an input parameter of an operation O_2. It is worth noting that, if $p_1.name$ is a sub-concept of $p_2.name$ (and, obviously, $p_1.type$ is - the same type a or - a subtype of $p_2.type$), then the output parameter p_1 can be used as input to operation O_2. The same substitution can be done if those parameter names are semantically equivalent.

Let us consider (separately) the above stated situations:

- let p be a parameter and let OS be the set of parameters whose names are (direct) sub-concept of $p.name$ and whose types are the same type as or a subtype of $p.type$. Then there must exist a (E_{sub}) hyperedge from OS to p:

$$\langle OS, \{p\}, nil \rangle \in E_{sub}$$

- let p_1 and p_2 be two (type-compatible) parameters whose names are semantically equivalent. Since we are working on the same

cloud service application, we can state that the problem of checking whether two operation names are semantically equivalent is not a cross-ontology one. It follows that, if two parameters have names which are semantically equivalent, then they will expose the same name. In other words, they will correspond to the same node in our hypergraph. So, there is no need of putting an hyperedge between them.

Before continuing our discussion, we have to clarify the sub-concept of relationship. With respect to the hierarchical structure of an ontology, c is a (direct) sub-concept of d if c is a child of d. So, E_{sub} hyperedges will (only) be between (type-compatible) parameters which are in this kind of relationship.

Please note that if c is a (direct) sub-concept of e and e is a (direct) sub-concept of d, then also c is a sub-concept of d. So, we have to extend our sub-concept notion in terms of the dependency hypergraph. Namely, c is a sub-concept of d if and only if there exist a path from c to d which consists of sub-concept relationships (i.e., it goes through E_{sub} hyperedges).

Dependency hypergraph construction

We have seen what *dependency hypergraph* means. The next question is how to build a dependency hypergraph.

The answer stays in the more obvious solution. Starting from the empty hypergraph $H = \langle V, E, l \rangle$ (with $V = \varnothing$ and $E = \varnothing$), we have to proceed for each operation O as follows:

1. for each input parameter $in \in O.InputParameters$, if $in \notin V$ then
   (a) add in to V;
   (b) modify E in order to make in point to the type-compatible

parameter p (if present) such that $in.name$ is a (direct) sub-concept of p.name;

(c) let $S \subseteq V$ be the set of type-compatible parameters whose names are (direct) sub-concept of $in.name$. Add the

$$\langle S, \{in\}, nil \rangle \in E_{sub}$$

hyperedge to E.

2. for each output parameter $out \in O.OutputParameters$, if $out \notin V$ then proceed as for input parameters.

3. add to E the hyperedge

$$\langle O.InputParameters, O.OutputParameters, O \rangle \in E_{intra}.$$

Please note that there is no need to consider *Plans* (as operations) in this procedure. This is because *Plans* are in turn operation sequences. So, they will be generated again with the *operation sequence detection* phase[4].

Practical considerations about complexity

While building a dependency hypergraph has exponential complexity (all pairs of operation must be composed), since the set of service components available operations is a static one, this procedure is executed only once. So, the high complexity could be paid only at the start of our (white-box) matching procedure.

Furthermore, a clever service provider could think about storing each *ServiceTemplate* along with its own dependency hypergraph. If this is the case, the matching cost will be significantly reduced because the

[4]Furthermore, if present, plans are exhibited out of service boundaries (as available operations). So, since they were matched from a black-box viewpoint, there is no need to search them from a white-box one.

hypergraph will be already available (and it won't be required to build it).

Last (but not least), the above consideration suggest us to have the dependency hypergraph somewhere available when we are going to start the *operation sequence detection* phase. So, in the following subsection we will explain how to perform that phase assuming the dependency hypergraph availability (as input).

4.3.2 Operation sequence detection

We are facing the problem of white-box matching between a *NodeType* N and a *ServiceTemplate* ST. More precisely, we are checking whether a N's *Operation* O can be substituted by a sequence of operations inside ST (since none of the available exposed operations flexibly matches with O). So, we have to search in the (available) dependency hypergraph for a functionally equivalent operation sequence. Before formalizing this equivalence notion, we will introduce a sub-concept (and type-compatibility) operator to increase readability.

Definition 4.8. Let a and b be two *Operation* parameters. We will say that a is (type-compatible with and) sub-concept of b ($a \lhd b$) if and only if

$$a.type \text{ is (the same as or) a sub-type of } b.type \wedge$$

$$a.name \text{ is a sub-concept of } b.name.$$

$\circ$

Definition 4.9. Let O be an *Operation* and OS be a set of *Operation* elements. Then O is functionally equivalent to OS ($O \Leftarrow OS$) if and only if

$$\forall out \in O.OutputParameters$$
$$\exists x \in \bigcup_{op \in OS} op.OutputParameters:$$
$$x = out \vee x \lhd out$$
$$\wedge$$
$$\forall in \in \bigcup_{op \in OS} op.InputParameters$$
$$\exists x \in ((\bigcup_{o \in OS} o.OutputParameters) \cup O.InputParameters):$$
$$x = in \vee x \lhd in$$

$\circ$

Namely, we say that a set OS of *Operation* elements is functionally equivalent to an *Operation* O if and only if:

- for each O's output parameter *out* there is a (type-compatible) output parameter x which is produced by some service in OS and whose name is (equivalent to or) a sub-concept of $out.name$, and
- for each *Operation* in OS, its input parameter names (are equivalent to or) subsume the names of (type-compatible) parameters that are given in input to O or that are produced as output by some service in OS.

Before describing how to determine the set(s) OS we have to make an observation. Why are we talking about *sets* (and not *sequences*) of *Operation* elements? This is because the solution we are going to introduce will discover all the possible (minimal) sets of *Operation* elements which satisfy the functional equivalence condition. Once those sets are available, obtaining the desired sequence is immediate. Indeed, we only have to start from O's input parameters and add each *Operation* in OS following the hyperedges of the dependency hypergraph.

So, in the following paragraphs we will show how to discover the minimal sets of *Operation* elements[5].

Discovering sets of *Operation* elements

As one may expect, the set discovering consists of a visit of the dependency hypergraph. More precisely, OperationSetsDiscovering (Figure 4.4) visits the dependency hypergraph starting from those vertices corresponding to the parameters (equivalent[6] to those) outputted by O, and it goes on by exploring backwards the hyperedges until reaching (if possible) the input parameters that O requires. As we will see, OperationSetsDiscovering detects all the minimal sets OS of *Operation* elements such that $O \Leftarrow OS$.

OperationSetsDiscovering requires five input parameters: the dependency hypergraph H, the *Operation* to "discover" O, the set *composition* of the operations selected so far (initially empty), the set of the *needed* output parameters to be generated (initially the O's outputs), and the set of the *available* output parameters (initially O's inputs).

If no more output parameters need to be generated ($needed = \varnothing$), OperationSetsDiscovering stores the set *composition* (such that $O \Leftarrow composition$) and the algorithm is stopped (lines 1-4).

Otherwise, the algorithm employs Extract[7] to (non-deterministical-

[5]The algorithm and the relative results are an adaptation (to our problem) of those proposed by Corfini [11].

[6]Please note that, since the *Operation* O comes from a different environment, there could be some mismatchings between the relative ontologies. For the sake of simplicity (and since - according to Assumptions 3.2 and 3.3 - we can check equivalent concepts before starting OperationSetsDiscovering), in the following we do not consider the cross-ontology problem.

[7]It is worth noting that Extract implementation strictly depends on the way in which the set *needed* is implemented. So, for the sake of simplicity, we assume the availability of that procedure.

Require: dependency hypergraph H, *Operation* O, set *composition*, set *needed*, set *available*

1: **if** $needed = \varnothing$ **then**
2: **store** *composition*;
3: **exit**;
4: **end if**
5: $out = \text{EXTRACT}(needed)$;
6: $Ops = \{o \mid \exists c \in o.OutputParameters : c = out \vee c \lhd out)\}$;
7: **if** $Ops = \varnothing$ **then**
8: **fail**;
9: **end if**
10: **for all** *Operation* $op \in Ops$ **do**
11: $composition' = composition \cup \{op\}$;
12: **for all** *Operation* $p \in composition$ **do**
13: $R = composition' \backslash \{p\}$;
14: **if** $\nexists x \in p.OutputParameters$:
$\exists o \in O.OutputParameters$:
$(x \lhd o \wedge \nexists z \in \bigcup_{r \in R} r.OutputParameters : z \lhd o)$
$\vee$
$\exists i \in \bigcup_{r \in R} r.InputParameters$:
$(x \lhd i \wedge \nexists z \in \bigcup_{r \in R} r.OutputParameters \cup O.InputParameters$:
$z \lhd i))$ **then**
15: **fail**
16: **end if**
17: **end for**
18: $available' = available \cup op.OutputParameters$;
19: $needed' = \{x \mid x \in (needed \cup \{y \mid y \in op.InputParameters \wedge \nexists z \in O.InputParameters : z \lhd y\}) \wedge \nexists a \in available' : a \lhd x\}$;
20: OPERATIONSETSDISCOVERING(H,O,*composition'*,*needed'*,*available'*);
21: **end for**

Figure 4.4: OPERATIONSETSDISCOVERING algorithm.

ly) withdraw an output parameter *out* from the set of *needed* outputs (line 5). Once *out* has been taken, OperationSetsDiscovering proceeds in computing the set *Ops* of the operations which produce a type-compatible parameter whose name is a sub-concept of *out.name* (line 6). Afterwards, if *Ops* is empty (i.e., *out* cannot be generated by any available operation), then OperationSetsDiscovering fails (since the operation cannot be matched by any sequence of available *Operation* elements - lines 7-9).

Otherwise, for each operation *op* in *Ops* (line 10), the algorithm adds *op* to *composition* (line 11) and updates the sets *available* and *needed* by adding them the outputs of *op* (line 18) and the unavailable inputs of *op* (line 19), respectively. Then, a new OperationSetsDiscovering instance is started on the computed sets (line 20).

In the next paragraph, we will discuss in detail how to reject (by failing) non-minimal operation sets (lines 12–17).

Minimality of discovered sets

As already anticipated, the role of the loop at lines 12-17 is to discard (by failing) any non-minimal set OS of *Operation* elements such that $O \Leftarrow OS$. Going on in their explanation requires to formalize the obvious notion of minimality.

Definition 4.10. Let O be an *Operation* and OS be a set of *Operation* elements such that $O \Leftarrow OS$. OS is *minimal* if and only if

$$\nexists OS' \subset OS : O \Leftarrow OS'$$

∘

Intuitively speaking, the loop under consideration checks whether the inclusion of the new *Operation op* in the set *composition* makes some other *Operation* elements in *composition* not strictly necessary to obtain $O \Leftarrow$ *composition*.

Observation 4.5. It is worth noting that *op* is certainly needed to satisfy the condition $O \Leftarrow composition$, since the set $composition \backslash \{op\}$ is not able to produce *out* (line 6). Note, indeed, that OperationSetsDiscovering does not consider the operation *op* (line 12).

OperationSetsDiscovering proceeds by checking, for each *Operation p* in *composition* (line 12), whether the condition at line 14 holds. Such condition is `true` if all the output parameters produced by p are already available since:

- they are generated by the other *Operation* elements in $composition \cup \{op\} \backslash \{p\}$, or
- they are provided as input parameters of O.

In other words, if the condition at lines 14 holds, then the inclusion of *op* in the set of operations has made the *Operation p* not strictly necessary to achieve the goal. If this is the case, OperationSetsDiscovering fails (line 15) to avoid constructing non-minimal sets of *Operation* elements.

It is worth noting that, despite the condition under consideration (line 14) is quite verbose, its practical checking consists only of a few trivial operations among small sets of data.

Last (but not least), we have to answer to the following question: are we sure that line 14 condition is both necessary and sufficient to establish the minimality of a set OS of operations (such that $O \Leftarrow OS$)?

Property 4.1. Let O be an *Operation* and let OS be a set of *Operation* elements such that $O \Leftarrow OS$. OS is *minimal* if and only if:

$$\forall p \in OS,\ \exists x \in p.OutputParameters:$$
$$\text{(a) } \exists o \in O.OutputParameters\ :$$
$$(x \lhd o \wedge \nexists z \in \bigcup_{r \in OS \backslash \{p\}} r.OutputParameters:\ z \lhd o)$$
$$\vee$$

(b) $\exists i \in \bigcup_{r \in OS \setminus \{p\}} r.InputParameters$:
$(x \lhd i \wedge$
$\nexists z \in \bigcup_{r \in OS \setminus \{p\}} r\ OutputParameters \cup O.InputParameters$:
$z \lhd i))$

Proof. We have to prove a $\Longleftrightarrow$ relationship. So, we will proceed demonstrating the two directions separately.

($\Longrightarrow$) Let us assume that OS (such that $O \Leftarrow OS$) is *minimal* (see Definition 4.10). This means that the following condition holds.

$\forall p \in OS$:
$\exists o \in O.OutputParameters$:
$\nexists z \in \bigcup_{r \in OS \setminus \{p\}} r.OutputParameters$: $z \lhd o)$
$\vee$
$\exists i \in \bigcup_{r \in OS \setminus \{p\}} r.InputParameters$:
$(\nexists z \in \bigcup_{r \in OS \setminus \{p\}} r\ OutputParameters \cup O.InputParameters$: $z \lhd i))$

Since $O \Leftarrow OS$, applying what Definition 4.9 states, we can derive the desired thesis.

$\forall p \in OS, \exists x \in p.OutputParameters$:
$\exists o \in O.OutputParameters$:
$(x \lhd o \wedge \nexists z \in \bigcup_{r \in OS \setminus \{p\}} r.OutputParameters$: $z \lhd o)$
$\vee$
$\exists i \in \bigcup_{r \in OS \setminus \{p\}} r.InputParameters$:
$(x \lhd i \wedge \nexists z \in \bigcup_{r \in OS \setminus \{p\}} r\ OutputParameters \cup O.InputParameters$:
$z \lhd i))$

($\Longrightarrow$) We will prove the desired property by contradiction. Let us assume that

OS is not minimal. This means that the following condition holds.

$$
\begin{aligned}
&\exists p \in OS : \\
&\quad \forall o \in O.OutputParameters : \\
&\qquad \exists z \in \bigcup_{r \in OS \setminus \{p\}} r.OutputParameters \colon z \lhd o) \\
&\quad \vee \\
&\quad \forall i \in \bigcup_{r \in OS \setminus \{p\}} r.InputParameters : \\
&\qquad (\exists z \in \bigcup_{r \in OS \setminus \{p\}} r\ OutputParameters \cup O.InputParameters : z \lhd i))
\end{aligned}
$$

Now, since

$$\exists p \in OS(\forall a \in A(\exists b \in B : P(p,a,b))) \Longrightarrow \exists p \in OS(\nexists a \in A(\nexists b \in B : P(p,a,b)))$$

we obtain the following condition.

$$
\begin{aligned}
&\exists p \in OS : \\
&\quad \nexists o \in O.OutputParameters : \\
&\qquad \nexists z \in \bigcup_{r \in OS \setminus \{p\}} r.OutputParameters \colon z \lhd o) \\
&\quad \vee \\
&\quad \nexists i \in \bigcup_{r \in OS \setminus \{p\}} r.InputParameters : \\
&\qquad (\nexists z \in \bigcup_{r \in OS \setminus \{p\}} r\ OutputParameters \cup O.InputParameters : z \lhd i))
\end{aligned}
$$

The above condition is clearly in contradiction with the (assumed) right part of the desired property. This implies that OS is minimal. □

Soundness, completeness and complexity

We are going to conclude this Subsection 4.3.2 with a (brief) discussion of the OperationSetsDiscovering soundness, completeness and complexity.

Soundness As mentioned before, each instance of OperationSetsDiscovering stores a set of *Operation* elements which is functionally equivalent ($\Leftarrow$) with the searched *Operation* O. The following proposition establishes the soundness of the proposed algorithm, namely, that each stored set of operations satisfies the functional equivalence condition (see Definition 4.9).

Proposition 4.1. Let OS be a set of *Operation* elements stored by the OperationSetsDiscovering for a given *Operation* O. Then, $O \Leftarrow OS$.

Proof. The proof is organised in three steps. First, we will establish an invariant property Φ which holds for every invocation of OperationSetsDiscovering (1). Then we will prove that Φ implies $O \Leftarrow OS$ when the algorithm terminates (2). Finally, we will demonstrate that OperationSetsDiscovering always terminates (3).

(1) Please note that the set *needed* (which initially contains the desired operation O output parameters) is updated whenever a new *Operation op* is added to the *composition* set. More precisely, OperationSetsDiscovering

- adds to *needed* the unavailable *op.InputParameters*, and
- removes from *needed* those elements with which *op.OutputParameters* are in the $\lhd$ relation.

So, whenever a recursive call of OperationSetsDiscovering is performed, the following invariant property Φ holds.

$$\begin{aligned}\Phi \equiv needed = \\ \{x \mid x \in (O.OutputParameters \cup \\ \{u \mid u \in \bigcup_{p \in OS} p.InputParameters \\ \wedge \nexists v \in O.InputParameters : v \lhd u\}) \\ \wedge \nexists y \in \bigcup_{p \in OS} p.OutputParameters : y \lhd x\}\end{aligned}$$

where OS denotes the set of *Operation* elements selected so far (i.e., the set *composition*).

(2) Now, we know that OperationSetsDiscovering returns the set *composition* (only) when *needed* is empty. But, what does the condition "$needed = \varnothing$" imply? Since Φ holds, it follows that:

$$\begin{aligned}\nexists x :\ & x \in (O.OutputParameters \cup \\ & \{u \mid u \in \bigcup_{p \in OS} p.InputParameters \\ & \wedge\ \nexists v \in O.InputParameters :\ v \lhd u\}) \\ \wedge\ & \nexists y \in \bigcup_{p \in OS} p.OutputParameters :\ y \lhd x\}\end{aligned}$$

Then, applying the following logical rules:

- $\nexists x : (x \in A \cup B \wedge \nexists y \in C : P(x,y)) \Longrightarrow$
  $\forall x : (x \in A \Longrightarrow \exists y \in C : P(x,y)) \wedge$
  $(x \in B \Longrightarrow \exists y \in C : P(x,y)))$;
- $((A \wedge \neg B) \Longrightarrow C) \Longrightarrow (A \Longrightarrow (B \vee C))$,

we could derive the following condition.

$$\begin{aligned}\forall x :& x \in O.OutputParameters \Longrightarrow \\ & \exists y \in \bigcup_{p \in OS} p.OutputParameters :\ y \lhd x \\ \wedge\ & \\ x \in & \bigcup_{p \in OS} p.InputParameters \Longrightarrow \\ & \exists y \in (\bigcup_{p \in OS} p.OutputParameters \cup O.InputParameters) : \\ & y \lhd x\end{aligned}$$

Looking at the above logical expression, we observe that (by Definition 4.9) it follows that $O \Leftarrow OS$. Hence the invariant property Φ guarantees that when $needed = \varnothing$ then *composition* is a set of services that satisfies the functional equivalence condition.

(3) To complete our proof, we have to show that OperationSetsDiscovering always terminates. First, let us remark that, if *Ops* (i.e., the set of those *Operation* elements which produce *out* - or

a parameter c such that $c \lhd out$) is not empty, then none of the operations in such a set is already contained in *composition* (since if it were contained, then *out* would not still belong to *needed*). This implies that each available *Operation* can be inserted in *composition* at most once.

Now, let *candidates* be the set of all *Service Template* operations which are candidate for insertion in *composition*. Since each *Operation* can be inserted in *composition* at most once, the *candidates* set cardinality will be decreased by one at every OperationSetsDiscovering invocation.

Then, the last recursive call of the algorithm under consideration:

- either succeeds (if $needed = \varnothing$), or
- fails (because of the absence of candidate operations).

□

The above introduced Proposition 4.1, along with Property 4.1, let us derive the following result.

Proposition 4.2. Let OS be a set of *Operation* elements returned by OperationSetsDiscovering for a given *Operation* O. Then, OS is minimal and the condition $O \Leftarrow OS$ holds.

Completeness We have shown that each result set stored by OperationSetsDiscovering satisfies the functional equivalence condition. Now we have to answer to the following question: does OperationSetsDiscovering store all the functional equivalent minimal sets? Hereafter we will show that this is the case.

Proposition 4.3. Let $OS = \{O_1, O_2, ..., O_n\}$ be a *minimal* set of *Operation* elements such that $O \Leftarrow OS$. If there exists such a set of operations, then it will be returned by OperationSetsDiscovering.

Proof. Suppose by contradiction that there exists a *minimal* operation set $OS = \{O_1, O_2, ..., O_n\}$ such that $O \Leftarrow OS$. Suppose also that OS is not stored by OperationSetsDiscovering. Is it possible? To answer this question we have to detect why OS can be discarded.

(A) There exists some operation $O_i \in OS$ which is not selected by OperationSetsDiscovering. In other words, none of the recursive calls of the analysed algorithm extracts from the *needed* set a parameter out_i outputted by O_i. This is possible in the following two cases:

- O_i generates no concepts useful to match O. In other words, it produces neither parameters which belongs to $O.$*Output-Parameters*, nor parameters taken as input by another operation in OS. If this is the case, O_i is completely useless to obtain $O \Leftarrow OS$. It follows the desired contradiction: the set OS is not minimal (as assumed).
- O_i generates a parameter c useful (to obtain $O \Leftarrow OS$) but c is also produced by another *Operation* $O_j \in OS$ $(i \neq j)$. Indeed, suppose that (at some step n) OperationSetsDiscovering extracts from the *needed* set a parameter $out \neq c$ which is produced by O_j. This causes the following steps.
  (i) O_j is added to *composition*;
  (ii) the output parameters of P_j are removed from the *needed* set and added to the set of *available* ones.

  It is worth noting that also the parameter c (if present) is removed from *needed* and added to *available*. So, c will never be extracted from the *needed* set (i.e., O_i will never be selected by OperationSetsDiscovering). Consequently, O_i is not strictly necessary to obtain $O \Leftarrow OS$. We earned the desired contradiction: the set OS is not minimal (as assumed).

(B) The set (or a subset of) $OS = \{O_1, O_2, ..., O_n\}$ is neither generated nor stored by OperationSetsDiscovering. Why could this happen? Let us consider an *Operation* $O_i \in OS$ which generates c and a *Operation* $O_j \in OS$ $(O_i \neq O_j)$ which generates out_j

and c. Suppose now that (at some step n) the algorithm extracts c from the *needed* set and selects O_i. The algorithm then goes on and when (at some step $m > n$) out_j is withdrawn from *needed*, OperationSetsDiscovering selects O_j. Consequently, O_i becomes useless, since c is also produced by O_j (Property 4.1 does not hold). Thus, we obtain the desired contradiction (since the set of operations OS is not minimal).

The above consideration let us take the desired conclusion: OperationSetsDiscovering stores *all* the minimal set of *Operation* elements which are functionally equivalent ($\Leftarrow$) to the provided *Operation* O. $\square$

Worst-case (time) complexity Finally, we will outline a worst-case analysis of the algorithm time complexity. The analysis requires to consider a possible execution of OperationSetsDiscovering. So, let OPS be the set of available *Operation* elements.

- The first instance of OperationSetsDiscovering extracts a parameter out_1 from the *needed* set. At most there exist $|OPS|$ operations which output the desired parameter (or a parameter c_1 such that $c_1 \lhd out_1$). Hence, OperationSetsDiscovering splits in $|OPS|$ instances.
- Consider now the i-th instance generated by first step of the algorithm (in which $composition = \{O_i\}$). Such an instance, in turn, withdraws a parameter out_2 from the *needed* parameters set. At most there exists $|OPS|-1$ available operations which produce the desired parameter (or a parameter c_2 such that $c_2 \lhd out_2$). Thus, the instance under consideration splits in $|OPS| - 1$ instances.

Therefore, OperationSetsDiscovering generates

$$|OPS| \times (|OPS| - 1) \times (|OPS| - 2) \times ... \times 2 \times 1$$

instances.

Furthermore, each instance complexity is dominated by the minimality check. This check executes at most $O(|OPS|^2)$ comparisons. Then, each instance of OperationSetsDiscovering costs $O(|OPS|^3)$ (since it executes the minimality check for each operation producing the *out* parameter extracted from the *needed* set).

The above consideration let us conclude that OperationSetsDiscovering requires exponential time.

$$T_{\text{OperationSetDiscovering}}(OPS) \in NP.$$

Practical consideration about complexity

A fussy reader could think our solution is expensive and inefficient.

Please note that the OperationSetsDiscovering must be executed by a service provider. In other words, it will be executed in a cloud environment and so the computational power is - potentially - infinite. This means that a clever implementation of the proposed algorithm could exploit its recursive definition to enforce parallelism.

Furthermore, (in an ideal situation) each instance of OperationSetsDiscovering could be executed by a different concurrent activity. If this is the case, the time complexity will be decreased significantly. Indeed, it can (potentially) become polynomial in the number of available operation OPS.

DIGRESSION: NP-hardness of the considered problem
Someone could think that our solution is too expensive. So, before concluding, we want to show that the problem to be solved is in turn an expensive one. More precisely, we want to discuss the NP-hardness of such a problem. To do it, we will connect it with the *subset sum problem* (which is known to be NP-hard [34]).

For the sake of simplicity let us make the following assumptions:
∘ the hypergraph H is such that $\forall i : |E_i| = 2$ (i.e., H is a graph);
∘ we do not matter about semantics (i.e., we want to find only parameters which have the same type and the same name).

The above assumptions simplify quite much our problem. Now we are considering the situation of finding a sub-graph of the (hyper-)graph H such that its borders are composed by those vertices corresponding to the input/output parameters of O.

Let us now simplify more our problem: we only want to determine the above mentioned borders (i.e., the sets of nodes corresponding to the input/output parameters of O). So, (since we are not worrying about interconnections) we are restricting our search to only the set V of vertices.

Let us now label each node with a different integer number. It follows that the whole set of O input/output parameters could be seen as an integer value *sum* (which is the sum of integer values assigned to each parameter in the O input/output parameters set).

So, our problem now consists of finding (one of) the subsets of V whose value sum is equal to *sum*. This clearly correspond to the *subset sum problem*.

Therefore, *simplifying* the problem under consideration we can connect it with a well known NP-hard problem. This means that the considered problem (of detecting a part of the dependency hypergraph such that required) is in turn NP-hard.

4.3.3 Example

The objective of this subsection is to show how the plan generation problem could be solved. We will proceed as follows: we will start by giving the set of available operation and showing the relative dependency hypergraph; then, we will introduce the target operation (i.e., the operation which requires the plan generation - if possible - to be matched); finally, we will perform a possible execution of the OperationSetsDiscovering algorithm.

The dependency hypergraph

Suppose the availability of a *ServiceTemplate* such that in Figure 4.1. Furthermore, suppose that the *WeatherApp*'s *GetWeather* interface exhibits the operations reported in the following table.

Name	**Input parameters**	**Output parameters**
weatherInfo	{country, city}	{weather, umidity, windSpeed, temperature}
getGMT	∅	{gmt}
getTemperature	{country, city}	{temperature}
perceivedTemperature	{windPower, umidity, temperature}	{perceived}

The above given table, along with the ontology of Figure 4.5, let us build the desired hypergraph. According to what stated at the end of Subsection 4.3.1, we assume the availability of the desired dependency hypergraph (see Figure 4.6).

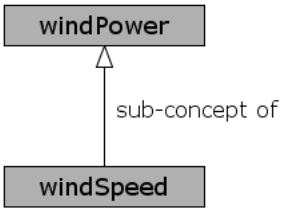


Figure 4.5: Parameters ontology example.

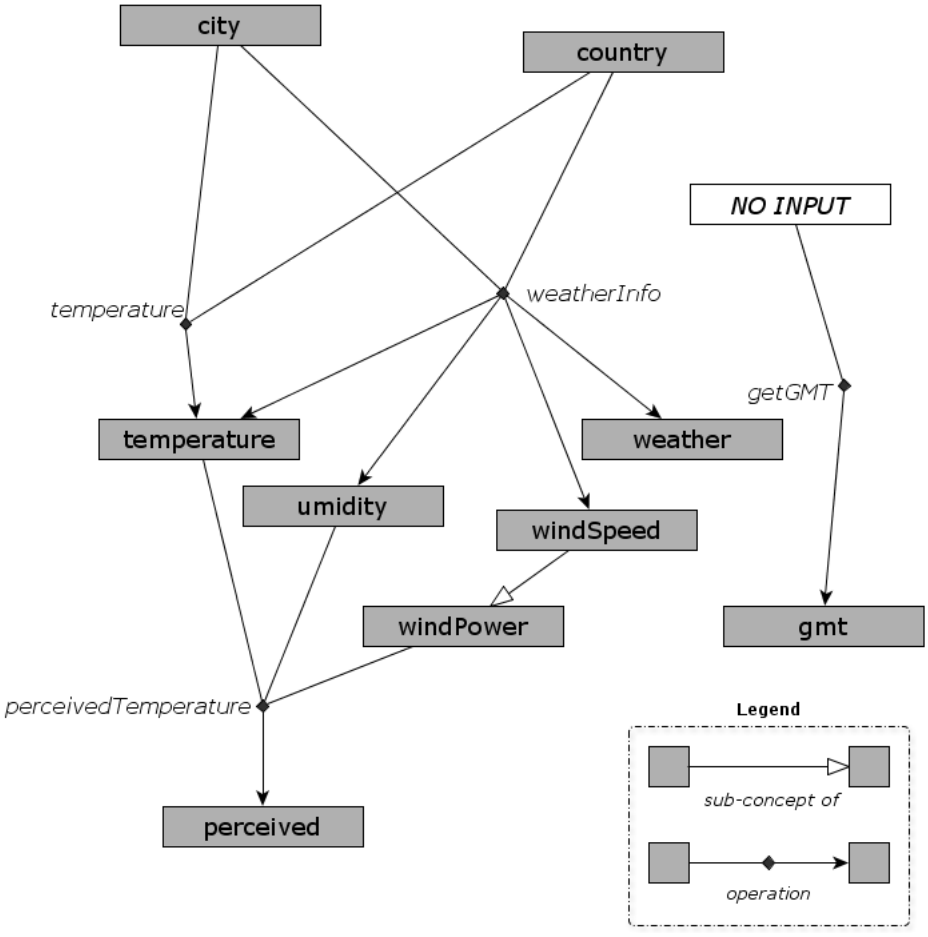


Figure 4.6: Dependency hypergraph example.

The target *Operation*

Consider the situation in which our matching procedure needs the *Operation getPerceivedTemperatureAndTime* which:

- takes {state, city} as input parameters set;
- outputs the {greenwichMeanTime, temperature, perceived} set of parameters.

In the following we will refer to the desired operation as O.

Suppose now the unavailability of O in the *ServiceTemplate* under consideration. This means that we have to check whether the condition

$$\exists\ \textit{Plan}\ p \in OC_{ST}\colon\ x \cong_O (\textit{Operation})\ p.$$

is satisfied. To do it, we need to solve the plan generation problem.

OperationSetsDiscovering simulation

We start observing that some O parameters are not findable in those available with the *ServiceTemplate* under consideration. This is the case of the "state" input parameter and the "greenwichMeanTime" output parameter. It is worth nothing that those parameters could be matched[8] with the available "country" and "gmt" parameter. So, for the sake of simplicity, in the following we will consider an *Operation* O' (which exposes the semantically equivalent available parameters) instead of the real *Operation* O.

Now, let us start the OperationSetsDiscovering algorithm. Its first instance receives the following input (in addition to the dependency hypergraph H and the target operation O'):

[8] With a cross-ontology matching analogue to that exposed in Sections 3.3 and 4.2.

$$composition = \varnothing$$

$$needed = \{\text{gmt, temperature, perceived}\}$$

$$available = \varnothing$$

Clearly, the condition $needed = \varnothing$ is not satisfied. Therefore, the algorithm proceeds in withdrawing one of the *needed* outputs: "gmt", for example. This means that the set *Ops* available operations (whose output is equivalent to - or compatible with - "gmt") is composed only by the *getGMT* operation. So, OperationSetsDiscovering adds the first *Operation* to *composition* and updates the *needed* and *available* sets (removing "gmt" and adding "gmt", respectively). Once these operations have been performed, the discovering procedure goes on with a new instance.

The second instance of OperationSetsDiscovering receives the following input (along with the hypergraph H and the *Operation* O'):

$$composition = \{getGMT\}$$

$$needed = \{\text{temperature, perceived}\}$$

$$available = \{\text{gmt}\}$$

As before, the set of *needed* output parameters is not empty. So, the algorithm can withdraw another parameter from such set: suppose (for example) that the parameter "perceived" is taken. As it was for the previous instance, there is only one operation that generates an equivalent or compatible parameter: *perceivedTemperature*. Therefore that *Operation* is added to the *composition* set. Differently from before, (before adding the new operation) the *composition* set is not empty. Therefore, it must be runt the usefulness check on that operation (*getGMT*) which it is contained in *composition*. Obviously, this usefulness condition is not satisfied. So, the procedure can go on updating the *needed* and *available* sets. More precisely:

- "perceived" is removed from the *needed* set and the unavailable input parameters of *perceivedTemperature* are added to the same set;
- "perceived" is added to the *available* parameters set.

Then, a new instance with the computed sets is started.

The third instance of the algorithm under consideration receives the following input (in addition to H - the dependency hypergraph - and O' - the target operation):

$$composition = \{getGMT, perceivedTemperature\}$$

$$needed = \{\text{temperature, windPower, umidity}\}$$

$$available = \{\text{gmt, perceived}\}$$

Since $needed \neq \varnothing$, OperationSetsDiscovering withdraws a parameter from that set. Suppose the extracted parameter is "temperature". Differently from before, two different operations generate the desired parameter. So, we need to consider them both.

weatherInfo This operation is added to *composition* and the usefulness of the already contained operations is checked. Since all of them are useful, the execution proceeds with the usual behaviour: *needed* and *available* are updated[9] and a new instance A (whose input is composed by the computed sets) is runt.

getTemperature Similarly as before, the sets are updated and a new instance B is executed.

[9]It is worth noting what happens to the "windPower" *needed* parameter. Since *weatherInfo* generates a parameter "windSpeed" which is in the *sub-concept* hypergraph relation, it is removed from the *needed* set (and "windSpeed" is added to the *available* one).

Consider the instance A. Its input (in addition to the dependency hypergraph H and the target *Operation* O') consists of:

$$composition = \{getGMT, perceivedTemperature, weatherInfo\}$$

$$needed = \varnothing$$

$$available = \{\text{gmt, perceived, weather, windSpeed, umidity, temperature}\}$$

Differently from previous instances, $needed = \varnothing$. Therefore, the set *composition* is stored (since it is capable to exhibit - more than - the required input/output parameter) and the A instance is stopped.

Consider now the instance B. Its input consist of (the dependency hypergraph H, the operation O' and):

$$composition = \{getGMT, perceivedTemperature, getTemperature\}$$

$$needed = \{\text{windPower, umidity}\}$$

$$available = \{\text{gmt, perceived, temperature}\}$$

As already happened, $needed \neq \varnothing$. This means that another parameter ("windPower", for example) is withdrawn from that set. Then, the set *Ops* is computed and, since it contains only *perceivedTemperature*, that *Operation* is added to *composition*. Once this has been done, the minimality checking is performed: is there any operation which makes the *composition* set not minimal? Please note that the only *getTemperature* output parameter is produced also by the *weatherInfo* operation. So, since the minimality condition is not satisfied, the execution is stopped by failing.

Plan generation

The (possible) execution shown in the previous paragraph has produced the storage of the following set.

$$composition = \{getGMT, perceivedTemperature, weatherInfo\}$$

We are going to show how to exploit this set in plan generation.

The procedure simply consist of answering iteratively to the following question: which operations in *composition* are performable? Let us start with the target operation input parameters. With this set of parameters we can execute the *weatherInfo* and *getGMT*[10] operations. Then, the available parameters set is updated with those produced by the performed operations. Once this has been done, also *perceivedTemperature* is performable.

With the above (simple) procedure we can generate the plan of Figure 4.7.

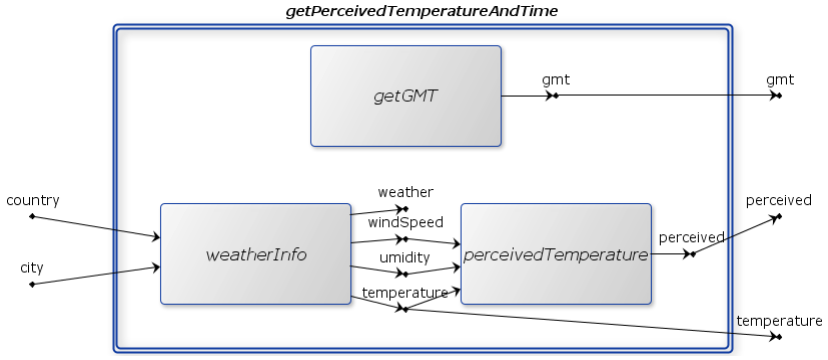


Figure 4.7: Example of generated plan.

[10]This operation does not require any input parameter.

4.4 Concluding remarks

This chapter has provided a solution to the white-box matching problem. This solution completes the discussion about the type-checking (and adaptation) between a *NodeType* N and a *ServiceTemplate* ST. Before going on, it is worth to make some final observations.

4.4.1 A complete example

Consider the problem of matching between the *ServiceTemplate* ST and the *NodeType* $WeatherAppType$ (see Figure 4.8). Suppose that the $GetWeather$ interfaces (inside and on the boundaries of ST) exhibit the same operation as those of the example in Subsection 4.3.3. Furthermore, suppose that the $getPerceivedTemperatureAndTime$ interface (of $WeatherAppType$) declares to export the homonym operation (which is the same as the target operation of the example in Subsection 4.3.3).

If we black-box match the *ServiceTemplate* and the *NodeType* under consideration, then we obtain a fail (since ST does not expose the *Name* property and the desired operation). In other words, accordingly to Figure 3.10, we obtain:

$$unmatchedRequirementSet = \varnothing$$

$$unmatchedCapabilitySet = \varnothing$$

$$unmatchedOperationSet = \{getPerceivedTemperatureAndTime\}$$

$$unmatchedPropertySet = \{Name\}$$

So, we have to proceed with the white-box viewpoint approach explained in this chapter.

First, we have to search for the *Name* property in the nodes inside ST. Our property search, as mentioned in (the relative paragraph of)

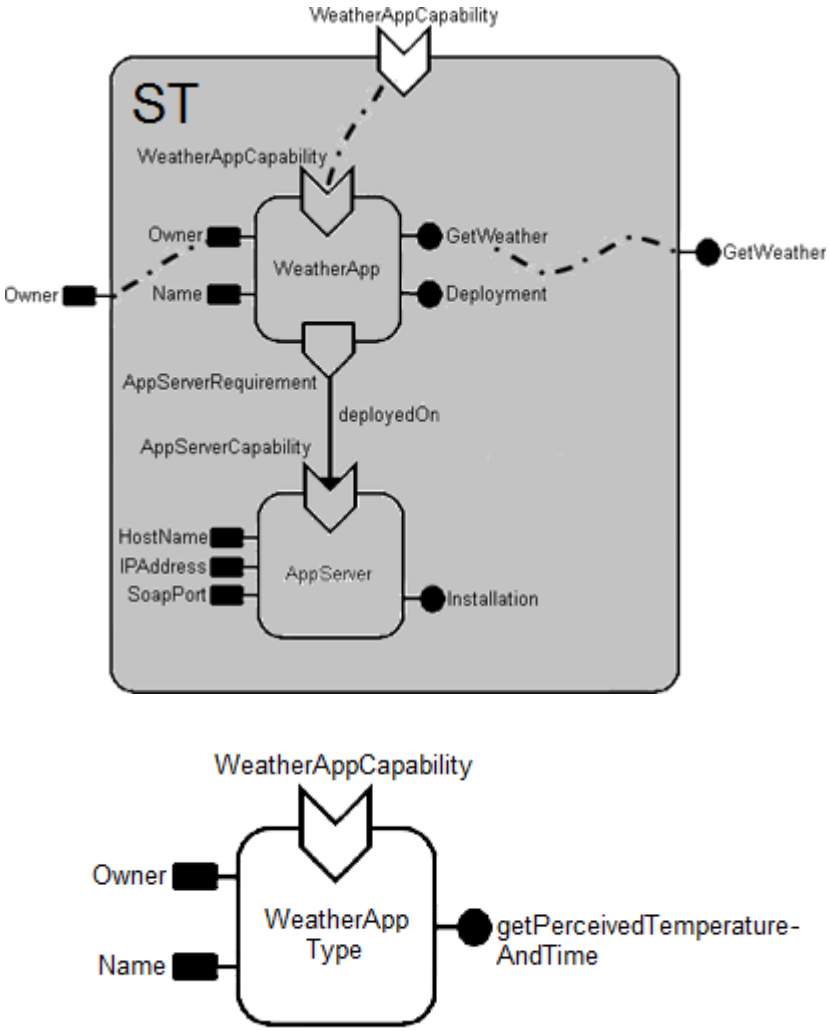


Figure 4.8: A complete matching example.

Subsection 4.2, will not be a signature one. Indeed, we will use the cross-ontology approach to find a property which is semantically equivalent to the required one. So, suppose that the desired *Name* property is semantically equivalent to the available *HostName* property of *AppServer*. Then, we have to modify *ST* boundaries in order to expose also the desired (renamed) property.

Once the property search is done, we have to proceed in looking for the desired operation. Suppose that the condition

$$\forall\ Operation\ x \in N.Interfaces.Interface$$

$$\exists\ Operation\ y \in ST.Interfaces.Interface:$$

$$x \cong_O y$$

does not hold. This means that we have to employ the OperationSets-Discovering algorithm in order to check whether some available operation combinations let us satisfy the desired following condition.

$$\exists\ Plan\ p \in OC_{ST}:$$

$$x \cong_O (Operation)\ p.$$

So, following a behaviour analogue to the one of the example in Subsection 4.3.3, we obtain the desired plan (see Figure 4.7). Now, we have to modify *ST* boundaries in order to exhibit the obtained plan as an operation semantically equivalent to the desired one.

Please note that, once the above modifications has been performed, we obtain a (new) *ServiceTemplate ST″* which is in black-box flexible matching with the desired *WeatherAppType NodeType*. So, we simply have to employ the (extended) oblivion boundaries adaptation approach in order to obtain the *ServiceTemplate ST′* which can be used as a substitute for the desired *NodeType* (see Figure 4.9).

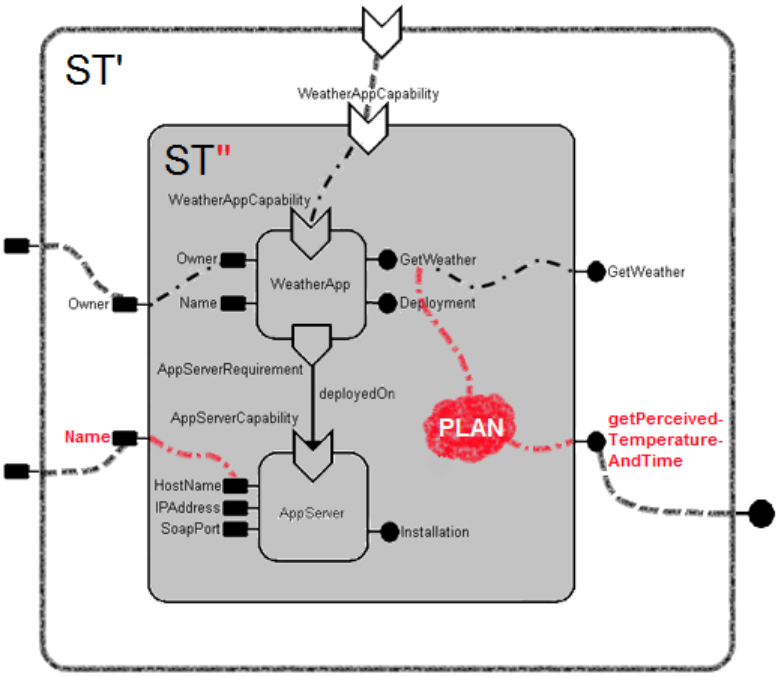


Figure 4.9: Complete matching example adaptation.

4.4.2 Matching and adaptation

As shown in Figure 4.10, all the matching procedure (both the black-box part and the white-box one) is completely automatable. Using this procedure we can generate the desired *ServiceTemplate* ST' (i.e., the one which can be used as a substitute for the needed *NodeType* N) by adapting the available ST. What kind of adaptation is performed?

While in the black-box matching case the adaptation consists (at most) in filtering and renaming the available feature, the white-box matching situation is more complicated. We have seen that the operations to be performed are the following ones: *extracting* (inside) available features and *generating plans* (if needed). So, referring Figure 3.11, the adapter A must implement the above mentioned functionalities.

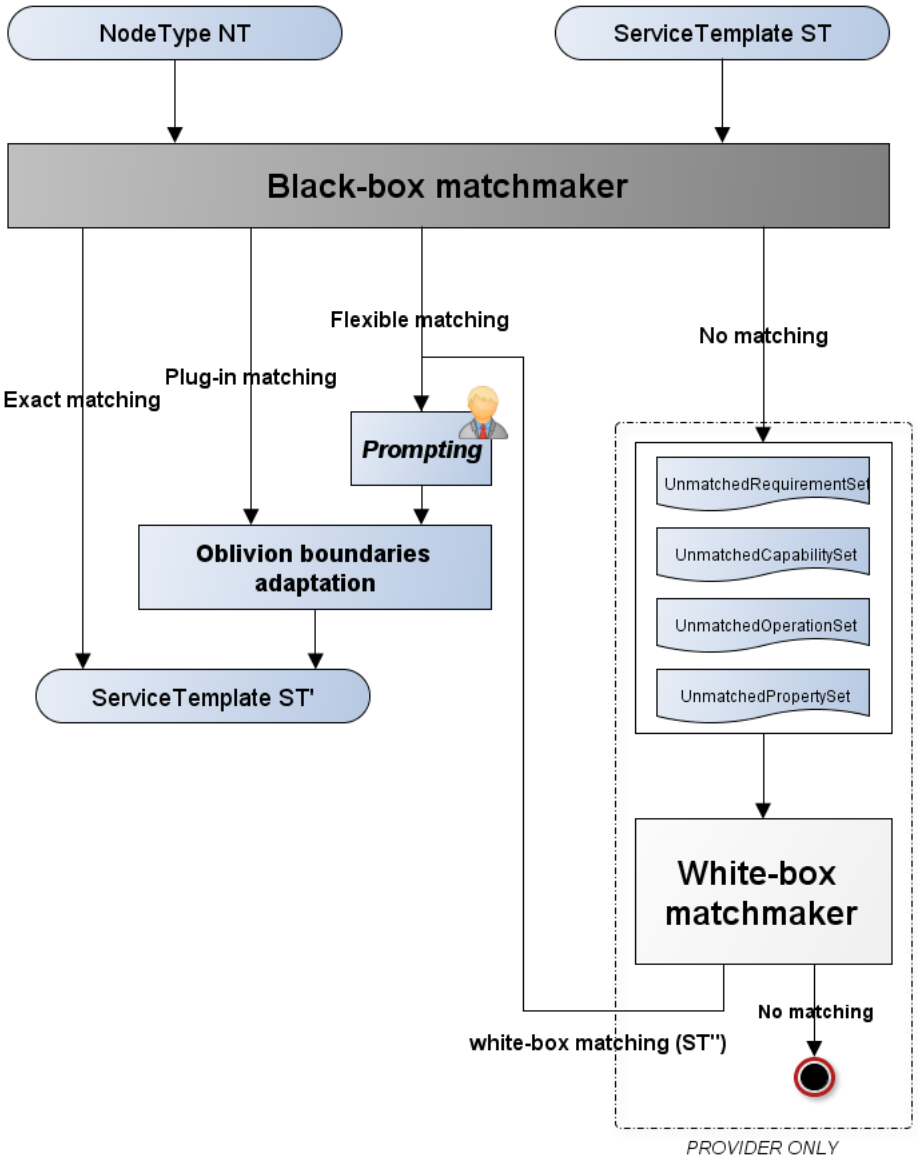


Figure 4.10: Extended matching (and adaptation) procedure.

4.4.3 Taking semantics into account

As for the flexible matching approach, the white-box viewpoint procedure exploits ontologies to obtain the desired matchings. More precisely, ontologies are used to:

- pair (semantically) equivalent features, and
- generate plans (which can be used as substitutes for missing operations - if possible).

Please note that the above operations exactness strictly depends on both the input ontologies and the cross-ontology matchmaker. It follows that, if those elements are not so accurate as required, some generated pairings/plans could not be significant and should not be used in our matching (and adaptation) procedure.

As shown in Figure 4.10, when the considered *NodeType* N white-box matches with the *ServiceTemplate* ST, we have to prompt to the user the white-box matchmaker decisions taken to transform ST in a *ServiceTemplate* ST'' which flexible matches N. If she accepts those decisions (along with those taken with the flexible matching procedure), then we can proceed in developing the adapted *ServiceTemplate* ST' via the *oblivion boundaries* approach.

It is worth noting that the way in which the white-box decision prompting is performed strictly depends on the matchmaker implementation. We recommend to start by submitting to the user the decisions which are most probable and to let the user mark, adjust or discard the wrong decisions.

4.4.4 Practical observations

It is worth making some remarks on the usability of matching methodologies explained in Chapters 3 and 4. The black-box matching procedure employs the available *ServiceTemplate* ST without doing any modification on its specification. This means that the adapter A (see Figure 3.11) could be developed by both the *service client* and the *service provider*. Conversely, the white-box matchmaker (possibly) requires to update ST boundaries and to generate plans of available operations. In other words it needs to work inside of ST (as recalled by the methodology name). So, as reported in Figure 4.10, the only one who can exploit this method to obtain the adapted *ServiceTemplate* ST' is the *service provider*.

Since the white-box matching procedure can only be executed by the *service provider*, the execution environment is a cloud one. This implies that this procedure can (ideally) count on an infinite amount of both computational power and storage space. So, as already mentioned in Subsections 4.3.1 and 4.3.2, this huge amount of resources could be employed to decrease significantly the time complexity of the matching and adaptation procedure (pre-computing and storing the dependency hypergraph and parallelizing the OperationSetsDiscovering implementation).

4.4.5 What's next?

So far, we have explained a way to match *NodeType* and *ServiceTemplate* elements from both a black-box viewpoint (Chapter 3) and a white-box one (Chapter 4).

Please recall that the objective of this thesis is to find a way to check whether available (TOSCA-compliant) services can be used as *NodeTemplate* components of a TOSCA multi-cloud *ServiceTemplate*. So, the presented matchings between node *types* and service *templates* can be seen

as type-matchings. Those checking (and adaptation) procedures can be used only in the first step of the desired complete checking. Which is the second (and last) one? Once the type-checking is satisfied, we have to compare values in order to determine if they are compatible. Then, as we will state in Chapter 6, a way to obtain the desired value comparison between the considered TOSCA elements must be individuated.

Chapter 5

Proof-of-concept implementation

As mentioned in Chapter 1, the different types of matching defined in this thesis should be used to develop a matchmaker to be fruitfully integrated in the TOSCA implementations that are currently under development. In order to enforce this observation, this Chapter 5 aims at showing the feasibility of such (pluggable) matchmaker.

This chapter will start by showing a possible implementation of the TOSCA basic features (viz., capabilities, requirements, policies, properties and interface operations - Section 5.1). Then, in Section 5.2, such features are opportunely grouped to let the needed TOSCA elements be implemented (viz., *NodeType* and *ServiceTemplate* elements). Afterwards, with Section 5.3, a possible implementation of the exact and plug-in matchmakers is given. Finally, Section 5.4 provides some concluding remarks.

All the following listings contain JAVA source code. The reason why we select JAVA as source code language resides in two main factors:

- the TOSCA specification implements inheritance in a way strictly

similar to that of object-oriented languages, and

- the only TOSCA implementation currently available (see [33]) is written in JAVA.

5.1 Implementation of basic features

This section aims at showing how the TOSCA basic features can be implemented. So, in the following paragraphs we will show a possible realization of capabilities, requirements, policies, properties and interface operations.

Implementation of capabilities

We know that each component capability is identified via its name and its type. As we observe in Listing 5.1, how to name the desired capability is not a problem: we simply have to equip the capability class with a `name` field. But, how can we deal with the typing problem? It is worth pointing out that, since we define the abstract `CapabilityType` class (which represents the generic characteristics of a capability type), we can make each desired capability type correspond to a class derived from the `CapabilityType` one. So, the capability typing comes naturally with the class instantiation.

```
1  public abstract class CapabilityType {
2      protected String name;
3
4      public CapabilityType(String name) {
5          this.name = name;
6      }
7
8      public String getName(){
9          return name;
10     }
```

```
11
12      public boolean exactMatch(CapabilityType c){
13          return (name.equals(c.getName()) &&
14                  this.getClass()==c.getClass());
15      }
16
17      public boolean plugInMatch(CapabilityType c) {
18          return (c.getClass().isInstance(this));
19      }
20  }
```

Listing 5.1: JAVA implementation of a generic capability (type).

Please note that the provided `CapabilityType` class is equipped with two non-obvious methods: `exactMatch` (lines 12-15) and `plugInMatch` (lines 17-19). Such methods are used in the similar-named matching procedures (since they return a `boolean` which represents whether the current capability type and the passed one satisfy the exact or plug-in[1] matching condition, respectively).

Implementation of requirements

As we already mentioned in Chapter 2, there is a high similarity between capability and requirement structures. So, the requirement JAVA implementation is analogous to the capability one (see Listing 5.2).

```
1  public abstract class RequirementType {
2      protected String name;
3
4      public RequirementType(String name){
5          this.name = name;
6      }
```

[1]Please remember that with the presented implementation the type of a capability corresponds to the instantiated class. So, what we have to do (to check whether the current capability is of a type derived from that of the passed one) is to check whether the current object can be considered an instance of the passed object class.

```
7
8       public String getName(){
9           return name;
10      }
11
12      public boolean exactMatch(RequirementType r){
13          return (name.equals(r.getName()) &&
14                  this.getClass()==r.getClass());
15      }
16
17      public boolean plugInMatch(RequirementType r) {
18          return (r.getClass().isInstance(this));
19      }
20  }
```

Listing 5.2: JAVA implementation of a generic requirement (type).

Implementation of policies

Please recall that (as discussed in Chapter 2) we identify each policy via its own type. As for requirements and capabilities, we decide to implement such a policy typing via the definition of a generic `PolicyType` (Listing 5.3). Such a generic element is an abstract class which contains all the features common to all the definable policy types (viz., the set of nodes to which it is applicable and some management methods).

```
1  public abstract class PolicyType {
2      protected ArrayList<NodeType> applicabilityDomain;
3
4      public PolicyType(){
5          applicabilityDomain = new ArrayList<NodeType>();
6      }
7
8      public PolicyType(ArrayList<NodeType>
9                        applicabilityDomain){
10         this.applicabilityDomain = applicabilityDomain;
11     }
```

```
12
13      public boolean isApplicableTo(NodeType n){
14          if(applicabilityDomain.isEmpty())
15              return true;
16          for(int i=0; i<applicabilityDomain.size(); i++)
17              if(applicabilityDomain.get(i).getClass()
18                  .isInstance(n))
19                   return true;
20          return false;
21      }
22  }
```

Listing 5.3: JAVA implementation of a generic policy (type).

Please focus on method `isApplicableTo` (lines 13-21). As anticipated by its name, such method is used to check whether the current policy is applicable to the argument node `n` (of a certain `NodeType` - see Subsection 5.2). The applicability checking consist of verifying whether:

- the current policy type is applicable to every node type (lines 14-15), or
- the (current policy type) set of applicable node types contains a *NodeType* N equal to `n`'s type (or from which it is derived - lines 16-19).

So, we can represent each TOSCA *PolicyType* with a class derived from the `PolicyType` one. Now, since policies matching is quite different from other feature matching, we decided to represent the whole policy set (of a node/service template) with a separated `Policies` class (Listing 5.4).

```
1  public class Policies {
2      protected ArrayList<PolicyType> policies;
3
4      public Policies(){
5          policies = new ArrayList<PolicyType>();
```

```
6       }
7
8       public ArrayList<PolicyType> getPolicies(){
9           return policies;
10      }
11
12      public void addPolicy(PolicyType p) {
13          policies.add(p);
14      }
15
16      public boolean areApplicableTo(NodeType n){
17          for(int i=0; i<policies.size(); i++)
18              if(!policies.get(i).isApplicableTo(n))
19                  return false;
20          return true;
21      }
22  }
```

Listing 5.4: JAVA implementation of a set of policies.

The reason why we decided to separate such policy set implementation mainly resides in the possibility of defining the `areApplicableTo` method. This method implements the operator $\equiv_{PO}$ (which checks whether the current set of policies is applicable to a certain *NodeType* - see Section 3.1).

Implementation of properties

We know that each component property can be seen as a pair

$$\langle \texttt{name}, \texttt{value} \rangle.$$

It is worth noting that, according to our type-checking purposes, taking care of the `value` type is quite important. So, as done before, such a type is related to a JAVA class. Please observe that, while for capabilities, requirements and properties we needed to define a generic class (viz.,

type), here we already have it. Indeed, since we are considering "real" values, we need to employ the (generic) `Object` class.

The above reported observations let us implement a TOSCA property as shown in Listing 5.5.

```
1  public class Property {
2      protected String name;
3      protected Object value;
4
5      public Property(String name, Object value) {
6          this.name = name;
7          this.value = value;
8      }
9
10     public String getName() {
11         return name;
12     }
13
14     public Object getValue() {
15         return value;
16     }
17
18     public boolean exactMatch(Property p) {
19         return (name.equals(p.getName()) &&
20                 value.getClass()==p.getValue().getClass());
21     }
22
23     public boolean plugInMatch(Property p) {
24         return (name.equals(p.getName()) &&
25                 p.getValue().getClass().isInstance(value));
26     }
27 }
```

Listing 5.5: JAVA implementation of a property.

Please note that (as was for `CapabilityType` and `RequirementType` classes) the `Property` class is equipped with the methods `exactMatch` and `plugInMatch`. As before, such method returns a `boolean` which

represents whether the current property and the passed one satisfy the exact and plug-in[2] (single property) matching condition, respectively.

Implementation of interface operations

TOSCA component interfaces implementation requires the realization of three distinct elements: *parameters*, *operations* and *interfaces*. In the following, we will use a bottom-up to show such implementations.

The first element to be modeled is the operation *parameter*. According to the TOSCA specification [25], we implement such a parameter as a triple

$$\langle \texttt{name}, \texttt{value}, \texttt{required} \rangle$$

where `name` is the parameter name, `value` is its value and `required` is a truth value which indicates whether the user must specify the parameter value. As before, our type-checking approach requires to take care of `value`'s type. So, we make it correspond to the class of the `value` object.

The above stated consideration let us implement the operation parameters via the `OperationParameter` class (Listing 5.6).

```
1  public class OperationParameter {
2      private String name;
3      private Object value;
4      private boolean required;
5
6      public OperationParameter(String name, Object value,
7                                boolean required){
8          this.name=name;
9          this.value=value;
10         this.required=required;
```

[2]Please recall that the property `value` type corresponds to a JAVA class. So, to obtain the plug-in matching we need to ensure that the current property `value` is an instance of the passed property `value` class.

```
11      }
12
13      public String getName(){
14          return name;
15      }
16
17      public Object getValue(){
18          return value;
19      }
20
21      public boolean isRequired(){
22          return required;
23      }
24
25      public boolean exactMatch(OperationParameter p){
26          return (name.equals(p.getName())
27                  &&
28                  value.getClass()==p.getValue().getClass()
29                  &&
30                  required==p.isRequired());
31
32      }
33  }
```

Listing 5.6: JAVA implementation of an operation parameter.

It is worth noting that, despite we perform both exact and plug-in matching, operation parameters only require to implement the `exactMatch` operation (which checks whether the current parameter and the passed one satisfy the exact matching condition). This is because, looking at the definitions in Subsections 3.1 and 3.2, we observe that both the two matching levels employ the exact matching between parameters.

Once parameters have been implemented, we can move on with *operation* implementation. Intuitively speaking, each operation can be viewed as a triple

$$\langle \texttt{name}, \texttt{inputParameters}, \texttt{outputParamters} \rangle$$

where `name` is the operation name, `inputParameters` and `outputParameters` are the set of input and output parameters, respectively. So, we can model such operation with the class of Listing 5.7.

```
1  public class Operation {
2      private String name;
3      private ArrayList<OperationParameter> inputParameters;
4      private ArrayList<OperationParameter> outputParameters;
5
6      public Operation(String name,
7            ArrayList<OperationParameter> inputParameters,
8            ArrayList<OperationParameter> outputParameters){
9          this.name = name;
10         this.inputParameters = inputParameters;
11         this.outputParameters = outputParameters;
12     }
13
14     public String getName(){
15         return name;
16     }
17
18     public ArrayList<OperationParameter>
19            getInputParameters(){
20         return inputParameters;
21     }
22
23
24     public ArrayList<OperationParameter>
25            getOutputParameters(){
26         return outputParameters;
27     }
28
29
30     public boolean exactMatch(Operation op){
31         //operations names checking
32         if(!name.equals(op.getName()))
33             return false;
34
35         //input parameters (1-to-1) checking
```

```
36          int nInputs = inputParameters.size();
37          if(nInputs!=op.getInputParameters().size())
38              return false;
39          boolean matched;
40          OperationParameter p;
41          for(int i=0; i<nInputs; i++){
42              p=inputParameters.get(i);
43              matched = false;
44              for(int j=0; j<nInputs && !matched; j++){
45                  if(p.exactMatch(op.getInputParameters()
46                                  .get(j)))
47                      matched = true;
48              }
49              if(!matched)
50                  return false;
51          }
52
53          //output parameters (1-to-1) checking
54          int nOutputs = outputParameters.size();
55          if(nOutputs!=op.getOutputParameters().size())
56              return false;
57          for(int i=0; i<nOutputs; i++){
58              p=outputParameters.get(i);
59              matched = false;
60              for(int j=0; j<nOutputs && !matched; j++){
61                  if(p.exactMatch(op.getOutputParameters()
62                                  .get(j)))
63                      matched = true;
64              }
65              if(!matched)
66                  return false;
67          }
68
69          return true;
70      }
71  }
```

Listing 5.7: JAVA implementation of an operation parameter.

As for parameters, both the exact and plug-in matching employ the operations exact matching. So, only the `exactMatch` method is developed. Let us now focus on such a method. It starts by checking whether the two operations are same-named (lines 31-33). Afterwards, it proceeds by looking for a one-to-one correspondence between the two operations input parameters (lines 35-51) and between their output parameters (lines 53-67). If none of the previous matchings fail, then it returns `true` (since the two operations can be considered exactly matched - line 69).

Let us now consider the *interface* implementation. As mentioned in Chapter 2, each interface can be viewed as a pair

$$\langle \texttt{name}, \texttt{operations} \rangle$$

where `name` is the interface name and `operations` is the set of interface operations. This let us implement such a TOSCA basic feature as reported in Listing 5.8.

```
1  public class Interface {
2      private String name;
3      private ArrayList<Operation> operations;
4
5      public Interface(String name,
6                       ArrayList<Operation> operations) {
7          this.name = name;
8          this.operations = operations;
9      }
10
11     public String getName(){
12         return name;
13     }
14
15     public ArrayList<Operation> getOperations() {
16         return operations;
17     }
18
19     public boolean exactMatch(Interface interf) {
```

```
20          Operation op;
21          boolean matched;
22
23          //interfaces names checking
24          if(!name.equals(interf.getName()))
25              return false;
26
27          //interfaces operations checking
28          if(operations.size()!=interf.getOperations()
29                                  .size())
30              return false;
31          for(int i=0; i<operations.size(); i++) {
32              op = operations.get(i);
33              matched = false;
34              for(int j=0; j<interf.getOperations().size() &&
35                           !matched; j++){
36                  if(op.exactMatch(interf.getOperations()
37                                      .get(j)))
38                      matched = true;
39              }
40              if(!matched)
41                  return false;
42          }
43          return true;
44      }
45
46      public Interface plugInMatch(TOSCAComponent comp) {
47          Operation op;
48          Interface compInterf;
49          ArrayList<Operation> resultIntOps =
50              new ArrayList<Operation>();
51          boolean matched;
52
53          //for each operation op of the current interface
54          for(int oi=0; oi<operations.size(); oi++) {
55              op = operations.get(oi);
56              matched = false;
57              //there must exist an interface of comp
```

```
58              //which contains the desired operation op
59              for(int j=0; j<comp.getInterfaces().size() &&
60                           !matched; j++){
61                  compInterf = comp.getInterfaces().get(j);
62                  for(int oj=0; oj<compInterf.getOperations()
63                                .size(); oj++){
64                      if(op.exactMatch(compInterf
65                         .getOperations().get(oj))) {
66                          resultIntOps.add(compInterf
67                                           .getOperations()
68                                           .get(oj));
69                          matched = true;
70                      }
71                  }
72              }
73              if(!matched)
74                  return null;
75          }
76
77          //returns the interface which contains comp
78          //operations in matching with those of the
79          //current interface
80          return new Interface(name, resultIntOps);
81      }
82  }
```

Listing 5.8: JAVA implementation of an interface.

As done for previous elements, `Interface` is equipped with two methods (`exactMatch` and `plugInMatch`) which let the user check whether the current interface exactly matches the passed one and whether the current interface plug-in matches the interfaces of the passed `TOSCAComponent`[3], respectively. It is worth noting that, since the matching is performed in order to develop an adapted service template, the matched operations of the passed component are grouped in order to obtain a new interface which exactly matches the current one. Such a new interface is then

[3]See Section 5.2.

returned[4].

5.2 Implementation of the needed TOSCA components

Once the basic TOSCA features (viz., capabilities, requirements, policies, properties and interface operations) have been developed, we can proceed in implementing the TOSCA *NodeType* and *ServiceTemplate* elements. So, the following paragraphs aim at illustrating (a sketch of) how to develop in JAVA such TOSCA elements.

Please remember that we are going to implement a sketch of the matching procedure proposed in this thesis (in order to show its - JAVA - feasibility). More precisely, what we are going to do is to implement the black-box exact and plug-in matching notions. So, to our purposes, restricting the view of a TOSCA service component to only what it exposes is enough[5].

Abstract `TOSCAComponent` class

The above simple observation let us model TOSCA service components as classes derived from the abstract `TOSCAComponent` one (Listing 5.9).

```
1 public abstract class TOSCAComponent {
2     protected ArrayList<CapabilityType> capabilities;
3     protected ArrayList<RequirementType> requirements;
4     protected ArrayList<Property> properties;
5     protected ArrayList<Interface> interfaces;
```

[4] If the matching fails, then a `null` interface is returned.

[5] Furthermore, assuming to look at TOSCA service component from a black-box viewpoint is not so restricting as someone could think. All of the TOSCA IDE will probably let users look at service components from both the black-box and white-box viewpoint.

```
6
7       public TOSCAComponent(){
8           capabilities = new ArrayList<CapabilityType>();
9           requirements = new ArrayList<RequirementType>();
10          properties = new ArrayList<Property>();
11          interfaces = new ArrayList<Interface>();
12      }
13
14      public ArrayList<CapabilityType> getCapabilities(){
15          return capabilities;
16      }
17
18      public ArrayList<RequirementType> getRequirements(){
19          return requirements;
20      }
21
22      public ArrayList<Property> getProperties(){
23          return properties;
24      }
25
26      public ArrayList<Interface> getInterfaces(){
27          return interfaces;
28      }
29  }
```

Listing 5.9: JAVA implementation of a generic service component.

Intuitively speaking, the above reported JAVA class let us model the elements common to all the TOSCA service components. Furthermore, this let us use such an abstract class as a generic type to be passed to such methods which are for instance applicable to both *NodeType* and *ServiceTemplate* elements (e.g., the `plugInMatch` method of the `Interface` class).

NodeType implementation sketch

To our type-checking purposes, a generic *NodeType* can be viewed as a quadruple

⟨`capabilities`,`requirements`,`properties`,`interfaces`⟩.

Please note that all the needed features are already contained in the `TOSCAComponent` class. So, we can develop the generic *NodeType* simply extending the above mentioned class (Listing 5.10).

```
1 public abstract class NodeType extends TOSCAComponent{
2     public NodeType(){
3         super();
4     }
5 }
```

Listing 5.10: JAVA implementation of a generic *NodeType*.

Once the generic (viz., abstract) `NodeType` has been developed, our matchmaker user simply needs to declare each desired node type as derived from it.

ServiceTemplate implementation sketch

Differently from *NodeType* elements, a *ServiceTemplate* could also expose policies. So, it extends the generic `TOSCAComponent` adding such a field (with its own management methods - Listing 5.11).

```
1 public abstract class ServiceTemplate
2                       extends TOSCAComponent {
3     protected Policies policies;
4
5     public ServiceTemplate(){
6         super();
7         policies = new Policies();
8     }
```

```
9
10      public Policies getPolicies(){
11          return policies;
12      }
13  }
```

Listing 5.11: JAVA implementation of a generic *Service*.

As was for `NodeType` elements, our matchmaker user simply needs to declare each desired service template as derived from the generic (viz, abstract) `ServiceTemplate`.

It is worth noting that, while both `NodeType` and `ServiceTemplate` can be used as substitutes for the generic `TOSCAComponent`, thanks to JAVA inheritance they are two distinct elements. Such a distinction is made in order to not break the rules of the TOSCA specification [25].

5.3 Implementation of the matchmakers

Sections 5.1 and 5.2 have provided all the ground needed to implement the exact and plug-in matchmakers. To develop such matchmakers, we will follow the same extensive approach used in Chapter 3. In other words, we will develop the desired matchmakers following the class hierarchy of Figure 5.1.

Abstract `Matchmaker` class

Before giving the employable matchmakers, we provide an abstract `Matchmaker` (Listing 5.12) which defines the common characteristics of each deployed matchmaker.

```
1  public abstract class Matchmaker {
2      //TOSCA elements to be matched
3      protected NodeType n;
```

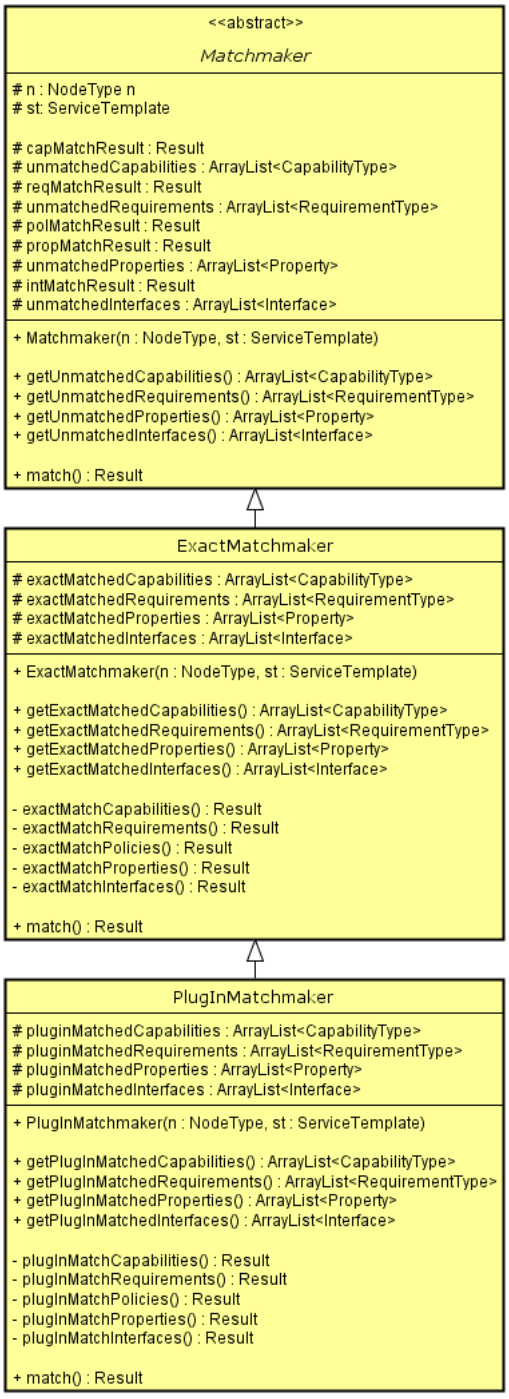


Figure 5.1: UML diagram of the developed matchmakers.

```
4     protected ServiceTemplate st;
5
6     //variables needed to perform matching of capabilities
7     protected Result capMatchResult;
8     protected ArrayList<CapabilityType>
9                         unmatchedCapabilities;
10    //variables needed to perform matching of requirements
11    protected Result reqMatchResult;
12    protected ArrayList<RequirementType>
13                        unmatchedRequirements;
14    //variables needed to perform matching of policies
15    protected Result polMatchResult;
16    //variables needed to perform matching of properties
17    protected Result propMatchResult;
18    protected ArrayList<Property> unmatchedProperties;
19    //variables needed to perform matching of interfaces
20    protected Result intMatchResult;
21    protected ArrayList<Interface> unmatchedInterfaces;
22
23    public enum Result {
24        EXACT,
25        PLUGIN,
26        NOMATCH
27    };
28
29    public Matchmaker(NodeType n, ServiceTemplate st) {
30        this.n = n;
31        this.st = st;
32
33        //matching variables are initialized to null
34        capMatchResult = null;
35        unmatchedCapabilities = null;
36        reqMatchResult = null;
37        unmatchedRequirements = null;
38        polMatchResult = null;
39        propMatchResult = null;
40        unmatchedProperties = null;
41        intMatchResult = null;
```

```
42          unmatchedInterfaces = null;
43      }
44
45      public ArrayList<CapabilityType>
46                      getUnmatchedCapabilities(){
47          return unmatchedCapabilities;
48      }
49
50      public ArrayList<RequirementType>
51                      getUnmatchedRequirements(){
52          return unmatchedRequirements;
53      }
54
55      public ArrayList<Property> getUnmatchedProperties(){
56          return unmatchedProperties;
57      }
58
59      public ArrayList<Interface> getUnmatchedInterfaces(){
60          return unmatchedInterfaces;
61      }
62
63      public abstract Result match();
64  }
```

Listing 5.12: JAVA implementation of the `abstract Matchmaker`.

Please note that for each basic feature (to be matched) we define two fields: *result* and *unmatcheds*[6]. Intuitively speaking, such fields will be used in the derived classes to indicate which kind of matching has been obtained (viz., *exact*, *plug-in* or *unmatched*) and (in case of *unmatched*) which features are left unmatched.

Finally, it is worth noting that the `match` method is declared `abstract`. So, (thanks to JAVA inheritance) the derived classes must declare a method which overrides it.

[6]Since policies matching only consists in checking whether they are applicable to `n`, only the *result* field is defined.

Implementation of the *exact* matchmaker

The first matchmaker to implement is the *exact* one (viz., the "≡" of Section 3.1). The source code[7] of such a matchmaker is reported in Listings 5.13 and 5.14.

```
1  public class ExactMatchmaker extends Matchmaker {
2      //needed to perform exact matching of capabilities
3      protected ArrayList<CapabilityType>
4                          exactMatchedCapabilities;
5      //needed to perform exact matching of requirements
6      protected ArrayList<RequirementType>
7                          exactMatchedRequirements;
8      //needed to perform exact matching of properties
9      protected ArrayList<Property> exactMatchedProperties;
10     //needed to perform exact matching of interfaces
11     protected ArrayList<Interface> exactMatchedInterfaces;
12
13     public ExactMatchmaker(NodeType n,
14                            ServiceTemplate st) {
15         super(n,st);
16
17         //matching variables are initialized to null
18         exactMatchedCapabilities = null;
19         exactMatchedRequirements = null;
20         exactMatchedProperties = null;
21         exactMatchedInterfaces = null;
22     }
23
24     public ArrayList<CapabilityType>
25                     getExactMatchedCapabilities(){
26         return exactMatchedCapabilities;
27     }
28
29     public ArrayList<RequirementType>
30                     getExactMatchedRequirements(){
31         return exactMatchedRequirements;
```

[7] To better understand the unexplained source code please look at *inline comments*.

```
32      }
33
34      public ArrayList<Property>
35                      getExactMatchedProperties(){
36          return exactMatchedProperties;
37      }
38
39      public ArrayList<Interface>
40                      getExactMatchedInterfaces(){
41          return exactMatchedInterfaces;
42      }
43
44      private Result exactMatchCapabilities(){
45          exactMatchedCapabilities =
46                              new ArrayList<CapabilityType>();
47          unmatchedCapabilities =
48                              new ArrayList<CapabilityType>();
49
50          //takes the sets of capabilities
51          ArrayList<CapabilityType> nCaps =
52                                      n.getCapabilities();
53          ArrayList<CapabilityType> stCaps =
54                                      st.getCapabilities();
55
56          //checks whether such sets exactly match
57          CapabilityType cap;
58          for(int i=0; i<nCaps.size(); i++){
59              cap = nCaps.get(i);
60              boolean matched = false;
61              for(int j=0; j<stCaps.size() && !matched; j++){
62                  if(cap.exactMatch(stCaps.get(j))){
63                      exactMatchedCapabilities.add(stCaps
64                                                  .get(j));
65                      matched = true;
66                  }
67              }
68              if(!matched)
69                  unmatchedCapabilities.add(cap);
```

```
70          }
71
72          //if ST and N expose the same number of
73          //capabilities and there are no unmatched
74          //capabilities, then the result is an
75          //"exact" match.
76          if(nCaps.size()==stCaps.size() &&
77             unmatchedCapabilities.isEmpty()){
78              return Result.EXACT;
79          }
80          //otherwise there is "no-match"
81          return Result.NOMATCH;
82      }
```

Listing 5.13: JAVA implementation of the `ExactMatchmaker` (1).

Let us focus on the `exactMatchCapabilities` method (which returns a `Result` to indicate whether the capabilities of *NodeType* `n` and those of the *ServiceTemplate* `st` exactly match). We know that, intuitively speaking,

$$\texttt{n}.\mathit{CapabilityDefinitions} \equiv_C \texttt{st}.\mathit{Capabilities}$$

if and only if the capabilities of `n` are in a one-to-one correspondence with those of `st`. To obtain such a one-to-one correspondence we simply have to check whether

- for each capability `cap` of `n` there exists a capability of `st` which is in `exactMatching` with `cap` (lines 56-70), and
- `n` exposes the same number of capabilities as `st` does (lines 76-77).

If this is the case, then we obtain that `n` *exactly* matches `st` (line 79). Otherwise, we obtain the *unmatched* case (line 81).

It is worth noting that, if a capability is matched, then the relative capability of `st` is stored in the `exactMatchedCapabilities` set (in order

to ease `st` adaptation - lines 62-64). Otherwise, the unmatched capability `cap` of `n` is stored in the `unmatchedCapabilities` set (lines 68-79).

Please note that, as reported in Listing 5.14, the requirements, properties and interfaces matching (lines 84-121, 129-164 and 166-201, respectively) is analogous to the one above presented. This is clearly not the case of policies: we know that, intuitively speaking,

$$PolicyType \text{ applicable to } \texttt{n} \equiv_{PO} \texttt{st}.Policies$$

consists in checking whether `st`'s policies are of a type applicable to `n` (lines 123-127).

```
84      private Result exactMatchRequirements(){
85          exactMatchedRequirements =
86                              new ArrayList<RequirementType>();
87          unmatchedRequirements =
88                              new ArrayList<RequirementType>();
89
90          //takes the sets of requirements
91          ArrayList<RequirementType> nReqs =
92                                          n.getRequirements();
93          ArrayList<RequirementType> stReqs =
94                                          st.getRequirements();
95
96          //checks whether such sets exactly match
97          RequirementType req;
98          for(int i=0; i<stReqs.size(); i++){
99              req = stReqs.get(i);
100             boolean matched = false;
101             for(int j=0; j<nReqs.size() && !matched; j++){
102                 if(req.exactMatch(stReqs.get(j))){
103                     exactMatchedRequirements.add(req);
104                     matched = true;
105                 }
106             }
107             if(!matched)
108                 unmatchedRequirements.add(req);
```

```
109         }
110
111         //if ST and N expose the same number of
112         //requirements and there are no unmatched
113         //requirements, then the result is an
114         //"exact" match.
115         if(stReqs.size()==nReqs.size() &&
116            unmatchedRequirements.isEmpty()){
117             return Result.EXACT;
118         }
119         //otherwise there is "no-match"
120         return Result.NOMATCH;
121     }
122
123     protected Result exactMatchPolicies(){
124         if(st.getPolicies().areApplicableTo(n))
125             return Result.EXACT;
126         return Result.NOMATCH;
127     }
128
129     protected Result exactMatchProperties(){
130         exactMatchedProperties = new ArrayList<Property>();
131         unmatchedProperties = new ArrayList<Property>();
132
133         //takes the sets of properties
134         ArrayList<Property> nProps = n.getProperties();
135         ArrayList<Property> stProps = st.getProperties();
136
137         //checks whether such sets exactly match
138         Property prop;
139         for(int i=0; i<nProps.size(); i++){
140             prop = nProps.get(i);
141             boolean matched = false;
142             for(int j=0; j<stProps.size() &&
143                          !matched; j++){
144                 if(prop.exactMatch(stProps.get(j))){
145                     exactMatchedProperties.add(stProps
146                                                .get(j));
```

```
147                     matched = true;
148                 }
149             }
150             if(!matched)
151                 unmatchedProperties.add(prop);
152         }
153 
154         //if ST and N expose the same number of
155         //properties and there are no unmatched
156         //properties, then the result is an
157         //"exact" match.
158         if(nProps.size()==stProps.size() &&
159            unmatchedProperties.isEmpty()){
160             return Result.EXACT;
161         }
162         //otherwise there is "no match"
163         return Result.NOMATCH;
164     }
165 
166     protected Result exactMatchInterfaces(){
167         exactMatchedInterfaces =
168                         new ArrayList<Interface>();
169         unmatchedInterfaces =
170                         new ArrayList<Interface>();
171 
172         //takes the sets of interfaces
173         ArrayList<Interface> nInts = n.getInterfaces();
174         ArrayList<Interface> stInts = st.getInterfaces();
175 
176         //checks whether such sets exactly match
177         Interface interf;
178         for(int i=0; i<nInts.size(); i++){
179             interf = nInts.get(i);
180             boolean matched = false;
181             for(int j=0; j<stInts.size() && !matched; j++){
182                 if(interf.exactMatch(stInts.get(j))){
183                     exactMatchedInterfaces.add(stInts
184                                                 .get(j));
```

```
185                     matched = true;
186                 }
187             }
188             if(!matched)
189                 unmatchedInterfaces.add(interf);
190         }
191         //if ST and N expose the same number of
192         //interfaces and there are no unmatched
193         //interfaces, then the result is an
194         //"exact" match.
195         if(nInts.size()==stInts.size() &&
196            unmatchedInterfaces.isEmpty()){
197             return Result.EXACT;
198         }
199         //otherwise there is "no match"
200         return Result.NOMATCH;
201     }
202 
203     @Override
204     public Result match() {
205         //if the policies exposed by ST are not
206         //applicable to N, then we can stop our
207         //matching procedure.
208         polMatchResult = this.exactMatchPolicies();
209         if(polMatchResult==Result.NOMATCH)
210             return Result.NOMATCH;
211 
212         //otherwise, exactly match each kind of feature
213         capMatchResult = this.exactMatchCapabilities();
214         reqMatchResult = this.exactMatchRequirements();
215         propMatchResult = this.exactMatchProperties();
216         intMatchResult = this.exactMatchInterfaces();
217 
218         //and check whether is an "exact" match
219         if(capMatchResult==Result.EXACT &&
220                 reqMatchResult==Result.EXACT &&
221                 propMatchResult==Result.EXACT &&
222                 intMatchResult==Result.EXACT)
```

```
223             return Result.EXACT;
224         return Result.NOMATCH;
225     }
226 }
```

Listing 5.14: JAVA implementation of the `ExactMatchmaker` (2).

Once the features *exact* matching has been defined, we can override the `match` method to implement the

$$\texttt{n} \equiv \texttt{st}$$

expression. So, after having checked whether `st`'s policies are applicable to `n` (lines 205-210), we invoke `private` methods to check whether the capabilities, requirements, properties and interfaces of `n` exactly matches those of `st` (lines 213-216). Once this has been performed, we can employ partial results to compute the overall matching `Result` (lines 219-224).

Implementation of the *plug-in* matchmaker

So far, we developed the *exact* matchmaker. Now, we want to extend it in order to obtain the *plug-in* one (Listings 5.15 and 5.16).

```
1  public class PlugInMatchmaker extends ExactMatchmaker {
2      //needed to perform plugin matching of capabilities
3      protected ArrayList<CapabilityType>
4                          pluginMatchedCapabilities;
5      //needed to perform plugin matching of requirements
6      protected ArrayList<RequirementType>
7                          pluginMatchedRequirements;
8      //needed to perform plugin matching of properties
9      protected ArrayList<Property>
10                         pluginMatchedProperties;
11     //needed to perform plugin matching of interfaces
12     protected ArrayList<Interface>
13                         pluginMatchedInterfaces;
```

```
14
15      public PlugInMatchmaker(NodeType n,
16                              ServiceTemplate st) {
17          super(n,st);
18
19          //matching variables are initialized to null
20          pluginMatchedCapabilities = null;
21          pluginMatchedRequirements = null;
22          pluginMatchedProperties = null;
23          pluginMatchedInterfaces = null;
24      }
25
26      public ArrayList<CapabilityType>
27                      getPlugInMatchedCapabilities(){
28          return pluginMatchedCapabilities;
29      }
30
31      public ArrayList<RequirementType>
32                      getPlugInMatchedRequirements(){
33          return pluginMatchedRequirements;
34      }
35
36      public ArrayList<Property>
37                      getPlugInMatchedProperties(){
38          return pluginMatchedProperties;
39      }
40
41      public ArrayList<Interface>
42                      getPlugInMatchedInterfaces(){
43          return pluginMatchedInterfaces;
44      }
45
46      private Result plugInMatchCapabilities() {
47          pluginMatchedCapabilities =
48                              new ArrayList<CapabilityType>();
49
50          //if the capabilities of N "exact" matches those
51          //of ST, then the result is an "exact" matching
```

```
52          if(capMatchResult==Result.EXACT)
53              return Result.EXACT;
54
55          //otherwise, if the "exact" matching fails only
56          //because ST offers more capabilities than N,
57          //then the "plugin" matching is satisfied.
58          if(unmatchedCapabilities.isEmpty())
59              return Result.PLUGIN;
60
61          //otherwise, we have to check whether the
62          //unmatchedCapabilities of N let us obtain
63          //the "plugin" matching
64          ArrayList<CapabilityType> unmatcheds =
65                              new ArrayList<CapabilityType>();
66          ArrayList<CapabilityType> stCaps =
67                              st.getCapabilities();
68          CapabilityType cap;
69          boolean matched;
70          for(int i=0; i<unmatchedCapabilities.size(); i++) {
71              cap = unmatchedCapabilities.get(i);
72              matched = false;
73              for(int j=0; j<stCaps.size() &&
74                           !matched; j++) {
75                  if(stCaps.get(j).plugInMatch(cap)) {
76                      pluginMatchedCapabilities.add(stCaps
77                                                    .get(j));
78                      matched = true;
79                  }
80              }
81              if(!matched)
82                  unmatcheds.add(cap);
83          }
84          unmatchedCapabilities = unmatcheds;
85
86          //if there are no unmatchedCapabilities, then
87          //the plugin match is obtained.
88          if(unmatchedCapabilities.isEmpty())
89              return Result.PLUGIN;
```

```
90          return Result.NOMATCH;
91      }
```

Listing 5.15: JAVA implementation of the `PlugInMatchmaker` (1).

Let us focus on the `plugInMatchCapabilities` method (which returns a `Result` to indicate whether the capabilities of the *NodeType* `n` *plug-in* match those of the *ServiceTemplate* `st`). We know that, intuitively speaking,

$$\texttt{n}.\mathit{CapabilityDefinitions} \subseteq_C \texttt{st}.\mathit{Capabilities}$$

if and only if for each capability `cap` of `n` there exists a capability of `st` which is in `plugInMatching` with `cap` (lines 55-84). If this is the case, then we obtain that `n` *exactly* matches `st` (lines 86-89). Otherwise, we obtain the *unmatched* case (line 90). Please note that, if the capabilities of `n` exactly match those of `st`, then there is no need to proceed with the *plug-in* matching (lines 50-53).

As before, the way in which requirements, properties and interfaces matching is implemented (lines 93-136, 138-182 and 184-224) is analogous to the capabilities one (Listing 5.16).

```
93      private Result plugInMatchRequirements() {
94          pluginMatchedRequirements =
95                              new ArrayList<RequirementType>();
96
97          //if the requirements of ST "exact" matches those
98          //of N, then the result is an "exact" matching
99          if(reqMatchResult==Result.EXACT)
100             return Result.EXACT;
101
102         //otherwise, if the "exact" matching fails only
103         //because N exhibits more requirements than ST,
104         //then the "plugin" matching is satisfied.
105         if(unmatchedRequirements.isEmpty())
106             return Result.PLUGIN;
```

```
107
108         //otherwise, we have to check whether the
109         //unmatchedRequirements of ST let us obtain
110         //the "plugin" matching
111         ArrayList<RequirementType> unmatcheds =
112                             new ArrayList<RequirementType>();
113         ArrayList<RequirementType> nReqs =
114                             n.getRequirements();
115         RequirementType req;
116         boolean matched;
117         for(int i=0; i<unmatchedRequirements.size(); i++) {
118             req = unmatchedRequirements.get(i);
119             matched = false;
120             for(int j=0; j<nReqs.size() && !matched; j++) {
121                 if(nReqs.get(j).plugInMatch(req)) {
122                     pluginMatchedRequirements.add(req);
123                     matched = true;
124                 }
125             }
126             if(!matched)
127                 unmatcheds.add(req);
128         }
129         unmatchedRequirements = unmatcheds;
130
131         //if there are no unmatchedRequirements, then
132         //the "plugin" match is obtained.
133         if(unmatchedRequirements.isEmpty())
134             return Result.PLUGIN;
135         return Result.NOMATCH;
136     }
137
138     private Result plugInMatchProperties() {
139         pluginMatchedProperties =
140                               new ArrayList<Property>();
141
142         //if the properties of N "exact" matches those
143         //of ST, then the result is an "exact" matching
144         if(propMatchResult==Result.EXACT)
```

```
145             return Result.EXACT;
146
147         //otherwise, if the "exact" matching fails only
148         //because ST offers more properties than N,
149         //then the "plugin" matching is satisfied.
150         if(unmatchedProperties.isEmpty())
151             return Result.PLUGIN;
152
153         //otherwise, we have to check whether the
154         //unmatchedProperties of N let us obtain
155         //the "plugin" matching
156         ArrayList<Property> unmatcheds =
157                                 new ArrayList<Property>();
158         ArrayList<Property> stProps = st.getProperties();
159         Property prop;
160         boolean matched;
161         for(int i=0; i<unmatchedProperties.size(); i++) {
162             prop = unmatchedProperties.get(i);
163             matched = false;
164             for(int j=0; j<stProps.size() &&
165                         !matched; j++) {
166                 if(stProps.get(j).plugInMatch(prop)) {
167                     pluginMatchedProperties.add(stProps
168                                                 .get(j));
169                     matched = true;
170                 }
171             }
172             if(!matched)
173                 unmatcheds.add(prop);
174         }
175         unmatchedProperties = unmatcheds;
176
177         //if there are no unmatchedProperties, then
178         //the "plugin" match is obtained.
179         if(unmatchedProperties.isEmpty())
180             return Result.PLUGIN;
181         return Result.NOMATCH;
182     }
```

```
183
184     private Result plugInMatchInterfaces() {
185         pluginMatchedInterfaces =
186                                 new ArrayList<Interface>();
187
188         //if the interfaces of N "exact" matches those
189         //of ST, then the result is an "exact" matching
190         if(intMatchResult==Result.EXACT)
191             return Result.EXACT;
192
193         //otherwise, if the "exact" matching fails only
194         //because ST offers more interfaces than N,
195         //then the "plugin" matching is satisfied.
196         if(unmatchedInterfaces.isEmpty())
197             return Result.PLUGIN;
198
199         //otherwise, we have to check whether the
200         //unmatchedInterfaces of N let us obtain
201         //the "plugin" matching
202         ArrayList<Interface> unmatcheds =
203                                 new ArrayList<Interface>();
204         ArrayList<Interface> stInts = st.getInterfaces();
205         Interface interf;
206         Interface matched;
207         for(int i=0; i<unmatchedInterfaces.size(); i++) {
208             interf = unmatchedInterfaces.get(i);
209             //the "plugin" matching is performed by the
210             //class "Interface"
211             matched = interf.plugInMatch(st);
212             if(matched==null)
213                 unmatcheds.add(interf);
214             else
215                 pluginMatchedInterfaces.add(matched);
216         }
217         unmatchedInterfaces = unmatcheds;
218
219         //if there are no unmatchedInterfaces, then
220         //the plugin match is obtained.
```

```
221         if(unmatchedInterfaces.isEmpty())
222             return Result.PLUGIN;
223         return Result.NOMATCH;
224     }
225 
226     @Override
227     public Result match() {
228         Result superRes = super.match();
229 
230         //if ST "exactly" matches N, then we do
231         //not need to proceed further.
232         if(superRes==Result.EXACT)
233             return Result.EXACT;
234 
235         //otherwise, if the policies exposed by ST
236         //are not applicable to N, then we can
237         //stop our matching procedure.
238         if(polMatchResult==Result.NOMATCH)
239             return Result.NOMATCH;
240 
241         //otherwise, "plugin" match each kind of feature
242         capMatchResult = plugInMatchCapabilities();
243         reqMatchResult = plugInMatchRequirements();
244         propMatchResult = plugInMatchProperties();
245         intMatchResult = plugInMatchInterfaces();
246 
247         if(((capMatchResult==Result.EXACT ||
248              capMatchResult==Result.PLUGIN))
249            &&
250            ((reqMatchResult==Result.EXACT ||
251              reqMatchResult==Result.PLUGIN))
252            &&
253            ((propMatchResult==Result.EXACT ||
254              propMatchResult==Result.PLUGIN))
255            &&
256            ((intMatchResult==Result.EXACT ||
257              intMatchResult==Result.PLUGIN)))
258             return Result.PLUGIN;
```

```
259             return Result.NOMATCH;
260         }
261 }
```

Listing 5.16: JAVA implementation of the `PlugInMatchmaker` (2).

Once the features *plug-in* matching has been defined, we can override the `match` method to implement the

$$\texttt{n} \subseteq \texttt{st}$$

expression. So, after having checked whether `n` *exact* matches `st` (lines 230-233) and whether `st`'s policies are applicable to `n` (lines 235-239), we invoke `private` methods to check whether the capabilities, requirements, properties and interfaces of `n` plug-in matches those of `st` (lines 242-245). Once this has been performed, we can employ partial results to compute the overall matching `Result` (lines 247-259).

5.4 Concluding remarks

So far, we provide a (tested[8]) JAVA implementation of the black-box *exact* and *plug-in* matching notions. It is worth noting that the computed matching sets (such as `exactMatchedCapabilities`, `pluginMatched-Capabilities` and `unmatchedCapabilities`) will be employed in the development of the `st` adaptation and of the other matching notions. How to do it is out of the purposes of this chapter (since we only want to demonstrate the feasibility of the proposed matching procedure).

[8] Appendix A shows an example of made tests.

Chapter 6

Conclusions

Before concluding, it is worth making some final remarks so as to:

- summarize the contributions of this thesis to the TOSCA specification (Section 6.1),
- discuss related work (Section 6.2), and
- provide an overview of possible future work (Section 6.3).

6.1 Summary of contributions

In this thesis, after defining the notion of *exact matching* between TOSCA *ServiceTemplate* and *NodeType* elements, we have defined three other types of matching (*plug-in*, *flexible* and *white-box*), each permitting to ignore larger sets of non-relevant syntactic differences when type-checking *ServiceTemplate* elements with respect to node types. More precisely:

- the *plug-in* matching extends the *exact* one by considering a *ServiceTemplate* that "require less" and "offers more" than a *NodeType* compatible with the latter;

- the *flexible* matching in turn extends the *plug-in* one by employing ontologies to check whether differently named features are semantically equivalent (so as to ignore non-relevant syntactic differences);
- the *white-box* matching in turn extends the *flexible* one by searching missing (equivalent) features inside the service topology. It still employs ontologies to check whether differently named features can be considered semantically equivalent. Furthermore, it employs a recursive algorithm to detect available compositions of operations which are semantically equivalent to needed (missing) operations.

Furthermore, we have also described how a *ServiceTemplate* that *plug-in*, *flexibly* or *white-box* matches a *NodeType* can be suitably adapted so as to *exactly* match it.

As we already mentioned at the very beginning of this work, the presented results intend to contribute to the formal definition of TOSCA. More precisely, the different types of matching defined in this thesis can be used to develop a matchmaker to be fruitfully integrated in the TOSCA implementations that are currently under development (such as the Valesca editor [32] and the OpenTOSCA IDE [27]) in order to enhance their typed node matching capabilities. The development of such (pluggable) matchmaker will contribute to cloud service portability and multi-cloud service development. Indeed, with the availability of such an implementation, a cloud service developer will have the possibility to:

- employ more available (adapted) cloud services instead of developing her application's encompassed components,
- migrate more application's components across heterogeneous clouds by changing the used available (adapted) cloud services, and
- choose between more different cloud service providers the one which provides the compatible service with the best quality-price ratio.

6.2 Related work

Service matching

As we already mentioned at the very beginning of this thesis, our work started from the observation that while the matching between *ServiceTemplate* and *NodeType* elements is indicated in [26] as a way to instantiate TOSCA *NodeType* elements, no (formal) definition of *matching* is given either in [25] or in [26]. A concrete definition of matching for TOSCA is used in [33] to define a way to merge TOSCA services by matching entire portions of their topology templates. The definition of matching of single service components employed in [33] is however very strict, as two service components are considered to match only if they expose the same qualified name. This work aims at contributing to the TOSCA specification by proposing four definitions of matching between *ServiceTemplate* and *NodeType* elements, each identifying larger sets of *ServiceTemplate* elements that can be adapted so as to (exactly) match a *NodeType*.

The problem of how to match (Web) services has been extensively studied in recent years. Many approaches are ontology-aware [28], like for instance the ontology-aware matchmaker for OWL-S services described in [16]. Other approaches are behaviour-aware, like the (ontology-aware) trace-based matching of YAWL services defined in [9], the (ontology-aware) behavioural congruence for OWL-S services defined in [3], or the graph transformation based matching defined in [12] and the heuristic black-box matching described in [14] for WS-BPEL processes. The main difference between the aforementioned approaches and ours is the type of information considered when matching single nodes. The matching levels considered for instance in [16] and [14] are all defined in terms of input and output data, while we consider also technology requirements and capabilities, properties and policies.

On the other hand, many proposals of QoS-aware service matching

have been developed, like for instance [19] or [23]. Generally speaking, the notion of matching defined in the present thesis differs from most QoS-aware matching approaches since it compares *types* rather than actual *values* of extra-functional features (like QoS). A type-based definition of matching is defined in [13] to type check *stream flows* for interactive distributed multimedia applications. While the context of [13] is different from ours, two of the matching conditions considered in [13] resemble our notions of exact and plug-in matching, even if for simpler service abstractions.

Summing up, to the best of our knowledge, our definition of matching is the first definition of (TOSCA) node matching to take into account both functional and extra-functional features, by relying both on types and on ontologies to overcome non-relevant syntactic information.

Service adaptation

As we already mentioned (in Sections 3.2 and 3.3) a *ServiceTemplate ST* that *plug-in* or *flexibly* matches a *NodeType N* can be adapted into a new *ServiceTemplate ST′* that *exactly* matches N. The adaptation technique basically consists of creating a new *ServiceTemplate ST′* that includes ST as internal node, and of suitably exposing (via *BoundaryDefinitions*) the capabilities, policies, properties, and interfaces of the *NodeType* to be matched. The transformation implemented by such adaptation vaguely reminds the adaptation techniques described in [9] and [10] to implement (more complex) input-output and behaviour transformations of YAWL workflows, respectively.

Furthermore, as we discussed in Chapter 4, the adaptation needed to transform a *ServiceTemplate ST* that *white-box* matches a *NodeType N* into a new *ServiceTemplate ST′* that *exactly* matches N requires to generate a plan to combine a set of operations into an input-output behaviour equivalent to a given operation. As seen, such plans can be generated by

adapting the ontology-aware discovery algorithm presented in [5].

6.3 Future work

The whole thesis orbits on the definition of matchmaking procedures between *NodeType* and *ServiceTemplate* elements. A (tested) partial[1] implementation of such matchmaking procedures is described in Chapter 5. Completing the implementation of the whole matchmaker is one direction for immediate future work.

Furthermore, we employed plan generation only when *white-box* matching the operations of node types and service templates. As someone could note, such a plan generation should be employed also when we (black-box) *plug-in/flexibly* match such features. Suitably integrating the OperationSetsDiscovering algorithm with such matching procedures is another extension left for future work.

Finally, we restricted our work to service type-checking. It is clear that a full-fledged matchmaker will need to employ also *values* when comparing services. To do it, we have to move the focus from *NodeType* elements to *NodeTemplate* ones (see Figure 6.1). Since the type of *NodeTemplate* elements is known, we do not encounter problems when type-checking it with respect to service templates. But, how can we match *values*? To consider more than trivial equality comparison, we have to define a way to indicate how to match those features. As an example, we could think of some kind of (*policies/properties*) "contracts" to indicate the way in which each single feature value should be matched. A definition of such contracts is another possible extension of this work.

[1] Please recall that the objective of such an implementation objective is only to show the JAVA feasibility of the matchmaker.

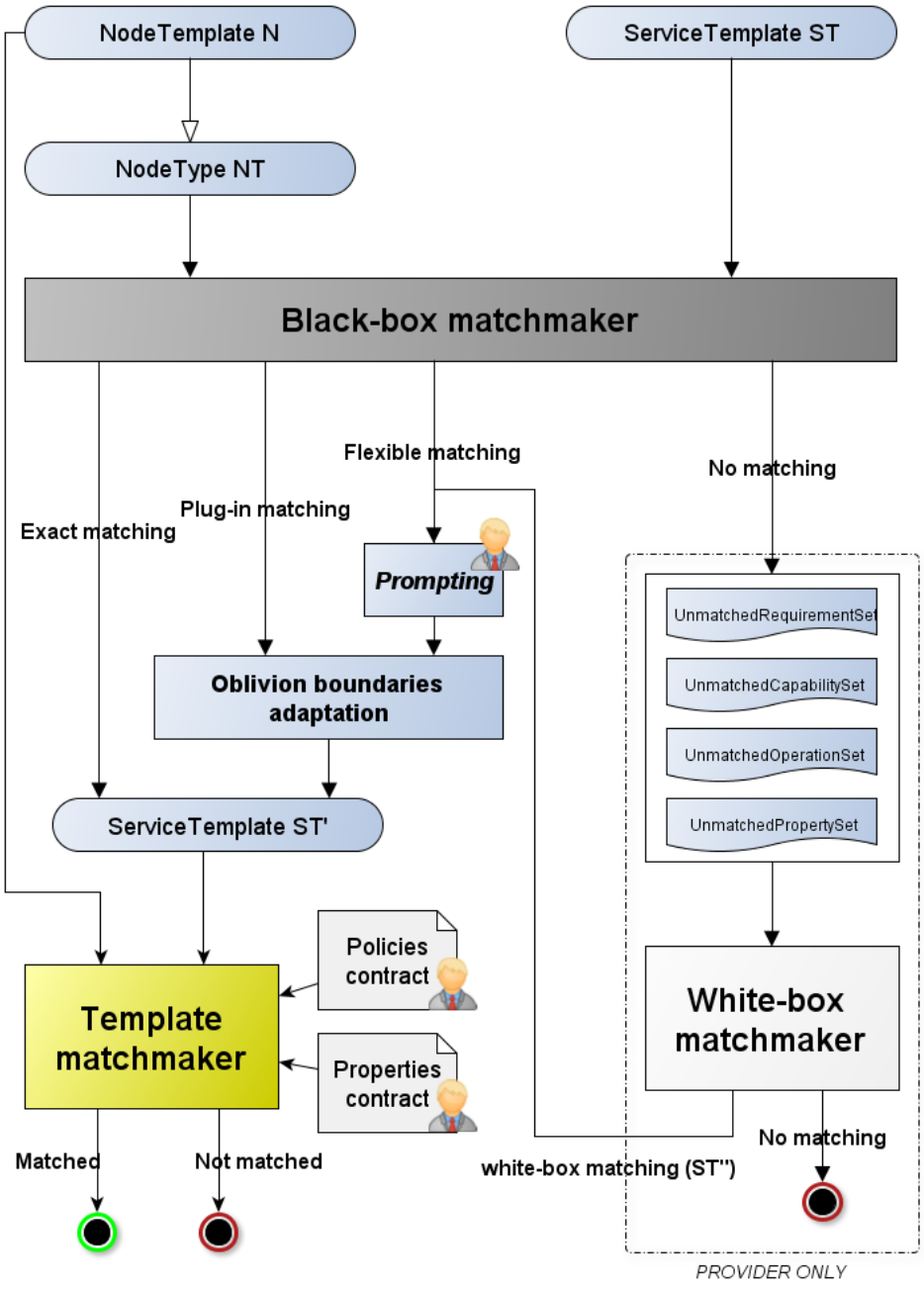


Figure 6.1: Complete matching (and adaptation) procedure.

Appendix A

Example of test of the proof-of-concept implementation

Chapter 5 has shown a (tested) partial implementation of the matchmaking procedure. This Appendix aims at showing an example of the tests which can be executed on such an implementation.

Consider the node type and service templates in Figure A.1 and suppose that all services exhibit a policy (of type *RapidCalculatorPolicyType*) which is applicable to *CalculatorNodeType*. Looking at the definitions in Chapter 3, we observe that the following conditions hold:

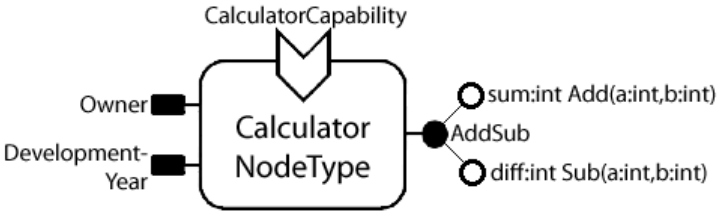
$$CalculatorNodeType \equiv Service,$$

$$CalculatorNodeType \not\equiv ServiceBis,$$

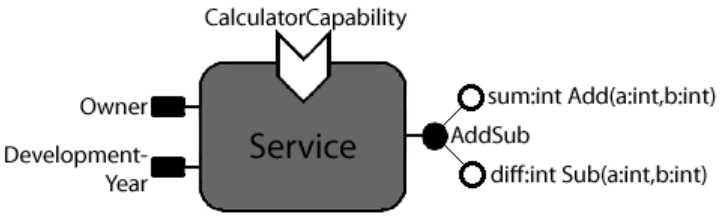
$$CalculatorNodeType \subseteq ServiceBis, \text{ and}$$

$$CalculatorNodeType \not\subseteq ServiceTer.$$

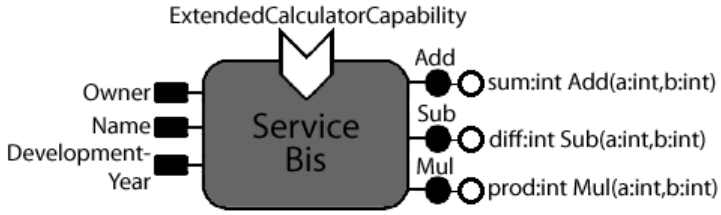
We are going to show that, employing the developed (partial) matchmaker, we obtain such results.



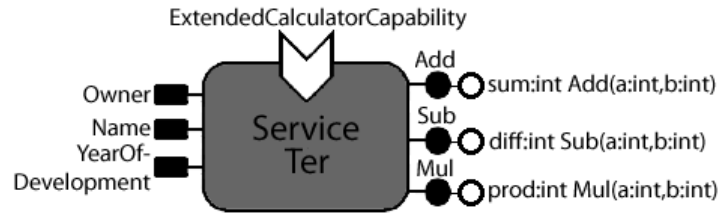
(a)



(b)



(c)



(d)

Figure A.1: Node type and service templates employed in testing the *proof-of-concept implementation.*

A.1 Implementation of the needed input

This section will show how to represent the considered services (Figure A.1) in JAVA.

Implementation of basic features

Looking at the problem definition, we observe that (in order to represent the desired services) we need to develop some basic features.

First, we need to implement the required *CalculatorCapabilityType* (Listing A.1) and *ExtendedCalculatorCapabilityType* (Listing A.2). To do it, we have to suitably extend the generic *CapabilityType* class.

```
1 public class CalculatorCapabilityType
2              extends CapabilityType {
3     public CalculatorCapabilityType(String name){
4         super(name);
5     }
6 }
```

Listing A.1: JAVA implementation of the `CalculatorCapabilityType`.

```
1 public class ExtendedCalculatorCapabilityType
2              extends CalculatorCapabilityType {
3     public ExtendedCalculatorCapabilityType(String name){
4         super(name);
5     }
6 }
```

Listing A.2: JAVA implementation of the `ExtendedCalculatorCapabilityType`.

Then, we have to implement the type of the policy[1] exposed by all services (Listing A.3).

[1]Please recall that such a policy type is assumed to be applicable to the *CalculatorNodeType*.

```
1 public class RapidCalculatorPolicyType extends PolicyType {
2     public RapidCalculatorPolicyType() {
3         super();
4         applicabilityDomain.add(new CalculatorNodeType());
5     }
6 }
```

Listing A.3: JAVA implementation of the `RapidCalculatorPolicyType`.

Now, we have all the ground needed to implement the desired node type and service templates.

Implementation of *CalculatorNodeType*

The JAVA source code which implements the *CalculatorNodeType* is shown in Listing A.4.

```
1  public class CalculatorNodeType extends NodeType {
2      public CalculatorNodeType() {
3          super();
4          //capabilities
5          CapabilityType calcCap = new
6                         CalculatorCapabilityType(
7                                 "CalculatorCapability");
8          capabilities.add(calcCap);
9
10         //properties
11         Property owner = new Property("Owner","");
12         properties.add(owner);
13         Property devYear = new Property("DevelopmentYear",
14                                         new Integer(0));
15         properties.add(devYear);
16
17         //interfaces
18         //Add operation
19         OperationParameter add1 = new
20                         OperationParameter("a",
```

```
21                                          new Integer(0),
22                                          true);
23          OperationParameter add2 = new
24                      OperationParameter("b",
25                                          new Integer(0),
26                                          true);
27          ArrayList<OperationParameter> addInputs = new
28                      ArrayList<OperationParameter>();
29          addInputs.add(add1);
30          addInputs.add(add2);
31          OperationParameter addRes = new
32                      OperationParameter("sum",
33                                          new Integer(0),
34                                          true);
35          ArrayList<OperationParameter> addOutputs = new
36                      ArrayList<OperationParameter>();
37          addOutputs.add(addRes);
38          Operation add = new Operation("Add", addInputs,
39                                        addOutputs);
40          //Sub operation
41          OperationParameter sub1 = new
42                      OperationParameter("a",
43                                          new Integer(0),
44                                          true);
45          OperationParameter sub2 = new
46                      OperationParameter("b",
47                                          new Integer(0),
48                                          true);
49          ArrayList<OperationParameter> subInputs = new
50                      ArrayList<OperationParameter>();
51          subInputs.add(sub1);
52          subInputs.add(sub2);
53          OperationParameter subRes = new
54                      OperationParameter("diff",
55                                          new Integer(0),
56                                          true);
57          ArrayList<OperationParameter> subOutputs = new
58                      ArrayList<OperationParameter>();
```

```
59          subOutputs.add(subRes);
60          Operation sub = new Operation("Sub", subInputs,
61                                        subOutputs);
62          //AddSub interface
63          ArrayList<Operation> ops = new
64                          ArrayList<Operation>();
65          ops.add(add);
66          ops.add(sub);
67          Interface addSub = new Interface("AddSub", ops);
68          interfaces.add(addSub);
69      }
70  }
```

Listing A.4: JAVA implementation of the `CalculatorNodeType`.

Implementation of *Service*

The available *Service* is implemented in JAVA as shown in Listing A.5.

```
1  public class Service extends ServiceTemplate {
2      public Service() {
3          super();
4          //capabilities
5          CapabilityType calcCap = new
6                          CalculatorCapabilityType(
7                                  "CalculatorCapability");
8          capabilities.add(calcCap);
9
10         //policies
11         PolicyType rapid = new RapidCalculatorPolicyType();
12         policies.addPolicy(rapid);
13
14         //properties
15         Property owner = new Property("Owner","Goofy");
16         properties.add(owner);
17         Property devYear = new Property("DevelopmentYear",
18                                         new Integer(2013));
19         properties.add(devYear);
```

```
20
21          //interfaces
22          //Add operation
23          OperationParameter add1 = new
24                      OperationParameter("a",
25                                         new Integer(0),
26                                         true);
27          OperationParameter add2 = new
28                      OperationParameter("b",
29                                         new Integer(0),
30                                         true);
31          ArrayList<OperationParameter> addInputs = new
32                      ArrayList<OperationParameter>();
33          addInputs.add(add1);
34          addInputs.add(add2);
35          OperationParameter addRes = new
36                      OperationParameter("sum",
37                                         new Integer(0),
38                                         true);
39          ArrayList<OperationParameter> addOutputs = new
40                      ArrayList<OperationParameter>();
41          addOutputs.add(addRes);
42          Operation add = new Operation("Add", addInputs,
43                                        addOutputs);
44          //Sub operation
45          OperationParameter sub1 = new
46                      OperationParameter("a",
47                                         new Integer(0),
48                                         true);
49          OperationParameter sub2 = new
50                      OperationParameter("b",
51                                         new Integer(0),
52                                         true);
53          ArrayList<OperationParameter> subInputs = new
54                      ArrayList<OperationParameter>();
55          subInputs.add(sub1);
56          subInputs.add(sub2);
57          OperationParameter subRes = new
```

```
58                          OperationParameter("diff",
59                                                  new Integer(0),
60                                                  true);
61          ArrayList<OperationParameter> subOutputs = new
62                          ArrayList<OperationParameter>();
63          subOutputs.add(subRes);
64          Operation sub = new Operation("Sub", subInputs,
65                                          subOutputs);
66          //AddSub interface
67          ArrayList<Operation> ops = new
68                          ArrayList<Operation>();
69          ops.add(add);
70          ops.add(sub);
71          Interface addSub = new Interface("AddSub", ops);
72          interfaces.add(addSub);
73      }
74  }
```

Listing A.5: JAVA implementation of `Service`.

Implementation of *ServiceBis*

Looking at Figure A.1, we observe that *ServiceBis* differs from *Service* since:

- it exposes a capability whose type is derived from that of the capability of *Service*,
- it exhibits an additional *Name* property, and
- it splits the *AddSub* interface into two distinct interfaces and adds a new operation *Mul* (contained in a homonym interface).

So, if we (suitably) modify the *Service*'s source code, then we obtain *ServiceBis* (Listing A.6).

```
1  public class ServiceBis extends ServiceTemplate {
2      public ServiceBis() {
3          super();
4          //capabilities
5          CapabilityType calcCap = new
6                          ExtendedCalculatorCapabilityType(
7                              "ExtendedCalculatorCapability");
8          capabilities.add(calcCap);
9
10         //policies
11         PolicyType rapid = new RapidCalculatorPolicyType();
12         policies.addPolicy(rapid);
13
14         //properties
15         Property owner = new Property("Owner","Goofy");
16         properties.add(owner);
17         Property name = new Property("Name","ServiceBis");
18         properties.add(name);
19         Property devYear = new Property("DevelopmentYear",
20                                         new Integer(2013));
21         properties.add(devYear);
22
23         //interfaces
24         //Add operation
25         OperationParameter add1 = new
26                         OperationParameter("a",
27                                            new Integer(0),
28                                            true);
29         OperationParameter add2 = new
30                         OperationParameter("b",
31                                            new Integer(0),
32                                            true);
33         ArrayList<OperationParameter> addInputs = new
34                         ArrayList<OperationParameter>();
35         addInputs.add(add1);
36         addInputs.add(add2);
37         OperationParameter addRes = new
38                         OperationParameter("sum",
```

```
39                                          new Integer(0),
40                                          true);
41          ArrayList<OperationParameter> addOutputs = new
42                      ArrayList<OperationParameter>();
43          addOutputs.add(addRes);
44          Operation add = new Operation("Add", addInputs,
45                                      addOutputs);
46          //Sub operation
47          OperationParameter sub1 = new
48                      OperationParameter("a",
49                                          new Integer(0),
50                                          true);
51          OperationParameter sub2 = new
52                      OperationParameter("b",
53                                          new Integer(0),
54                                          true);
55          ArrayList<OperationParameter> subInputs = new
56                      ArrayList<OperationParameter>();
57          subInputs.add(sub1);
58          subInputs.add(sub2);
59          OperationParameter subRes = new
60                      OperationParameter("diff",
61                                          new Integer(0),
62                                          true);
63          ArrayList<OperationParameter> subOutputs = new
64                      ArrayList<OperationParameter>();
65          subOutputs.add(subRes);
66          Operation sub = new Operation("Sub", subInputs,
67                                      subOutputs);
68          //Mul operation
69          OperationParameter mul1 = new
70                      OperationParameter("a",
71                                          new Integer(0),
72                                          true);
73          OperationParameter mul2 = new
74                      OperationParameter("b",
75                                          new Integer(0),
76                                          true);
```

```
77          ArrayList<OperationParameter> mulInputs = new
78                          ArrayList<OperationParameter>();
79          mulInputs.add(sub1);
80          mulInputs.add(sub2);
81          OperationParameter mulRes = new
82                          OperationParameter("prod",
83                                                  new Integer(0),
84                                                  true);
85          ArrayList<OperationParameter> mulOutputs = new
86                          ArrayList<OperationParameter>();
87          mulOutputs.add(mulRes);
88          Operation mul = new Operation("Mul", mulInputs,
89                                          mulOutputs);
90          //Add interface
91          ArrayList<Operation> opsAdd = new
92                          ArrayList<Operation>();
93          opsAdd.add(add);
94          Interface addInt = new Interface("Add", opsAdd);
95          interfaces.add(addInt);
96          //Sub interface
97          ArrayList<Operation> opsSub = new
98                          ArrayList<Operation>();
99          opsSub.add(sub);
100         Interface subInt = new Interface("Sub", opsSub);
101         interfaces.add(subInt);
102         //Mul interface
103         ArrayList<Operation> opsMul = new
104                         ArrayList<Operation>();
105         opsMul.add(add);
106         Interface mulInt = new Interface("Mul", opsMul);
107         interfaces.add(mulInt);
108     }
109 }
```

Listing A.6: JAVA implementation of `ServiceBis`.

Implementation of *ServiceTer*

Looking at Figure A.1, we observe that *ServiceTer* differs from *ServiceBis* since it exposes the property *YearOfDevelopment* instead of the *DevelopmentYear* one. So, the source code of *ServiceTer* (Listing A.7) can be simply obtained by opportunely modifying line 19 of *ServiceBis* source code.

```
1  public class ServiceTer extends ServiceTemplate {
2      public ServiceTer() {
3          super();
4          //capabilities
5          CapabilityType calcCap = new
6                         ExtendedCalculatorCapabilityType(
7                             "ExtendedCalculatorCapability");
8          capabilities.add(calcCap);
9
10         //policies
11         PolicyType rapid = new RapidCalculatorPolicyType();
12         policies.addPolicy(rapid);
13
14         //properties
15         Property owner = new Property("Owner","Goofy");
16         properties.add(owner);
17         Property name = new Property("Name","ServiceBis");
18         properties.add(name);
19         Property devYear = new Property(
20                                       "YearOfDevelopment",
21                                       new Integer(2013));
22         properties.add(devYear);
23
24         //interfaces
25         //Add operation
26         OperationParameter add1 = new
27                         OperationParameter("a",
28                                            new Integer(0),
29                                            true);
30         OperationParameter add2 = new
```

```
31                      OperationParameter("b",
32                                         new Integer(0),
33                                         true);
34          ArrayList<OperationParameter> addInputs = new
35                      ArrayList<OperationParameter>();
36          addInputs.add(add1);
37          addInputs.add(add2);
38          OperationParameter addRes = new
39                      OperationParameter("sum",
40                                         new Integer(0),
41                                         true);
42          ArrayList<OperationParameter> addOutputs = new
43                      ArrayList<OperationParameter>();
44          addOutputs.add(addRes);
45          Operation add = new Operation("Add", addInputs,
46                                        addOutputs);
47          //Sub operation
48          OperationParameter sub1 = new
49                      OperationParameter("a",
50                                         new Integer(0),
51                                         true);
52          OperationParameter sub2 = new
53                      OperationParameter("b",
54                                         new Integer(0),
55                                         true);
56          ArrayList<OperationParameter> subInputs = new
57                      ArrayList<OperationParameter>();
58          subInputs.add(sub1);
59          subInputs.add(sub2);
60          OperationParameter subRes = new
61                      OperationParameter("diff",
62                                         new Integer(0),
63                                         true);
64          ArrayList<OperationParameter> subOutputs = new
65                      ArrayList<OperationParameter>();
66          subOutputs.add(subRes);
67          Operation sub = new Operation("Sub", subInputs,
68                                        subOutputs);
```

```
69          //Mul operation
70          OperationParameter mul1 = new
71                        OperationParameter("a",
72                                           new Integer(0),
73                                           true);
74          OperationParameter mul2 = new
75                        OperationParameter("b",
76                                           new Integer(0),
77                                           true);
78          ArrayList<OperationParameter> mulInputs = new
79                        ArrayList<OperationParameter>();
80          mulInputs.add(sub1);
81          mulInputs.add(sub2);
82          OperationParameter mulRes = new
83                        OperationParameter("prod",
84                                           new Integer(0),
85                                           true);
86          ArrayList<OperationParameter> mulOutputs = new
87                        ArrayList<OperationParameter>();
88          mulOutputs.add(mulRes);
89          Operation mul = new Operation("Mul", mulInputs,
90                                        mulOutputs);
91          //Add interface
92          ArrayList<Operation> opsAdd = new
93                        ArrayList<Operation>();
94          opsAdd.add(add);
95          Interface addInt = new Interface("Add", opsAdd);
96          interfaces.add(addInt);
97          //Sub interface
98          ArrayList<Operation> opsSub = new
99                        ArrayList<Operation>();
100         opsSub.add(sub);
101         Interface subInt = new Interface("Sub", opsSub);
102         interfaces.add(subInt);
103         //Mul interface
104         ArrayList<Operation> opsMul = new
105                       ArrayList<Operation>();
106         opsMul.add(add);
```

```
107         Interface mulInt = new Interface("Mul", opsMul);
108         interfaces.add(mulInt);
109     }
110 }
```

Listing A.7: JAVA implementation of `ServiceTer`.

A.2 Implementation of the `Test`

As mentioned at the very beginning of this Appendix A, we want to show that the following conditions hold:

$$CalculatorNodeType \equiv Service \text{ (1)},$$

$$CalculatorNodeType \not\equiv ServiceBis \text{ (2)},$$

$$CalculatorNodeType \subseteq ServiceBis \text{ (3), and}$$

$$CalculatorNodeType \not\subseteq ServiceTer \text{ (4)}.$$

To do it, we developed the runnable `Test` class (Listing A.8).

```
1  public class Test {
2      public static void main(String[] args) {
3      //needed inputs
4          CalculatorNodeType n = new CalculatorNodeType();
5          Service s = new Service();
6          ServiceBis sBis = new ServiceBis();
7          ServiceTer sTer = new ServiceTer();
8
9      //needed to store matching results
10         Matchmaker.Result res;
11
12         //test (1)
13         System.out.print("Test(1):");
14         ExactMatchmaker ex = new ExactMatchmaker(n, s);
15         res = ex.match();
```

```
16          if(res == ExactMatchmaker.Result.EXACT)
17              System.out.println("OK!");
18          else
19              System.out.println("KO!");
20
21          //test (2)
22          System.out.print("Test(2):");
23          ExactMatchmaker ex2 = new ExactMatchmaker(n, sBis);
24          res = ex2.match();
25          if(res == ExactMatchmaker.Result.NOMATCH)
26              System.out.println("OK!");
27          else
28              System.out.println("KO!");
29
30          //test (3)
31          System.out.print("Test(3):");
32          PlugInMatchmaker pl = new PlugInMatchmaker(n,
33                                                     sBis);
34          res = pl.match();
35          if(res == ExactMatchmaker.Result.PLUGIN)
36              System.out.println("OK!");
37          else
38              System.out.println("KO!");
39
40          //test (4)
41          System.out.print("Test(4):");
42          PlugInMatchmaker pl2 = new PlugInMatchmaker(n,
43                                                      sTer);
44          res = pl2.match();
45          if(res == ExactMatchmaker.Result.NOMATCH)
46              System.out.println("OK!");
47          else
48              System.out.println("KO!");
49      }
50  }
```

Listing A.8: JAVA implementation of the `Test`.

A.3 Concluding remarks

Running the presented `Test` JAVA code we obtain what shown in Figure A.2. So, with this Appendix A, we have shown how the provided

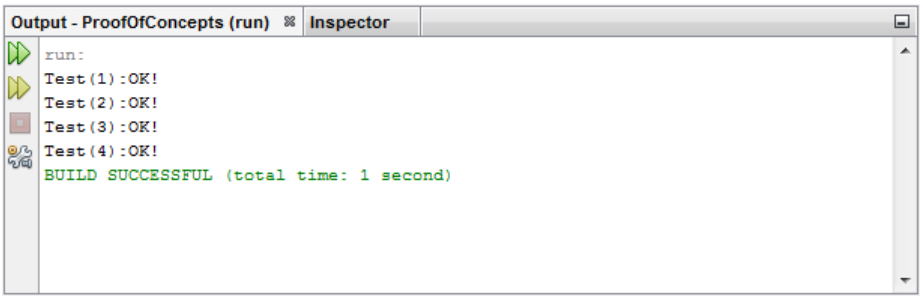


Figure A.2: Test results.

implementation (despite it is partial) let us perform some comparison tests.

Bibliography

[1] M. Armbrust, A. Fox, R. Griffith, A.D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, *A View of Cloud Computing*, Communications of the ACM, vol. 53, issue 4, pages 50-58, April 2010, doi:10.1145/1721654.1721672;

[2] T. Binz, G. Breiter, F. Leyman, T. Spatzier, *Portable Cloud Services Using TOSCA*, IEEE Internet Computing, vol. 16, no. 3, pp. 80-85, May-June 2012, doi:10.1109/MIC.2012.43;

[3] F. Bonchi, A. Brogi, S. Corfini, F. Gadducci, *A Net-based Approach to Web Services Publication and Replaceability*, Fundamenta informaticae, 94(3-4):205-309, 2009.

[4] A. Brogi, *Service Adaptation*, ESOCC'12, Bertinoro, September 2012;

[5] A. Brogi, S. Corfini, *Behaviour-aware discovery of Web service compositions*, International Journal of Web Service Research, vol. 4, issue 3, July 2007;

[6] A. Brogi, S. Corfini, J. F. Aldana, I. Navas, *Automated Discovery of Compositions of Services Described with Separate Ontologies*, Proceedings of the 4th International Conference on Service Oriented Computing (ICSOC 06), LNCS vol. 4294, pages 509-514, 2006;

[7] A. Brogi, S. Corfini, J.F. Aldana, I. Navas, *A Prototype for Discovering Compositions of Semantic Web Services*, Proceedings of the

Third Italian Workshop on Semantic Web Applications and Perspectives, 2006;

[8] A. Brogi, S. Corfini, R. Popescu, *Semantics-Based Composition-Oriented Discovery of Web Services*, ACM Transaction on Internet Technology, vol.8, no. 4, article 19, September 2008;

[9] A. Brogi, R. Popescu, *Service Adaptation through Trace Inspection*, International Journal of Business Process Integration and Management, vol. 2, no. 1, pp. 9-16 (8), June 2007;

[10] A. Brogi, R. Popescu, M. Tanca, *Design and implementation of SATOR: A Web service aggregator*, ACM Transactions on Software Engineering and Methodology, 19(3), 2010;

[11] S. Corfini, *Composition-oriented Web Service Discovery*, Lightning Source Incorporated, isbn:9783639041569, 2008;

[12] J. C. Corrales, D. Grigori, M. Bouzeghoub, *BPEL processes matchmaking for service discovery*, Proceeding ODBASE'06/OTM'06 Proceedings of the 2006 Confederated international conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I, pp. 237-254, doi:10.1007/11914853-15, 2006;

[13] F. Eliassen, S. Mehus, *Type Checking Stream Flow Endpoints*, Middleware '98, Springer London, pp. 305-320, doi:10.1007/978-1-4471-1283-9_19, 1998;

[14] R. Eshuis, P. Grefen, *Structural Matching of BPEL Processes*, Proceeding ECOWS '07 Proceedings of the Fifth European Conference on Web Services, pp. 171-180 , doi:10.1109/ECOWS.2007.26, 2007;

[15] G. Gallo, G. Longo, S. Nguyen, S. Pallottino, *Directed Hypergraphs and Applications*, Discrete Applied Mathematics, 42(2):177–201, 1993;

[16] M. Klusch, B. Fries, K. Sycara, Journal, *OWLS-MX: A hybrid Semantic Web service matchmaker for OWL-S services*, Web Semantics: Science, Services and Agents on the World Wide Web archive, vol. 7, issue 2, pp 121-133, doi:10.1016/j.websem.2008.10.001, April 2009;

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

[18] P. Lipton, *Escaping Vendor Lock-In with TOSCA, an Emerging Cloud Standard for Portability*, CA Technology Exchange (CA Labs Research), CA Technologies, vol.4, issue 1, pp. 49-55, January 2013;

[19] F. Mahdikhani, M.R. Hashemi, M. Sirjani, *QoS Aspects in Web Services Compositions*, Proceedings of IEEE International Symposium on Service-Oriented System Engineering SOSE '08., pp. 239-244, doi:10.1109/SOSE.2008.39, 2008;

[20] J. Martinez-Gil, I. Navas-Delgado, J.F. Aldana-Montes, *MaF: An Ontology Matching Framework*, Journal of Universal Computer Science, vol. 18, no. 3, pp. 194-217, January 2012;

[21] P. Mell, T. Grance, *The NIST Definition of Cloud Computing*, NIST Special Publication 800-145, September 2011, `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`;

[22] J. Miranda, J. Guillen, J.M. Murillo, C. Canal, *Development of Adaptive Multi-cloud Applications: A Model-Driven Approach*, MODELSWARD, 2013;

[23] S.B. Mokhtar, D. Preuveneers, N. Georgantas, V. Issarny, Y. Berbers, *EASY: Efficient semAntic Service discoverY in pervasive computing environments with QoS and context support*, J. Syst.

Softw., vol. 81, no. 5, pp. 785-808, doi: 10.1016/j.jss.2007.07.030, Elsevier Science Inc., May 2008;

[24] P. Muschamp, *An introduction to Web Services*, BT Technology Journal, vol. 22, no. 1, January 2004;

[25] OASIS TOSCA TC, *Topology and Orchestration Specification for Cloud Applications Version 1.0*, `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`;

[26] OASIS TOSCA TC, *Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0*, `http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf`;

[27] *OpenTOSCA*, `http://www.iaas.uni-stuttgart.de/OpenTOSCA/indexE.php`;

[28] D. O'Sullivan, D. Lewis, *Semantically driven service interoperability for pervasive computing*, Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access, pp. 17-24, doi:10.1145/940923.940927, 2003;

[29] D. Petcu, G. Macariu, S. Panica, C. Craciun, *Portable Cloud Applications - From Theory to Practice*, Future Generation Computer Systems, vol. 29, issue 6, pp. 1417–1430, 2012;

[30] L. Schubert, K. Jeffery (Eds.), *Advances in Clouds, Expert Group Report*, Public version 1.0, European Commission, `http://ec.europa.eu/information_society/newsroom/cf//document.cfm?doc_id=1174`, 2012;

[31] S. Tummalapalli, P. RaviKanth, K. Yuvaraj, S. Velagapudi, *TOSCA Enabling Cloud Portability*, International Journal of Advanced Research and Computer Engineering & Technology, vol. 2, no. 3, 2013;

[32] *Valesca*, `http://www.cloudcycle.org/en/valesca/`;

[33] A. Weiss, *Master Thesis: Merging of TOSCA Cloud Topology Templates*, Institute of Architecture of Application Systems, University of Stuttgart, http://elib.uni-stuttgart.de/opus/volltexte/2012/7932/pdf/MSTR_3341.pdf;

[34] G.J. Woeginger, Zhongliang Yu, *On the equal-subset-sum problem*, Information Processing Letters, vol. 42, no. 6, pp. 299-302, 1992;

[35] W3C, *Extensible Markup Language (XML) 1.0 (Fifth Edition)*, 26 November 2008, http://www.w3.org/TR/xml

[36] W3C, *XML Path Language (XPath) Version 1.0*, 16 November 1999, http://www.w3.org/TR/xpath/;

[37] W3C, *XML Schema Part 0: Primer Second Edition*, 28 October 2004, http://www.w3.org/TR/xmlschema-0

List of Abbreviations

IT	Information Technology
OASIS	Organization for the Advancement of Structured Information Standards
TOSCA	Topology and Orchestration Specification for Cloud Applications
URI	Universal Resource Identifier
URL	Universal Resource Locator
W3C	World Wide Web Consortium
XML	eXtensible Markup Language
XPath	XML Path Language

Index