Finding available services in TOSCA-compliant clouds☆

Antonio Brogi, Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

Abstract

The OASIS TOSCA specification aims at enhancing the portability of cloud applications by defining a language to describe and manage them 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 paper, we define and implement the notions of *exact* and *plug-in matching* between TOSCA service templates and node types. We then define two other types of matching (*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. The paper also describes how a service template that plug-in, flexibly or white-box matches a node type can be suitably adapted so as to exactly match it.

Keywords: service matching, software adaptation, cloud applications, TOSCA

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. Currently, migrating (parts of) an application from one cloud to another is still a costly and error-prone process. 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 [32].

In this scenario, OASIS recently released the *Topology and Orchestration Specification for Cloud Application* (TOSCA [28]), a standard to describe —in a vendor-agnostic way— complex cloud applications and to support the automation of their management. Essentially, TOSCA defines a modelling language that permits specifying the topology and management of an application as a service template that orchestrates typed nodes.

As stated in the TOSCA primer ([29], page 35): *"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 (*"service template ST may substitute node type N because the boundary of ST matches all defining elements of N"*), no formal definition of *matching* is given either in [28] or in [29].

☆This work was partly supported by project EU-FP7-ICT-610531 SeaClouds.

The objective of our work is to contribute to the TOSCA specification by first providing a formal definition of the notion of *exact matching* between TOSCA service templates and node types, 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 (Fig. 1).

exact matching	It mirrors the informal definition of matching in [29] by permitting to match a node type N by a service template S exposing exactly the same features as N on its boundaries.
plug-in matching	It relaxes the exact matching by permitting to match a node type N by a service template S which exhibits less requirements and offers more features on its boundaries.
flexible matching	It relaxes the plug-in matching by permitting to match a node type N by a service template S whose features may be syntactically different but semantically equivalent to those of N (e.g., N's property "CPUs" flexibly —but not plug-in— matches S's property "Central Processing Units").
white-box matching	It relaxes flexible matching by permitting to match the features of a node type N not only with the features on the boundaries of S, but also with those that S owns internally.

Figure 1: A snapshot of the notions of matching introduced in this paper.

In order to show the feasibility of the proposed notions, we include a proof-of-concept implementation of both the exact and the plug-in matching. Moreover, 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. We also provide a pseudo-code to adapt plug-in matched service templates and a methodology to adapt flexibly and white-box matched ones.

The results presented in this paper intend to contribute to the formal definition of TOSCA. The different types of matching defined in this paper can be fruitfully integrated in the TOSCA implementations (e.g., OpenTOSCA [4]) that are currently under development in order to enhance their type-checking capabilities. More in general, the definitions of matching presented in this paper can be exploited to implement type-checking mechanisms over service descriptions by taking into account, beyond functional features, also requirements, capabilities, policies, and properties.

The rest of paper is organised as follows. The main notions of TOSCA are introduced in Sect. 2. The notions of *exact* and *plug-in matching* between a service template and a node type are defined in Sect. 3, which also describes a proof-of-concept implementation of the defined matchings and shows how to adapt plug-in matched services. Two other notions of matching (*flexible* and *white-box*) are introduced, along with the corresponding adaptation techniques, in Sect. 4. Related work is discussed in Sect. 5, while some concluding remarks are drawn in Sect. 6.

2. Background: TOSCA

TOSCA [28] is an OASIS standard aimed at enabling the specification of portable cloud applications and the automation of their management. To do so, TOSCA provides a modelling language to describe the structure of a cloud application as a typed topology graph, and its tasks as plans. More precisely, each cloud application is represented as a `ServiceTemplate` (Fig. 2), consisting of a mandatory `TopologyTemplate` and of optional management `Plans`.

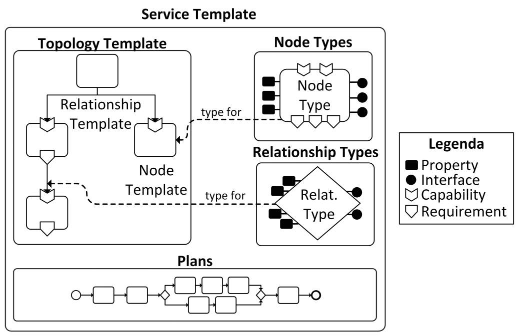


Figure 2: TOSCA `ServiceTemplate`.

The `TopologyTemplate` is a typed directed graph describing the structure of a composite cloud application. Its nodes (`NodeTemplate`s) model the application components, while its edges (`RelationshipTemplate`s) model the relations among those components. `NodeTemplate`s and `RelationshipTemplate`s are typed by means of `NodeType`s and `RelationshipType`s, respectively. A `NodeType` defines the requirements of an application component, the capabilities it offers to satisfy other components' requirements, its observable properties, and its management operations. `RelationshipType`s describe the properties of relationships occurring among components, as well as the operations to manage such relationships. Syntactically, requirements are described by `RequirementDefinitions` (of certain `RequirementType`s), capabilities by `CapabilityDefinition`s (of certain `CapabilityType`s), properties by `PropertiesDefinition`, and operations by `Interface`s and `Operation`s. Requirements, capabilities, properties and operations externally exposed by a `ServiceTemplate` can be described in its `BoundaryDefinitions`.

`NodeTemplate`s and `RelationshipTemplate`s can also declare QoS information by exposing `Policy` elements, which in turn must be typed by referring `PolicyType`s. A `PolicyType` defines the structure of the QoS information (as well as the `NodeType`s to which it is applicable), while a `Policy` assigns it concrete values.

Finally, `Plans` permit describing the management of a `ServiceTemplate`. More precisely, each `Plan` is a workflow orchestrating the `Operation`s offered by the application components to address (part of) the management of the whole cloud application.

A more detailed and self-contained introduction to TOSCA can be found in [13].

3. Matching `ServiceTemplates` with `NodeTypes`

According to the TOSCA primer [29], a `NodeType` can be made concrete by substituting it by a `ServiceTemplate`, provided that they expose the same features on their boundaries. While such matching is mentioned with reference to an example, no definition of *matching* is given either in TOSCA [28] or in its primer [29].

In this section we first formally define when a `ServiceTemplate` can *exactly* match ($\equiv$) a `NodeType`. Then, we formally define the *plug-in* matching ($\simeq$), which relaxes the *exact* one (viz., $\equiv \subset \simeq$) in order to identify larger sets of `ServiceTemplate`s that can be adapted so as to (exactly) match a `NodeType`. Finally, we show a proof-of-concept implementation of the introduced notions of matching.

3.1. Exact matching

In this section we formalize the definition of *exact* matching between a `ServiceTemplate` and a `NodeType`, which mirrors the informal definition of matching mentioned in [28, 29]. The following definition specifies when a `ServiceTemplate` S exactly matches a `NodeType` N in terms of the requirements (Reqs), capabilities (Caps), policies (Pols), properties (Props) and interfaces (Ints) of S and N[1].

Definition 1. *A* `ServiceTemplate` S exactly *matches a* `NodeType` N *($S \equiv N$) iff:*

$$\begin{array}{rrcll}
(1) & \mathsf{Reqs}(S) & \equiv_R & \mathsf{Reqs}(N) & \textit{and} \\
(2) & \mathsf{Caps}(S) & \equiv_C & \mathsf{Caps}(N) & \textit{and} \\
(3) & \mathsf{Pols}(S) & \equiv_{PO} & \mathsf{Pols}(N) & \textit{and} \\
(4) & \mathsf{Props}(S) & \equiv_{PR} & \mathsf{Props}(N) & \textit{and} \\
(5) & \mathsf{Ints}(S) & \equiv_I & \mathsf{Ints}(N). &
\end{array}$$

Before digging into the details of conditions (1—5), we introduce some shorthand notations to retrieve names and types of TOSCA elements.

Notation 1. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*
- $\mathsf{name}(x)$ *denotes the* name *of x, where x can be a requirement, capability, property, interface, operation, or parameter of N or S.*
- $\mathsf{type}(x)$ *denotes the* type *of x, where x can be a requirement, capability, policy, property, or parameter of N or S.*
- $\mathsf{XMLtype}(x)$ *denotes the* XML type *of x, where x can be* $\mathsf{Props}(N)$ *or* $\mathsf{Props}(S)$*.*

We now define the *exact* matching of requirements. Essentially, they must have the same name and type, and they must be in a one-to-one correspondence. The same holds for capabilities.

Definition 2. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*
$\mathsf{Reqs}(S) \equiv_R \mathsf{Reqs}(N)$ *iff*
$\quad \forall r_S \in \mathsf{Reqs}(S)\ \exists! r_N \in \mathsf{Reqs}(N) : \mathsf{name}(r_S) = \mathsf{name}(r_N) \wedge \mathsf{type}(r_S) = \mathsf{type}(r_N)$*, and*

[1] Strictly speaking, the definition relates the `Requirements` exposed by S with the `RequirementDefinitions` of N, the `Capabilities` exposed by S with the `CapabilityDefinitions` of N, the `Policies` exposed by S with the `PolicyTypes` applicable to N, and the `Properties` exposed by S with the `PropertiesDefinition` declared by N.

$\forall r_N \in \mathsf{Reqs}(N)\ \exists! r_S \in \mathsf{Reqs}(S) : \mathsf{name}(r_N) = \mathsf{name}(r_S) \wedge \mathsf{type}(r_N) = \mathsf{type}(r_S)$.
$\mathsf{Caps}(S) \equiv_C \mathsf{Caps}(N)$ *iff*
$\forall c_S \in \mathsf{Caps}(S)\ \exists! c_N \in \mathsf{Caps}(N) : \mathsf{name}(c_S) = \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) = \mathsf{type}(c_N)$, *and*
$\forall c_N \in \mathsf{Caps}(N)\ \exists! c_S \in \mathsf{Caps}(S) : \mathsf{name}(c_N) = \mathsf{name}(c_S) \wedge \mathsf{type}(c_N) = \mathsf{type}(c_S)$.

According to [28], a `PolicyType` can be associated with a set of `NodeType`s to which it is applicable[2]. To ensure exact matching, the type of each `Policy` of S must therefore be one of the `PolicyType`s applicable to N.

Definition 3. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*
$\mathsf{Pols}(S) \equiv_{PO} \mathsf{Pols}(N)$ *iff* $\forall pol_S \in \mathsf{Pols}(S) : \mathsf{type}(pol_S) \in \mathsf{Pols}(N)$.

Furthermore, since a `NodeType` only specifies the XML schema of its observable properties (while `ServiceTemplate`s specify actual values of properties), property matching reduces to comparing XML types.

Definition 4. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*
$\mathsf{Props}(S) \equiv_{PR} \mathsf{Props}(N)$ *iff* $\mathsf{XMLtype}(\mathsf{Props}(S)) = \mathsf{XMLtype}(\mathsf{Props}(N))$.

Finally, interfaces must have the same name and must be in a one-to-one correspondence. The same holds for interface operations and for operation parameters. Operation parameters must also have the same type.

Definition 5. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*
$\mathsf{Ints}(S) \equiv_I \mathsf{Ints}(N)$ *iff*
$\forall i_S \in \mathsf{Ints}(S)\ \exists! i_N \in \mathsf{Ints}(N) : \mathsf{name}(i_S) = \mathsf{name}(i_N) \wedge$
$\qquad \forall o_S \in \mathsf{Ops}(i_S)\ \exists! o_N \in \mathsf{Ops}(i_N) : o_S \equiv_o o_N$ *and*
$\forall i_N \in \mathsf{Ints}(N)\ \exists! i_S \in \mathsf{Ints}(S) : \mathsf{name}(i_N) = \mathsf{name}(i_S) \wedge$
$\qquad \forall o_N \in \mathsf{Ops}(i_N)\ \exists! o_S \in \mathsf{Ops}(i_S) : o_N \equiv_o o_S$
where $\mathsf{Ops}(.)$ *denotes the set of operations of an interface and where*
$o_x \equiv_o o_y$ *iff*
$\mathsf{name}(o_x) = \mathsf{name}(o_y)$ *and*
$\forall a \in \mathsf{I}(o_x), \exists! b \in \mathsf{I}(o_y) : \mathsf{name}(a) = \mathsf{name}(b) \wedge \mathsf{type}(a) = \mathsf{type}(b)$, *and*
$\forall b \in \mathsf{I}(o_y), \exists! a \in \mathsf{I}(o_x) : \mathsf{name}(a) = \mathsf{name}(b) \wedge \mathsf{type}(a) = \mathsf{type}(b)$, *and*
$\forall a \in \mathsf{O}(o_x), \exists! b \in \mathsf{O}(o_y) : \mathsf{name}(a) = \mathsf{name}(b) \wedge \mathsf{type}(a) = \mathsf{type}(b)$, *and*
$\forall b \in \mathsf{O}(o_y), \exists! a \in \mathsf{O}(o_x) : \mathsf{name}(a) = \mathsf{name}(b) \wedge \mathsf{type}(a) = \mathsf{type}(b)$
where $\mathsf{I}(o)$ *and* $\mathsf{O}(o)$ *denote the input and output parameters of operation o.*

It is easy to observe that the notion of exact matching is quite strict, as illustrated by the following example.

Example 1. Consider the `NodeType`s $\mathsf{N_1}$ and $\mathsf{N_2}$ and the `ServiceTemplate` S of Fig. 3, where C and $\mathsf{C_{sup}}$ denote sets of capabilities, R and $\mathsf{R_{sub}}$ denote sets of requirements, $\mathsf{p_j}$ denotes a property, $\mathsf{i_j}$ denotes an interface, $\mathsf{o_j}$ denotes an operation, and where policies and operation parameters are omitted for readability. Suppose that S exactly matches $\mathsf{N_1}$ (viz., $\mathsf{S} \equiv \mathsf{N_1}$) and that $\mathsf{N_2}$ differs from $\mathsf{N_1}$ since it exposes "more" requirements than $\mathsf{N_1}$ and "less" capabilities, properties and operations than $\mathsf{N_1}$. While, according to Defs. 1—5, S cannot exactly match $\mathsf{N_2}$ (viz., $\mathsf{S} \not\equiv \mathsf{N_2}$), a less strict definition of matching should allow S to match also $\mathsf{N_2}$ (as we will discuss in the next section). □

[2] We assume that a `PolicyType` is applicable to all `NodeType`s if not specified otherwise.

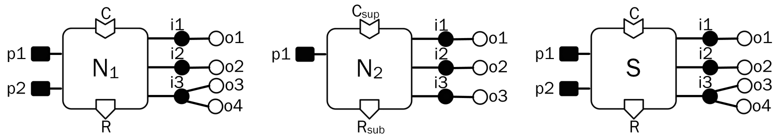


Figure 3: Exact matching examples.

3.2. Plug-in matching

Intuitively speaking, a `ServiceTemplate` plug-in matches a `NodeType` if the former "requires less" and "offers more" than the latter. Similarly to Def. 1, the following definition specifies when a `ServiceTemplate` S can plug-in match a `NodeType` N in terms of the requirements, capabilities, policies, properties and interfaces of S and N. As `NodeType`s do not specify concrete policies (just applicable policies), the matching of policies ($\equiv_{PO}$) is unchanged.

Definition 6. *A* `ServiceTemplate` S plug-in *matches a* NodeType N *(*$S \simeq N$*) iff:*

(1)	$\mathsf{Reqs}(S)$	$\simeq_R$	$\mathsf{Reqs}(N)$	*and*
(2)	$\mathsf{Caps}(S)$	$\simeq_C$	$\mathsf{Caps}(N)$	*and*
(3)	$\mathsf{Pols}(S)$	$\equiv_{PO}$	$\mathsf{Pols}(N)$	*and*
(4)	$\mathsf{Props}(S)$	$\simeq_{PR}$	$\mathsf{Props}(N)$	*and*
(5)	$\mathsf{Ints}(S)$	$\simeq_I$	$\mathsf{Ints}(N)$.	

Intuitively speaking, a `ServiceTemplate` must expose "less" requirements than a `NodeType`. According to [28], names of requirements cannot be different, but types do not need to strictly coincide.

Notation 2. *In the following we write* $t' \geq t$ *when type* t' *extends*[3] *or is equal to* t.

Definition 7. *Let* N *be a* `NodeType` *and let* S *be a* `ServiceTemplate`. *Then:*

$\mathsf{Reqs}(S) \simeq_R \mathsf{Reqs}(N)$ *iff*
$\forall r_S \in \mathsf{Reqs}(S)\ \exists r_N \in \mathsf{Reqs}(N) : \mathsf{name}(r_N) = \mathsf{name}(r_S) \wedge \mathsf{type}(r_N) \geq \mathsf{type}(r_S)$.

Dually, a `ServiceTemplate` must expose "more" capabilities and properties of a `NodeType`. According to [28], names of capabilities cannot be different, but types do not need to strictly coincide.

Definition 8. *Let* N *be a* `NodeType` *and let* S *be a* `ServiceTemplate`. *Then:*

$\mathsf{Caps}(S) \simeq_C \mathsf{Caps}(N)$ *iff*
$\forall c_N \in \mathsf{Caps}(N)\ \exists c_S \in \mathsf{Caps}(S) : \mathsf{name}(c_S) = \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) \geq \mathsf{type}(c_N)$.
$\mathsf{Props}(S) \simeq_{PR} \mathsf{Props}(N)$ *iff* $\mathsf{XMLtype}(\mathsf{Props}(S)) \geq \mathsf{XMLtype}(\mathsf{Props}(N))$.

Finally, a `ServiceTemplate` must expose all the operations exposed by a `NodeType`. The matching can focus on operations and abstract from (names of) interfaces.

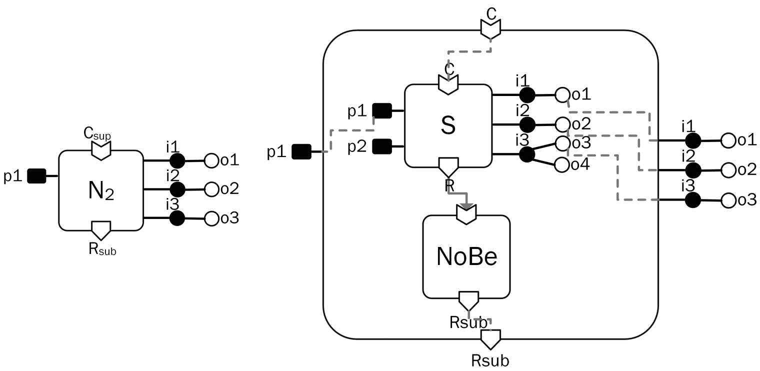
[3]More precisely, if t and t' are TOSCA elements then t' extends t if t' is (directly or undirectly) `DerivedFrom` t. If t and t' are instead XML types then the standard notion of XML extension applies.

Definition 9. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*

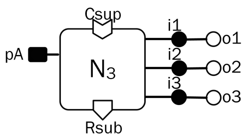
$$\mathsf{Ints}(S) \simeq_I \mathsf{Ints}(N) \text{ iff } \forall i_N, o_N : i_N \in \mathsf{Ints}(N) \wedge o_N \in \mathsf{Ops}(i_N)$$
$$\exists i_S, o_S : i_S \in \mathsf{Ints}(S) \wedge o_S \in \mathsf{Ops}(i_S) : \ o_S \equiv_o o_N .$$

It is worth noting that when a `ServiceTemplate` S plug-in matches a `NodeType` then S can be easily adapted into a new `ServiceTemplate` S' that exactly matches that `NodeType`. Such S' is built by creating a new `ServiceTemplate` having S as its only node, and by simply exposing (via the `BoundaryDefinitions`) the capabilities, policies, properties, and interfaces of the `NodeType` to be matched. If requirements plug-in match (but do not exactly match) then a dummy NoBe `NodeTemplate` is introduced to artificially extend the set of requirements of S so as to expose the same requirements of the `NodeType` to be matched.

Example 2. Example 1 illustrated a `ServiceTemplate` S that cannot exactly match a `NodeType` $\mathsf{N_2}$ since the latter exposes "more" requirements and "less" capabilities, properties and operations than the former. Since S exposes one property (p2) and one operation (o4) more than $\mathsf{N_2}$, we have that $\mathsf{Props(S)} \simeq_{PR} \mathsf{Props(N_2)}$ and $\mathsf{Ints(S)} \simeq_{PR} \mathsf{Ints(N_2)}$ by Defs. 8 and 9, respectively. Therefore, if $\mathsf{R} \simeq_R \mathsf{R_{sub}}$ and $\mathsf{C} \simeq_R \mathsf{C_{sup}}$ hold too, then S plug-in matches $\mathsf{N_2}$ ($\mathsf{S} \simeq \mathsf{N_2}$). Fig. 4.(a) illustrates how S can be adapted to exactly match $\mathsf{N_2}$.



(a)



(b)

Figure 4: Plug-in matching examples.

Consider now the `NodeType` $\mathsf{N_3}$ in Fig. 4.(b), which differs from $\mathsf{N_2}$ only since it exposes property pA instead of property p1. According to Def. 8, S cannot plug-in match $\mathsf{N_3}$ ($\mathsf{S} \not\simeq \mathsf{N_3}$). However, if p1 and pA were (syntactically) different names for the same property and if the type of p1 were compatible with the type of pA (i.e., $\mathsf{type(p1)} \geq \mathsf{type(pA)}$), then a less strict definition of matching should allow S to match also $\mathsf{N_3}$ (as we will discuss in Sect. 4). $\square$

3.3. Proof-of-concept implementation of exact and plug-in matching

The definitions of matching presented in this paper can be fruitfully integrated in the OpenTOSCA open source environment [4], as well as in other TOSCA implementations currently under development, in order to enhance their type-checking capabilities. Since current TOSCA implementations are all written in Java, we shall now describe a proof-of-concept Java implementation[4] of both *exact* and *plug-in* matchings.

3.3.1. High-level modeling of TOSCA

The OpenTOSCA environment directly exploits TOSCA XSD [30] to automatically generate the Java representation of TOSCA files. The obtained representation is quite low-level and it does not ease the development of integrated plug-ins. For example, consider the management of the relationships between capabilities, capability types and capability definitions. In TOSCA both `Capabilities` and `CapabilityDefinitions` can reference `CapabilityType`s by means of *QNames*. To avoid that the automated XSD-based conversion of TOSCA `Capabilities` and `CapabilityDefinitions` looses such references, ad-hoc mechanisms must be developed to generate an explicit representation of such references that associates *QNames* with the corresponding Java classes.

Since OpenTOSCA and Winery do not provide a high-level API to manage TOSCA elements [3], we will employ a higher level Java representation of TOSCA elements, which is still a hierarchy of classes that corresponds to the hierarchy of elements defined in the TOSCA XSD. For instance, the management of `Capabilities`, `CapabilityTypes` and `CapabilityDefinitions` is simply performed by directly referring the relative Java objects (Fig. 5). Thanks to its schema definition orientedness, such higher level representation can be easily mapped on the lower level representations currently employed by the available TOSCA implementations[5].

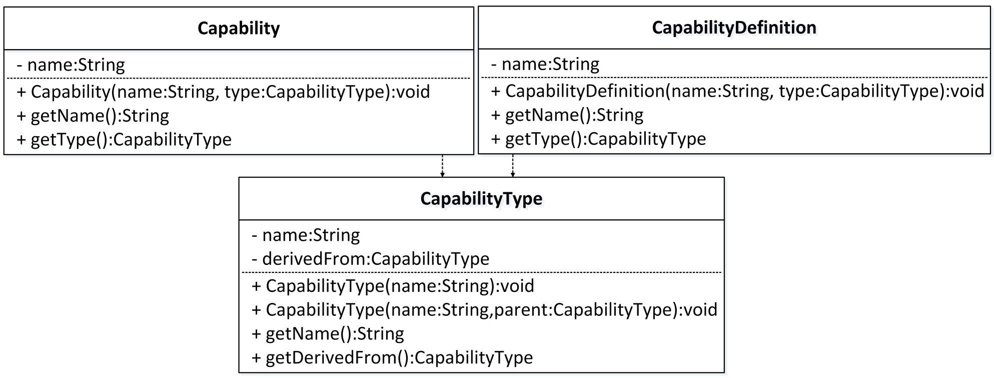


Figure 5: High-level management of TOSCA capabilities.

[4]The source code is publicly available on GitHub at `https://github.com/jacopogiallo/Finding-available-services-in-TOSCA-compliant-clouds`.

[5]The documentation of the higher level API is available at `http://jacopogiallo.github.io/Finding-available-services-in-TOSCA-compliant-clouds/`.

3.3.2. Implementation of the matchmakers

Since *plug-in* matching generalizes *exact* matching (viz., $\equiv\subset\simeq$), we implemented the two matchings as a class hierarchy. The top element of such hierarchy will be the abstract `Matchmaker` class. It groups the fields and methods common to both the *exact* and the *plug-in* matchmakers. More precisely, it declares the `ServiceTemplate s` and the `NodeType n` to be matched, and the sets of unmatched elements (e.g., `unmatchedCapabilities`). It also provides the constructor method, as well as the abstract methods to check whether `s` `match`es `n` and to access the above mentioned sets of unmatched elements (e.g., `getUnmatchedCapabilities`).

The abstract `Matchmaker` class is then extended to provide the implementation of the *exact* matchmaker. The resulting `ExactMatchmaker` suitably stores the exactly matched TOSCA elements (e.g., `exactlyMatchedCapabilities`) and provides access to them (e.g., `getExactlyMatchedCapabilities`). It also implements the `match` method by

```
01  public boolean match() {
02     matchCapabilities();
03     matchRequirements();
04     matchPolicies();
05     matchProperties();
06     matchInterfaces();
07     return (areCapabilitiesMatched && areRequirementsMatched && arePoliciesMatched &&
08          arePropertiesMatched && areInterfacesMatched);
09  }
```

Figure 6: `ExactMatchmaker.match()` method.

checking whether a `ServiceTemplate s` *exactly* matches a `NodeType n` (Fig. 6). According to Def. 1, the matching is performed in a step-wise way (lines 2-6). Each kind of element is matched with a separate method (e.g., `matchCapabilities`) which properly instantiates the corresponding boolean variable (e.g., `areCapabilitiesMatched`). The result of the whole matchmaking is the logical *and* among all sub-results (lines 7-8).

Consider, for instance, the matchmaking of capabilities[6] (Fig. 7). After initialization (lines 2-7), the method checks whether all the capabilities defined by the `NodeType` are present on the boundaries of `s`. More precisely, for each `CapabilityDefinition` in `n` (line 8) it checks whether there exists a `Capability` on the boundaries of `s` such that they *exactly* match (lines 10-16). The comparison is performed by the `match` method which checks whether a `CapabilityDefinition` and a `Capability` have same name and type (lines 24-27). If no capability matches the capability definition under consideration, then the latter is added to a new set of unmatched `CapabilityDefinitions` (line 17). After the end of the loop, the set of `unmatchedCapabilities` is updated (line 19). Then, to ensure the one-to-one correspondence needed by Def. 2, the method checks whether both `n` and `s` expose the same number of capabilities (line 20). If so, and if there are no unmatched `CapabilityDefinitions`, then the `areCapabilitiesMatched` variable is set to `true` (line 21). Otherwise, the method ends (by leaving it set to `false`).

The `ExactMatchmaker` is in turn extended by the `PlugInMatchmaker`. The latter stores and provides access to the plug-in matched TOSCA elements (e.g., via the field

[6] The (exact) matchmaking of the other TOSCA elements is analogous.

```
protected void matchCapabilities() {
    exactlyMatchedCapabilities = new ArrayList<Capability>();
    areCapabilitiesMatched = false;
    unmatchedCapabilities = n.getCapabilityDefinitions().getList();
    List<Capability> sCaps = s.getBoundaryDefinitions().getCapabilities().getList();
    List<CapabilityDefinition> newUnmatchedCapabilities =  new ArrayList<CapabilityDefinition>();
    boolean matched;
    for(CapabilityDefinition cDef : unmatchedCapabilities) {
         matched = false;
         for(Capability c : sCaps) {
              matched = match(cDef, c);
              if(matched) {
                   exactlyMatchedCapabilities.add(c);
                   break;
              }
         }
         if(!matched) newUnmatchedCapabilities.add(cDef);
    }
    unmatchedCapabilities = newUnmatchedCapabilities;
    if(n.getCapabilityDefinitions().getList().size() != sCaps.size()) return;
    if(unmatchedCapabilities.isEmpty()) areCapabilitiesMatched = true;
}

protected boolean match(CapabilityDefinition cDef, Capability c) {
    return (cDef.getName().equals(c.getName()) &&
         cDef.getCapabilityType().getName().equals(c.getType().getName()));
}
```

Figure 7: `ExactMatchmaker.matchCapabilities()` method.

`plugInMatchedCapabilities` and the method `getPlugInMatchedCapabilities`) and overrides the `match` method by making it check whether a `NodeType n` *plug-in* matches a `ServiceTemplate s` (Fig. 8). The method starts by checking whether the two elements *exactly* match (lines 2-8). If this is not the case, the *plug-in* matching of (unmatched) TOSCA elements is performed separately (lines 10-13). Finally, the whole matchmaking result is computed with the logical *and* among all partial results (lines 14-15).

```
01 public boolean match() {
02     super.matchCapabilities();
03     super.matchRequirements();
04     super.matchPolicies();
05     super.matchProperties();
06     super.matchInterfaces();
07     if(areCapabilitiesMatched && areRequirementsMatched && arePoliciesMatched
08          && arePropertiesMatched && areInterfacesMatched) return true;
09
10     if(!areCapabilitiesMatched) matchCapabilities();
11     if(!areRequirementsMatched) matchRequirements();
12     if(!arePropertiesMatched) matchProperties();
13     if(!areInterfacesMatched) matchInterfaces();
14     return (areCapabilitiesMatched && areRequirementsMatched && arePoliciesMatched &&
15          arePropertiesMatched && areInterfacesMatched);
16 }
```

Figure 8: `PlugInMatchmaker.match()` method.

Consider, for instance, the matchmaking of capabilities[7] (Fig. 9). Since it is performed after the *exact* matching, the set up of the environment is lighter than that of

[7]The (*plug-in*) matchmaking of the other TOSCA elements is analogous.

```
01  protected void matchCapabilities() {
02     pluginMatchedCapabilities = new ArrayList<Capability>();
03     List<Capability> sCaps = s.getBoundaryDefinitions().getCapabilities().getList();
04     List<CapabilityDefinition> newUnmatchedCapabilities = new ArrayList<CapabilityDefinition>();
05     boolean matched;
06     for(CapabilityDefinition cDef : unmatchedCapabilities) {
07          matched = false;
08          for(Capability c : sCaps) {
19               matched = match(cDef, c);
10               if(matched) {
11                    pluginMatchedCapabilities.add(c);
12                    break;
13               }
14          }
15          if(!matched) newUnmatchedCapabilities.add(cDef);
16     }
17     unmatchedCapabilities = newUnmatchedCapabilities;
18     if(unmatchedCapabilities.isEmpty()) areCapabilitiesMatched = true;
19  }
20
21  protected boolean match(CapabilityDefinition cDef, Capability c) {
22     if(!cDef.getName().equals(c.getName())) return false;
23     CapabilityType cType = c.getType();
24     while(cType != null) {
25          if(cDef.getCapabilityType().getName().equals(cType.getName())) return true;
26          cType = cType.derivedFrom();
27     }
28     return false;
29  }
```

Figure 9: `PlugInMatchmaker.matchCapabilities()` method.

Fig. 7 (lines 2-5). The method then proceeds by checking whether all the capabilities defined by `n` are compatible with those on the boundaries of `s`. More precisely, for each `CapabilityDefinition` in `n` (that has not yet been matched — line 6), it checks whether there exists a `Capability` on the boundaries of `s` such that they *plug-in* match (lines 7-14). The comparison is performed by the `match` method (lines 21-29) which checks whether a `Capability c` has the same name as `CapabilityDefinition cDef` (line 22) and whether `c` either has the same type of or is derived from `cDef` (lines 23-28). If no capability matches the capability definition under consideration, then the latter is added to the (new) set of unmatched capability definitions (line 15). After the end of the loop, the set of `unmatchedCapabilities` is properly updated (line 17). If there are no unmatched capability definitions, then the `areCapabilitiesMatched` variable is set to `true` (line 18). Otherwise, the method terminates (by leaving it set to `false`).

Example 3. We now use a (toy) example to illustrate the behaviour of our proof-of-concept implementation. Consider the `NodeType` Server and the `ServiceTemplates`[8] ApacheServer, PaaS-Server, and PaaSServer2 in Fig. 10. Suppose that the `Capability` WSRuntime of Server and ApacheServer is of WSRuntimeCapabilityType, while those of PaaSServer and PaaSServer2 are of WebAppCapabilityType (which is a sub-type of WSRuntimeCapabilityType). Suppose also that the type of all requirements is SWContainerRequirementType, the type of all properties is `String`, and all `ServiceTemplate`s expose a HighAvailabilityPolicy which is applicable to Server.

Please note that the example is built in such a way that, according to Defs. 1 and 6, all possible situations are covered:

$$\mathsf{ApacheServer} \equiv \mathsf{Server} \wedge \mathsf{PaaSServer} \not\equiv \mathsf{Server} \wedge \mathsf{PaaSServer} \simeq \mathsf{Server} \wedge \mathsf{PaaSServer2} \not\simeq \mathsf{Server}.$$

[8]For the sake of readability we abstract from the internal structure of the `ServiceTemplates`.

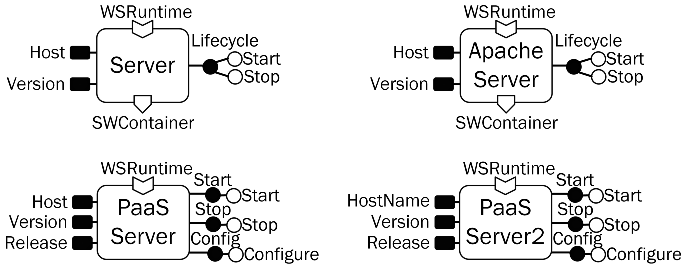


Figure 10: A `NodeType` (Server) and three `ServiceTemplates` (ApacheServer, PaaSServer, PaaSServer2).

We can easily develop a test class[9] which let us obtain the above mentioned results (Fig. 11) by employing the `ExactMatchmaker` and `PlugInMatchmaker` previously introduced. ☐

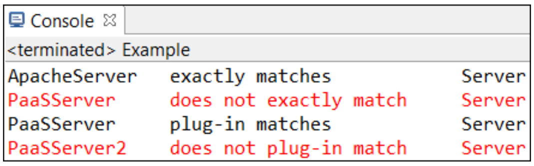


Figure 11: Snapshot of the matchmaking results.

3.3.3. Further remarks

Thanks to the way in which the `match()` methods were implemented (Figs. 6 and 8), the `PlugInMatchmaker` can be directly exploited to determine whether a `ServiceTemplate` *exactly* or *plug-in* matches a `NodeType`. Suppose for instance that `areCapabilitiesMatched` is `true`. If `plugInMatchedCapabilities` is empty, then capabilities were *exactly* matched. Otherwise, they were *plug-in* matched. The same holds for requirements, policies, properties, and interface operations.

The information in the fields of `PlugInMatchmaker` can also be employed to automate the adaptation of a matched `ServiceTemplate`. Fig. 12 shows the pseudo-code of a method to be included in the `PlugInMatchmaker` class in order to automatically adapt a `ServiceTemplate s` that it *plug-in* matches the `NodeType n`.

The proposed implementation can be fruitfully employed by the user also in case of no matching. Since the matchmaking is performed in a "verbose" way (i.e., instead of halting when a condition is not satisfied, it always checks all conditions and properly instantiates the corresponding fields), the collected information can be fruitfully exploited to manually adapt an available `ServiceTemplate`. In this respect, a methodology to manually adapt unmatched services (if possible) is described in [12].

[9] The source code of the example is available at `https://github.com/jacopogiallo/Finding-available-services-in-TOSCA-compliant-clouds/blob/master/src/di/unipi/example/Example.java`.

```
ServiceTemplate getAdaptation() {
  //Creation of the adapted ServiceTemplate
  Create the ServiceTemplate adapted;
  Add s to the topology of adapted;

  //Adaptation of the capabilities
  For each Capability c in exactMatchedCapabilities
    Expose c on the boundaries of adapted;
  For each Capability c in plugInMatchedCapabilities
    Expose c on the boundaries of adapted;

  //Adaptation of the requirements
  If plugInMatchedRequirements is empty
    For each Requirement r in exactMatchedRequirements
      Expose r on the boundaries of adapted;
  Else
    Create the (no-behaviour) NodeTemplate echo;
    Add echo to the topology of adapted;
    Create the relationship relEcho from s to echo;
    Add relEcho to the topology of adapted;
    For each Requirement r in n.getRequirements()
      Add r to the requirements of echo;
      Expose r on the boundaries of adapted;

  //Adaptation of the policies
  For each Policy pol in exactMatchedPolicies
    Expose pol on the boundaries of adapted;

  //Adaptation of the properties
  For each Property prop in exactMatchedProperties
    Expose prop on the boundaries of adapted;
  For each Property prop in plugInMatchedProperties
    Expose prop on the boundaries of adapted;

  //Adaptation of the interfaces
  For each Interface inf in exactMatchedInterfaces
    Expose inf on the boundaries of adapted;
  For each Interface inf in plugInMatchedCapabilities
    Expose inf on the boundaries of adapted;

  return adapted;
}
```

Figure 12: Pseudo-code of the adaptation of *plug-in* matched services.

4. Overcoming syntactic differences

Example 2 illustrated that a `ServiceTemplate` S may fail to plug-in match a `NodeType` N only because of syntactically different names for compatible features, while a less strict definition of matching should allow S to match also N. We now define two other types of matching (*flexible* and *white-box*), each permitting to ignore larger sets of non-relevant syntactic differences when type-checking `ServiceTemplate`s with respect to `NodeType`s. Finally, we show how to avoid the usage of ontologies by providing a methodology for adapting unmatched *plug-in* `ServiceTemplate`s which is based upon the notions of *flexible* and *white-box* matching.

4.1. Flexible matching

We now further extend the definition of matching of a `ServiceTemplate` with a `NodeType` in order to ignore non-relevant syntactic differences between names of features (i.e., by permitting to match features whose names are syntactically different, but semantically equivalent). Since the semantics of policies depends only on types, we extend plug-in matching (Def. 6) only on capabilities, requirements, properties and interfaces.

Definition 10. *A* `ServiceTemplate` S flexibly *matches a* `NodeType` N *(*$S \sim N$*) iff:*

$$\begin{array}{llcll} (1) & \mathsf{Reqs}(S) & \sim_R & \mathsf{Reqs}(N) & \textit{and} \\ (2) & \mathsf{Caps}(S) & \sim_C & \mathsf{Caps}(N) & \textit{and} \\ (3) & \mathsf{Pols}(S) & \equiv_{PO} & \mathsf{Pols}(N) & \textit{and} \\ (4) & \mathsf{Props}(S) & \sim_{PR} & \mathsf{Props}(N) & \textit{and} \\ (5) & \mathsf{Ints}(S) & \sim_I & \mathsf{Ints}(N). & \end{array}$$

Intuitively speaking, a `ServiceTemplate` must expose "less" requirements than a `NodeType`. Names of requirements can be semantically equivalent, and types of requirements do not need to strictly coincide.

Notation 3. *Let n_1 and n_2 be the names of two TOSCA definitions. We will write $n_1 \bowtie n_2$ to denote that names n_1 and n_2 are semantically equivalent*[10].

Definition 11. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*

$\mathsf{Reqs}(S) \sim_R \mathsf{Reqs}(N)$ *iff*
$\forall r_S \in \mathsf{Reqs}(S)\ \exists r_N \in \mathsf{Reqs}(N) : \mathsf{name}(r_N) \bowtie \mathsf{name}(r_S) \wedge \mathsf{type}(r_N) \geq \mathsf{type}(r_S)$.

A `ServiceTemplate` must expose all capabilities of a `NodeType`. Names of capabilities can be semantically equivalent, and types of capabilities do not need to stricly coincide. The same holds for properties.

Definition 12. *Let N be a* `NodeType` *and S a* `ServiceTemplate`*. Then:*

[10] The semantical equivalence of syntactically different names may be implemented by employing ontology-based descriptions of cloud service functionalities (e.g., [31]). Namely, TOSCA `NodeTypes` and `ServiceTemplates` may include ontology-based annotations associated with the names of their capabilities, requirements, properties and operations. Instead of assuming that all TOSCA cloud service descriptions are ontology-annotated, we will describe (Sect. 4.3) an ontology-free methodology for adapting a `ServiceTemplate` S that flexibly or white-box matches a `NodeType` N so as to match N.

$\mathsf{Caps}(S) \sim_C \mathsf{Caps}(N)$ *iff*
$\quad\forall c_N \in \mathsf{Caps}(N)\ \exists c_S \in \mathsf{Caps}(S) : \mathsf{name}(c_S) \bowtie \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) \geq \mathsf{type}(c_N).$
$\mathsf{Props}(S) \sim_{PR} \mathsf{Props}(N)$ *iff*
$\quad\forall p_N \in \mathsf{Props}(N)\ \exists p_S \in \mathsf{Props}(S)\text{:}\ \mathsf{name}(p_S) \bowtie \mathsf{name}(p_N)\ \wedge\ \mathsf{type}(p_S) \geq \mathsf{type}(p_N).$

A `ServiceTemplate` must also expose all the operations exposed by a `NodeType`. Names of operations can be ignored, while names of operation parameters can be semantically equivalent and their types do not need to strictly coincide.

Definition 13. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*
$\mathsf{Ints}(S) \sim_I \mathsf{Ints}(N)$ *iff* $\forall i_N, o_N : i_N \in \mathsf{Ints}(N) \wedge o_N \in \mathsf{Ops}(i_N)$
$\qquad\exists i_S, o_S : i_S \in \mathsf{Ints}(S) \wedge o_S \in \mathsf{Ops}(i_S) :\ o_S \sim_o o_N.$
where $o_x \sim_o o_y$ *iff*
$\quad|\mathsf{I}(o_x)| = |\mathsf{I}(o_y)|$ *and*
$\quad|\mathsf{O}(o_x)| = |\mathsf{O}(o_y)|$ *and*
$\quad\forall a \in \mathsf{I}(o_x), \exists! b \in \mathsf{I}(o_y) : \mathsf{name}(a) \bowtie \mathsf{name}(b) \wedge \mathsf{type}(b) \geq \mathsf{type}(a)$ *and*
$\quad\forall b \in \mathsf{O}(o_y), \exists! a \in \mathsf{O}(o_x) : \mathsf{name}(a) \bowtie \mathsf{name}(b) \wedge \mathsf{type}(a) \geq \mathsf{type}(b).$

In Sect. 3.2 we illustrated how a `ServiceTemplate` S that plug-in matches a `NodeType` can be easily adapted so as to exactly match that `NodeType`. The same holds for flexible matching: A `ServiceTemplate` S that fexibly matches a `NodeType` can be easily adapted into a new `ServiceTemplate` S' that exactly matches that `NodeType`. As for the case of plug-in matching, S' is built by creating a new `ServiceTemplate` having S as its only node, and by simply exposing (via the `BoundaryDefinitions`) the capabilities, policies, properties, and interfaces of the `NodeType` to be matched. If requirements flexibly match (but do not exactly match) then a dummy *NoBe* node is introduced to artificially extend the set of requirements of S so as to expose the same requirements of `NodeType` to be matched. Moreover, differently from plug-in adaptation, flexible adaptation may rename properties, interfaces, operations, and operation parameters.

Example 4. Example 2 illustrated a `ServiceTemplate` S that does not plug-in match a `Node-Type` N_3 since S exposes a property `p1` different from the property `pA` exposed by N_3. It is easy

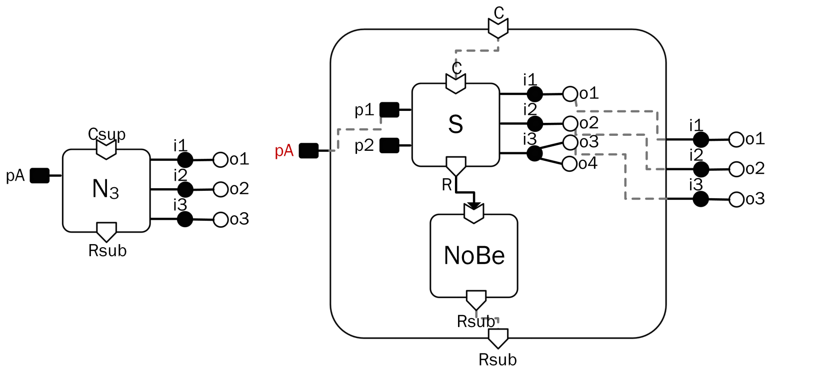


Figure 13: Flexible matching example.

to see that Def. 12 permits S to flexibly match N_3 (viz., $\mathsf{S} \sim \mathsf{N}_3$) if the type of `p1` extends or is equal to the type of `pA` and if `p1` and `pA` —even if syntactically different— refer to the same property (viz., $\mathsf{name}(\texttt{p1}) \bowtie \mathsf{name}(\texttt{pA})$). Fig. 13 illustrates how S can be adapted so as to exactly match N_3, by letting the new `ServiceTemplate` S′ expose also the renamed property `pA`. □

Example 5. Suppose that a cloud application developer needs to employ a `NodeType` OS (Fig. 14), whose management interface M exposes the following operations

$$\mathsf{Start} : \{\} \to \{\},\ \mathsf{InstallPkg} : \{\mathsf{name}\} \to \{\mathsf{succeded}\}, \text{ and } \mathsf{Shutdown} : \{\} \to \{\}.$$

Suppose also that a `ServiceTemplate` UbuntuOS is available, and that it exhibits a management interface U featuring the following operations:

$$\mathsf{Start} : \{\} \to \{\},\ \mathsf{Shutdown} : \{\} \to \{\},\ \mathsf{Retrieve} : \{\mathsf{pkgName}\} \to \{\mathsf{url}\},$$
$$\mathsf{Download} : \{\mathsf{url}\} \to \{\mathsf{sourcePath}\}, \text{ and } \mathsf{Install} : \{\mathsf{sourcePath}\} \to \{\mathsf{installed}\},$$

with $\mathsf{name} \bowtie \mathsf{pkgName}$ and $\mathsf{succeded} \bowtie \mathsf{installed}$. For the sake of simplicity we also assume that $\mathsf{name}(x) \bowtie \mathsf{name}(y)$ implies $\mathsf{type}(x) = \mathsf{type}(y)$.

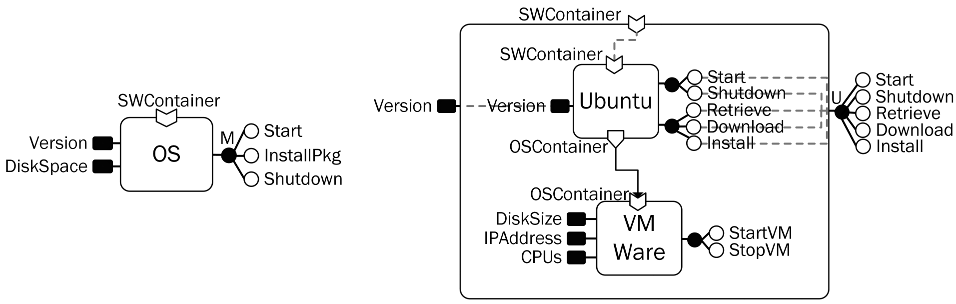


Figure 14: `ServiceTemplate` that cannot flexibly match a `NodeType`.

It is easy to see that while UbuntuOS capabilities exactly match OS capabilties, UbuntuOS properties and interfaces cannot flexibly match OS's ones. This is because OS is exposing a property (DiskSpace) that UbuntuOS is not featuring, and because UbuntuOS does not offer operation InstallPkg exposed by OS. Still, one may observe that property DiskSpace may correspond to one of the properties of an internal node of UbuntuOS (i.e., to VMWare's property DiskSize) and that operation getTemp might be offered by UbuntuOS by suitably combing some of its operations. This suggests that a "white-box" definition of matching could allow the `ServiceTemplate` UbuntuOS to match the desired `NodeTypeOS` (as we will discuss in the next section). □

4.2. White-box matching

A `ServiceTemplate` S that does not flexibly match a `NodeType` because of some missing requirement, capability, property, or operation, may actually include such missing elements internally, without exposing them on its boundaries.

As for the previous definitions of matching, the following definition specifies when a `ServiceTemplate` S *white-box* matches a `NodeType` N in terms of the requirements, capabilities, policies, properties and interfaces of S and N. As we already observed in Sect. 4.1, intuitively speaking, a `NodeType` N must expose (at least) a set of requirements which are semantically equivalent to (all) those of the *ServiceTemplate* S. Moreover, `NodeTypes` do not specify concrete policies. For these reasons, the following definition extends Def. 10 only on capabilities, properties and interfaces.

Definition 14. *A* `ServiceTemplate` S white-box *matches a* `NodeType` N *($S \Box N$) iff:*

$$
\begin{array}{llcll}
(1) & \mathsf{Reqs}(S) & \sim_R & \mathsf{Reqs}(N) & \textit{and} \\
(2) & \mathsf{Caps}(S) & \Box_C & \mathsf{Caps}(N) & \textit{and} \\
(3) & \mathsf{Pols}(S) & \equiv_{PO} & \mathsf{Pols}(N) & \textit{and} \\
(4) & \mathsf{Props}(S) & \Box_{PR} & \mathsf{Props}(N) & \textit{and} \\
(5) & \mathsf{Ints}(S) & \Box_I & \mathsf{Ints}(N). &
\end{array}
$$

The following definition extends the matching of capabilities and properties (Defs. 8 and 12) to consider also the internal nodes of a `ServiceTemplate`.

Notation 4. *Let S be a* `ServiceTemplate`*, and let E be a* `NodeTemplate` *or a* `RelationshipTemplate`*. We denote by $S \to E$ the fact that E is an element of the (internal) topology of S.*

Definition 15. *Let N be a* `NodeType` *and let S be a* `ServiceTemplate`*. Then:*

$$
\begin{array}{l}
\mathsf{Caps}(S) \ \Box_C \ \mathsf{Caps}(N) \ \textit{iff} \ \forall c_N \in \mathsf{Caps}(N) \ \exists c_S : \\
\qquad (c_S \in \mathsf{Caps}(S) \\
\qquad \vee \\
\qquad (\exists E : S \to E \wedge E \ \textit{is} \ \texttt{NodeTemplate} \wedge c_S \in \mathsf{Caps}(E))) \\
\quad \wedge \\
\quad (\mathsf{name}(c_S) \bowtie \mathsf{name}(c_N) \wedge \mathsf{type}(c_S) \geq \mathsf{type}(c_N)).
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Props}(S) \ \Box_{PR} \ \mathsf{Props}(N) \ \textit{iff} \ \forall p_N \in \mathsf{Props}(N) \ \exists p_S : \\
\qquad (p_S \in \mathsf{Props}(S) \\
\qquad \vee \\
\qquad (\exists E : S \to E \wedge (E \ \textit{is} \ \texttt{NodeTemplate} \ \textit{or} \ \texttt{RelationshipTemplate}) \\
\qquad\qquad\qquad \wedge \ p_S \in \mathsf{Props}(E))) \\
\quad \wedge \\
\quad (\mathsf{name}(p_S) \bowtie \mathsf{name}(p_N) \wedge \mathsf{type}(p_S) \geq \mathsf{type}(p_N)).
\end{array}
$$

The following definition extends the matching of operations (Def. 13) to consider also operations that a `ServiceTemplate` can feature by combining its operations in a suitable plan.

Definition 16. *Let N be a* `NodeType`*, let S be a* `ServiceTemplate`*, and let $\Pi(S)$ the set of all possible plans combining S operations. Then:*

$$
\begin{array}{l}
\mathsf{Ints}(S) \ \Box_I \ \mathsf{Ints}(N) \ \textit{iff} \ \forall i_N, o_N : i_N \in \mathsf{Ints}(N) \wedge o_N \in \mathsf{Ops}(i_N): \\
\quad (\exists i_S, o_S : i_S \in \mathsf{Ints}(S) \wedge o_S \in \mathsf{Ops}(i_S) \wedge o_S \sim_o o_N) \\
\quad \vee \\
\quad (\exists p : p \in \Pi(S) \wedge [p] \sim_o o_N)
\end{array}
$$

where $[p]$ is the operation modelling the overall input-output behaviour of plan p.

The existence of a plan that suitably combines a set of operations into an input-output behaviour equivalent to a given operation can be determined by adapting the (ontology-aware) discovery algorithm in [9].

The *FindOperations* algorithm (Fig. 15), given a set of available operations *Ops*, returns a set of *selectedOperations* $\subseteq$ *Ops* that can be composed into a plan featuring the input-output behaviour of a given operation *op*. The algorithm inputs a set of

```
   FindOperations(Ops, op, selectedOperations, needed, available) {
1    needed = {x | x ∈ needed ∧ ∄y ∈ available : y ▷ x};
2    if needed = ∅
3      then return selectedOperations;
4      else {
5        c = choose(needed);
6        needed = needed \ {c};
7        opSet = {o ∈ Ops | ∃d ∈ O(o) : d ▷ c};
8        if opSet = ∅
9          then fail;
10         else foreach o ∈ opSet do {
11           selectedOperations = selectedOperations ∪ {o};
12           if nonMinimal(selectedOperations, op)
13             then fail;
14             else {
15               available = available ∪ O(o);
16               needed = needed ∪ I(o);
17               FindOperations(Ops, op, selectedOperations, needed, available)
18             }
19         }
20     }
21 }
```

Figure 15: Algorithm to discover sets of operations that can be composed into plans featuring the input-output behaviour of a given operation.

available operations *Ops*, the operation *op* to be simulated, a (initially empty) set of *selectedOperations*, the set *needed* of outputs to be generated (initially the outputs $\mathsf{O}(op)$ of *op*), and the set of *available* outputs (initially the inputs $\mathsf{I}(op)$ of *op*). First, if the set of *available* outputs includes an output "equal to or more general than" some *needed* output z, then z is removed from the set of *needed* outputs (line 1). The notation $y \triangleright x$ stands for $\mathsf{name}(y) \bowtie \mathsf{name}(x)$ and $\mathsf{type}(y) \geq \mathsf{type}(x)$. Then, if there are no missing outputs to be generated the current set of *selectedOperations* is returned (lines 2-3). Otherwise, a missing output c is nondeterministically chosen[11] and removed from the set of missing outputs (lines 5 and 6). The algorithm then checks (lines 7 and 8) whether there is at least one operation in *Ops* that produces an output equal to or more general than c. If there is no such operation then the current instance of the algorithm fails (line 9). Otherwise, for each operation o in *Ops* producing an output equal to or more general than c, o is added to the current set of *selectedOperations* (line 11). If the obtained set of *selectedOperations* is not minimal[12] then (the instance of) the algorithm fails (lines 12 and 13). Otherwise the set of *available* outputs is extended with the outputs of o (line

[11] Execution of **choose** forks a new instance of the algorithm for each possible choice.

[12] Because of space limitations, we do not include here the definition of the *nonMinimal* function, which can be found in [9]. Following [9], a set S of operations can simulate the input-output behaviour of an operation *op* iff (1) $\forall x \in \mathsf{O}(op) \exists y \in \bigcup_{o \in S} \mathsf{O}(o) : y \triangleright x$, and (2) $\forall y \in \bigcup_{o \in S} \mathsf{I}(o) \exists x \in (\bigcup_{o \in S} \mathsf{O}(o) \cup \mathsf{I}(op)) : x \triangleright y$. A set S of operations that can emulate an operation *op* is minimal iff $\nexists S' \subset S$ that can emulate *op*.

15), and the set of *needed* outputs is extended (line 16) with the inputs of o. Finally, the algorithm recurs (line 17) on the new set of *selectedOperations*, and of *needed* and *available* outputs.

Finally, it is worth highlighting that when a `ServiceTemplate` S white-box matches a `NodeType` N then S can be adapted into a new `ServiceTemplate` S' that exactly matches that `NodeType`. Differently from the cases of plug-in and flexible matching, the `BoundaryDefinitions` of S are first extended in order to expose the capabilities, properties or plans internal to S that were detected by the white-box matching. The obtained `ServiceTemplate` S_{tmp} flexibly matches `NodeType` N, and the adaptation described in Sect. 4.1 can be now applied to build a `ServiceTemplate` S' having S_{tmp} as its only node, and by simply exposing (via the `BoundaryDefinitions`) the capabilities, policies, properties, and interfaces of the `NodeType` N to be matched. If requirements plug-in match (but do not exactly match) then a dummy node is introduced to artificially extend the set of requirements of S so as to expose the same requirements of the `NodeType` to be matched.

Example 6. Example 5 illustrated a `ServiceTemplate` UbuntuOS) that cannot flexibly match a `NodeType`OS since the latter exposes one property more than the former (i.e., DiskSpace), and since UbuntuOS does not offer the operation InstallPkg. We observe that Def. 14 permits UbuntuOS to white-box match OS (viz., UbuntuOS□OS) if, for instance, property DiskSize of node VMWare of UbuntuOS is semantically equivalent to property DiskSpace of OS, and if there exists a plan P combining some UbuntuOS's operations, whose input-output behaviour simulates operation InstallPkg (viz., [P] $\sim_o$ InstallPkg). It is easy to observe that algorithm *FindOperations* returns a minimal set of operations of UbuntuOS that can simulate InstallPkg, namely {Retrieve,Download,Install}. Such set can then be used to build a plan p simulating the input-output behaviour of the desired operation InstallPkg:

$$\mathsf{P} = \mathsf{Retrieve} \cdot \mathsf{Download} \cdot \mathsf{Install}$$

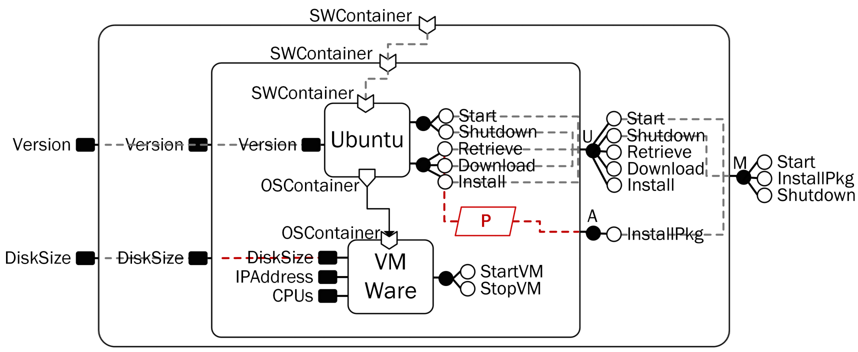


Figure 16: White-box adaptation of a *ServiceTemplate*.

Fig. 16 illustrates the adaptation of UbuntuOS. Its `BoundaryDefinitions` are first extended to expose property DiskSize of node VMWare as property DiskSpace, and to expose the plan P as operation InstallPkg. Then, the resulting `ServiceTemplate` is encapsulated into a new `ServiceTemplate` so as to expose only the capabilities, properties, and interfaces of the `NodeType` OS to be matched. □

4.3. Adaptation of matched `ServiceTemplate`*s*

The matching definitions given in the previous sections may be implemented by employing ontology-based descriptions of cloud services [31]. To avoid all the ontology-related problems (such as the cross-ontology matchmaking [22, 25]), in this section we propose a methodology to manually adapt unmatched plug-in `ServiceTemplate`s so as to exactly match the target `NodeType`s. Namely, we show how to exactly match target `NodeTypes` by non-intrusively adapting *flexibly* matched `ServiceTemplate`s and by intrusively adapting *white-box* matched `ServiceTemplate`s.

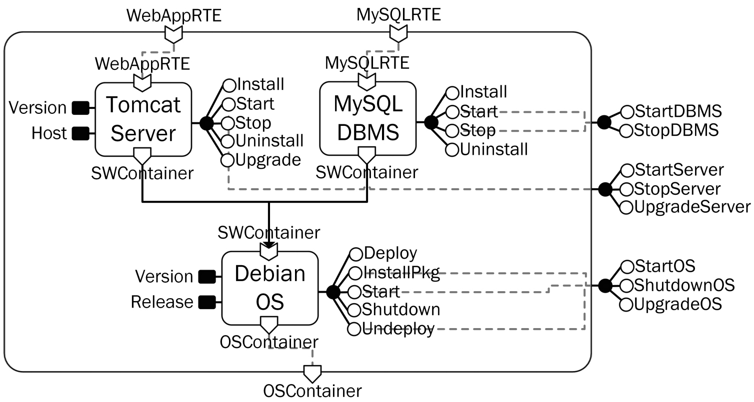


Figure 17: Available `ServiceTemplate` WebAppEnvironment.

In doing so, we also provide some examples showing how to adapt the `ServiceTemplate` WebAppEnviroment (Fig. 17) to exactly match different `NodeType`s. For the sake of simplicity, we will assume that each `Capability` or `Requirement` is of an homonym `CapabilityType` or `RequirementType` (e.g., the `Capability`WebAppRTE is of WebAppRTE `CapabilityType`, the `Requirement`SWContainer is of SWContainer `RequirementType`, etc.). We will also assume that all properties are of type `String`, and that all operations have no input parameters and return a `Boolean` parameter witnessing whether they successfully completed (e.g., the TomcatServer's operation Start, as well as the operation StartServer on the boundaries, return a parameter TomcatServerStarted, which is `true` if the TomcatServer has correctly started, and `false` otherwise). Finally, we will abstract from `Policies`, as they just require to check whether they are applicable to a `NodeType`, and do not require any adaptation.

4.3.1. Adaptation of flexibly matching `ServiceTemplate`*s*

As we illustrated in Sect. 4.1, a `ServiceTemplate` S *flexibly* matches a target `NodeType` N when the plug-in matching fails only because of non-relevant syntactic differences. Furthermore, if S *flexibly* matches N, then the former can be adapted into a new `ServiceTemplate` which exactly matches N. The adaptation consists in building a new `ServiceTemplate` which contains the available one as a `NodeTemplate` and whose boundaries are built by declaring the same features of N and by mapping each of them to the matched feature of S. This can be done automatically if we employ ontologies, otherwise we need the manual intervention of the application designer.

We now illustrate how the application designer may non-intrusively adapt a `ServiceTemplate` S which does not plug-in match a `NodeType` N into a new `ServiceTemplate` S' which exactly matches N. Fig. 18 describes how such an adaptation can be

(1) Create the adapted `ServiceTemplate` S' which initially contains S as the only `NodeTemplate` in its topology.

(2) For each capability (property) exposed by N

(a) define a capability (property) with the same name and type on the boundaries of S', and

(b) map the defined capability (property) onto the corresponding one of S.

(3) For each interface exposed by N, define an interface with the same name on the boundaries of S'. Then, for each operation o exposed by (an interface of) N

(a) define an operation with the same name and parameters in the corresponding interface exposed by S', and

(b) map such operation onto an operation of S which is semantically equivalent to o.

(4) Add a dummy `NodeTemplate` *NoBe* (whose capabilities satisfy the requirements of S and whose requirements are the same of N) to the topology of S'. Then, for each requirement exposed by N

(a) define a requirement with the same name and type on the boundaries of S', and

(b) map the defined requirement to the corresponding one of *NoBe*.

(where mapping f onto f' simply means that f is a reference to f')

Figure 18: Adaptation of *flexibly* matching `ServiceTemplates`.

successfully performed when S exposes all capabilities, properties, interface operations and requirements as N, but in a *syntactically* different way. The adaptation described in Fig. 18 implements the relaxed matching conditions of Def. 10 (in terms of ontology-based name equivalences). It is worth noting that, according to the definition of *flexible* matching, the adaptation process cannot succeed if capability, property, operation or requirement mismatches are not just syntactic (i.e., if they are not only due to names which are syntactically different but semantically equivalent). Namely, the adaptation in Fig. 18 will fail if one of the steps cannot be performed, while it succeed if all the steps are performed.

Example 7. Consider the target `NodeType` WebEnv in Fig. 19, where the capabilities WebAppRuntime and MySQLRuntime are respectively of type WebAppRTE and MySQLRTE, and where

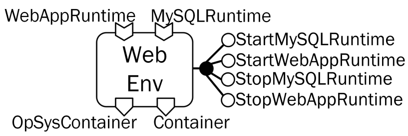


Figure 19: Target `NodeType` WebEnv.

both requirements are of type OSContainer. All operations are without input parameters, and return a `Boolean` parameter witnessing whether they successfully completed (e.g., the operation StartWebAppRuntime returns a parameter WebAppRuntimeStarted, which is `true` if the WebAppRuntime capability is concretely provided, and `false` otherwise). We observe that, according to Def. 10, the available `ServiceTemplate` WebAppEnvironment (Fig. 17) flexibly matches the target `NodeType` WebEnv.

Figs. 20, 21 and 22 illustrate how WebAppEnvironment can be adapted so as to exactly match WebEnv. First, (1) we create a new `ServiceTemplate` which contains WebAppEnvironment as the only `NodeTemplate` (Fig. 20).

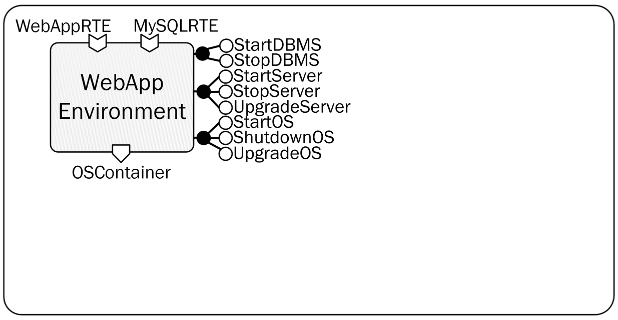


Figure 20: Example of application of step 1 (of the adaptation methodology).

Since WebAppRTE and WebAppRuntime, as well as MySQLRTE and MySQLRuntime, are of the same `CapabilityType`, (2) we adapt the capabilities by adding WebAppRuntime and MySQLRuntime to the boundaries of the adapted `ServiceTemplate` and by mapping them to the corresponding capabilities of the available `ServiceTemplate` (i.e., WebAppEnvironment. Analogously, (3) we adapt the operations StartMySQLRuntime, StartWebAppRuntime, StopMySQLRuntime, and StopWebAppRuntime, by mapping them to the corresponding operations of WebAppEnvironment (i.e., StartDBMS, StartServer, StopDBMS, and StopServer — Fig. 21).

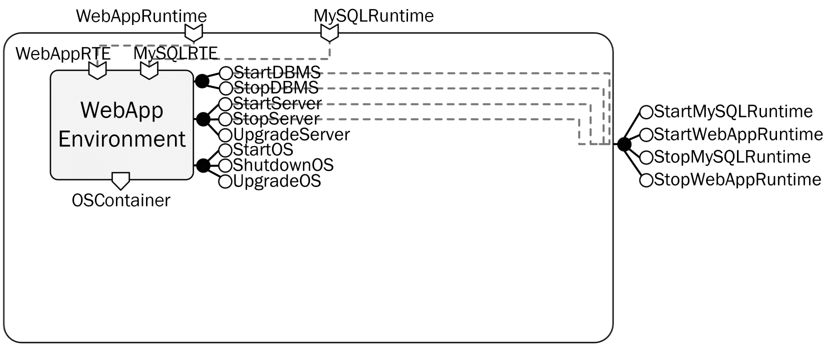


Figure 21: Example of application of steps 2 and 3 (of the adaptation methodology).

Finally, (4) we artificially extend the requirements of the available `ServiceTemplate` (i.e., WebAppEnvironment) to exactly match those of target `NodeType` (i.e., WebEnv). Namely, we add a dummy `NodeTemplate` NoBe (whose capabilities satisfy the requirements of WebAppEnvironment and whose requirements are the same of WebEnv) to the topology of the adapted

`ServiceTemplate`, we define the same requirements of the target `NodeType` on the boundaries of the adapted `ServiceTemplate`, and we map each of them to the corresponding one of NoBe (Fig. 22).

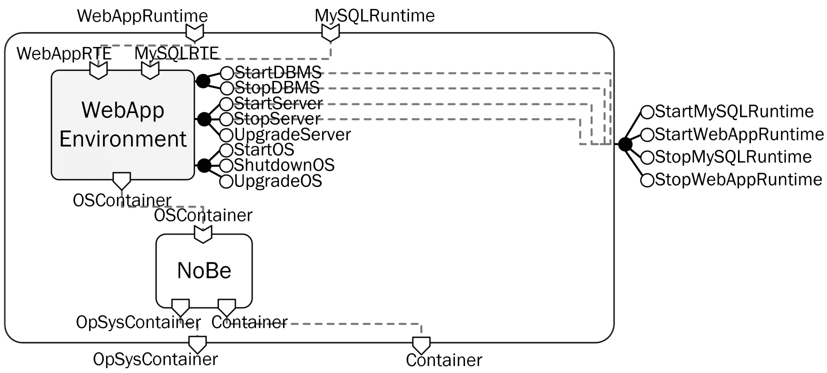


Figure 22: Example of application of step 4 (of the adaptation methodology).

The obtained `ServiceTemplate` (Fig. 22) exposes all the features exhibited by the target `NodeType` WebEnv. This implies that they exactly match, and subsequently that the adapted `ServiceTemplate` can be employed to instantiate WebEnv. □

4.3.2. Adaptation of white-box matching `ServiceTemplate`*s*

White-box matching relaxes *flexible* matching by extending the feature research to the internal elements of a `ServiceTemplate`'s topology, and by allowing to combine internal operations in order to obtain plans which are semantically equivalent to missing operations (Sect. 4.2). Furthermore, if S *white-box* matches N, then the former can be adapted into a new `ServiceTemplate` which exactly matches N. The adaptation consists of building a new `ServiceTemplate` which contains the available one as a `NodeTemplate` and whose boundaries are built by declaring the same features of N and by mapping each one of them to the matched feature of S. This can be done automatically if we employ ontologies, otherwise the application designer needs to manually adapt S.

Fig. 23 illustrates how an application designer may intrusively adapt a `ServiceTemplate` S which does not flexibly match a `NodeType` N into a new `ServiceTemplate` S' which exactly matches N. Namely, it describes how such an adaptation can be successfully performed when (i) S exposes all capabilities, properties, and requirements as N, but internally and/or in a *syntactically* different way, and (ii) when N features one or more interface operation which is not matched by any operation featured by S, while it can be matched by some composition of S's internal operations. The adaptation described in Fig. 23 implements the relaxed matching conditions that were defined in Def. 14 (in terms of ontology-based name equivalences).

It is worth noting that, according to the definition of *white-box* matching, the adaptation process cannot succeed if missing capabilities and properties cannot be matched internally as well as if operation mismatches cannot be solved by composing internal operations. Namely, the adaptation in Fig. 23 will fail if one of the steps cannot be performed, while it succeed if all the steps are performed.

(1) Create the adapted `ServiceTemplate` S' which initially contains S as the only `NodeTemplate` in its topology.

(2) For each capability (property) c exposed by N

(a) define a capability (property) with the same name and type on the boundaries of S',

(b) if c does not correspond to any capability (property) exposed by S, search inside of the topology of S a capability (property) corresponding to c and expose it on the boundaries of S, and

(c) map such capability (property) to the corresponding one exposed by S.

(3) For each interface exposed by N, define an interface with the same name on the boundaries of S'. Then, for each operation o exposed by N,

(a) define an operation with the same name and parameters in the corresponding interface exposed by S', and

(b) map the defined operation to

- an operation of S which is semantically equivalent to o, or
- a (new) operation o_{ex} of S which is suitably extracted from its internal definitions. With "suitably extracted" we mean that (i) a new interface has been defined on the boundaries of S, and (ii) o_{ex} has been added to its operations, and (iii) o_{ex} has been mapped to either an internal operation of S which is considered semantically equivalent to o or a plan which combines the internal operations of S to obtain an operation which is semantically equivalent to o.

(4) Add a dummy `NodeTemplate` *NoBe* (whose capabilities satisfy the requirements of S and whose requirements are the same of N) to the topology of S'. Then, for each requirement exposed by N

(a) define a requirement with the same name and type on the boundaries of S', and

(b) map the defined requirement to the corresponding one of *NoBe*.

(where mapping f onto f' simply means that f is a reference to f').

Figure 23: Adaptation of *white-box* matching `ServiceTemplate`s.

Example 8. Consider the target `NodeType` IntegratedWebEnv in Fig. 24, where the capabilities WebAppRuntime and MySQLRuntime are respectively of type WebAppRTE and MySQLRTE, and where both requirements are of type OSContainer. Both operations are without input parameters, and return two `Boolean` parameters witnessing whether they successfully completed (e.g., the operation Start returns two parameters WebAppRuntimeStarted and MySQLRuntime-

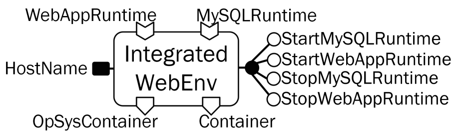


Figure 24: Target `NodeType` IntegratedWebEnv.

Started, each of which is `true` if the corresponding capability is concretely provided, and `false` otherwise). We observe that, according to Def. 14, the available `ServiceTemplate` WebAppEnvironment (Fig. 17) white-box matches the target `NodeType` IntegratedWebEnv.

Fig. 25 illustrates the `ServiceTemplate` obtained by applying the adaptation process of Fig. 23 to the `ServiceTemplate` WebAppEnvironment (to exactly match the target `NodeType` IntegratedWebEnvironment). Namely, we first created a new `ServiceTemplate` which contains WebAppEnvironment as the only `NodeTemplate` in its topology. Capabilities and requirements have been adapted as shown in Example 7. The HostName property has been extracted from WebAppEnvironment's internal topology and then mapped on the boundaries of the adapted `ServiceTemplate`. The operation Start required to generate a `Plan` $\mathrm{P}_{\mathrm{Start}}$ combining the Start operations of TomcatServer and MySQLDBMS, to expose such `Plan` as an operation of WebAppEnvironment, and then to expose such operation also on the boundaries of the adapted `ServiceTemplate`. The adaptation required to obtain operation Stop was analogous.

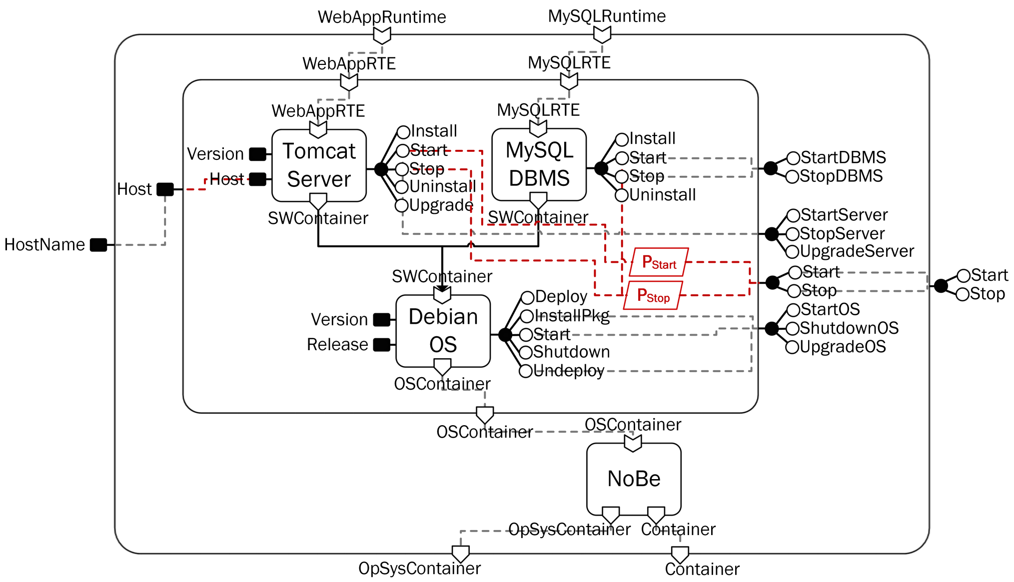


Figure 25: Example of intrusive adaptation of a *white-box* matching `ServiceTemplate`.

The obtained `ServiceTemplate` exposes all the features exhibited by the target `NodeType` IntegrateWebEnvironment. This implies that they exactly match and subsequently that the adapted `ServiceTemplate` can be employed to instantiate IntegratedWebEnvironment. □

4.3.3. Further remarks

It is important to observe that the adaptation of a TOSCA `ServiceTemplate` S to (exactly) match a `NodeType` N does suffice to reuse any actual service modelled by S to deploy cloud applications that rely on N. This is thanks to the powerful way in which TOSCA supports the deployment of cloud applications. TOSCA permits to pack in a CSAR (*Cloud Service ARchive*) file an application specification together with the actual executable files to be deployed on a cloud platform. When a CSAR file is given in input to a TOSCA container, the latter takes care of deploying and executing the application specification contained in the CSAR file [13, 29]. Therefore, in order to adapt an actual service modelled by a `ServiceTemplate` S to deploy an application that relies on a `No-`

deType N, it suffices to adapt S into a new ServiceTemplate S' that matches N — without having to generate an implementation of the adaptation specified by S.

Note that the adaptation works also in the case in which the CSAR of S should not be available, for instance when S is a proprietary service offered by a third party. In such cases it suffices to develop a simple proxy of the remote service modelled by S, and to pack it in a new CSAR file together with the application specification containing S' (and the executable files associated with such specification).

Finally, it is also worth highlighting that, thanks to the features of TOSCA, the simple adaptation methodology described in this paper considerably reduces the work needed to reuse cloud services if compared with the alternative of explicitly devising adapters as in traditional software adaptation approaches (e.g., [6, 20, 23]).

5. Related work

Our work started from the observation that while the matching between ServiceTemplates and NodeTypes is indicated in [29] as a way to instantiate abstract TOSCA NodeTypes, no formal definition of *matching* is given in either [28] or [29]. A concrete definition of matching for TOSCA was used in [35] to define a way to merge TOSCA services by matching entire portions of their TopologyTemplates. The definition of matching of single service components employed in [35] is however very strict, as two service components are considered to match only if they expose the same qualified name. Our work aims to contribute to the TOSCA specification by proposing four definitions of matching between ServiceTemplates and NodeTypes, each identifying larger sets of ServiceTemplates that can be adapted so as to (exactly) match a NodeType.

It is worth mentioning that, in our previous work [11], we assumed that the semantics of requirements was determined by RequirementTypes. Nevertheless, according to [28], a NodeType might define multiple requirements of the same RequirementType, in which case each occurrence of a requirement definition is uniquely identified by its name (e.g., two requirements of DBRequirementType where one could be named CustomerDatabase and the other one could be named ProductsDatabase). Since the same holds for capabilities, in this paper we refined the *plug-in*/*flexible*/*white-box* matching of requirements and capabilities accordingly.

This paper extends [11] also by providing a proof-of-concept implementation of both the *exact* and *plug-in* matching, by providing a pseudo-code for automatically adapt *plug-in* matched ServiceTemplate, and by defining a methodology for adapting ServiceTemplates which *flexibly*/*white-box* match target NodeTypes (which replaces the usage of ontologies — and subsequently removes all ontology-related problems — with the application developer decisions). Please note that the proposed methodology is an adaptation of that we proposed in [12] for non plug-in matching ServiceTemplates.

In the following we will position our work with respect to other solutions for the matching of available services (Sect. 5.1) and their adaptation (Sect. 5.2).

5.1. Service matching

The problem of how to match (Web) services has been extensively studied in recent years. Many approaches are ontology-aware [31], like for instance the ontology-aware matchmaker for OWL-S services described in [22]. Other approaches are behaviour-aware, like the (ontology-aware) trace-based matching of YAWL services defined in [10], the (ontology-aware) behavioural congruence for OWL-S services defined in [5], or the graph transformation based matching defined in [14] and the heuristic black-box matching described in [18] 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 [22] and [18] 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 [24] or [26]. Generally speaking, the notion of matching defined in the present paper differs from most QoS-aware matching approaches since it compares *types* rather than actual *values* of extra-functional features like QoS. For instance, a type-based definition of matching is defined in [17] to type check "stream flows" for interactive distributed multimedia applications. While the context of [17] is different from ours, two of the matching conditions considered in [17] 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 that takes into account both functional and extra-functional features, by relying both on types and on ontologies to overcome non-relevant syntactic information.

It is also worth highlighting that our notions of *plug-in*, *flexible* and *white-box* matching share the basic objectives with alternating refinement relations [1] (and more in general with the notion of simulation [33]). Indeed, they all check whether an available component is capable of offering all the features/options of a desired component (without imposing additional requirements). However, while [1, 33] rely on both the signature and behaviour of components, our notions of matching only rely on components' signature, as this is what can be described in TOSCA.

It is thus interesting to extend TOSCA by permitting to describe the behaviour of a component, and to extend our notions of matching to take into account such behaviour information. For instance, one may model a component's behaviour through labelled transition systems (e.g., management protocols [7, 8]) or service contracts [27]. This would permit extending our matching notions by checking whether a transition system "simulates" [33] another, whether they directly match one another (as for interface automata theory [15]), or whether the contract of an available `ServiceTemplate` is coherent with that of a desired `NodeType`.

5.2. Service adaptation

The development of systematic approaches to adapt (and reuse) existing software is widely recognized as one of the crucial problems in system integration [36]. In spite of the increasing availability of cloud services, currently platform-specific code often needs to be manually modified to (re)use services in cloud applications. This is obviously an expensive and error-prone activity, as pointed out in [34], both for the learning curve and for the testing phases needed.

Various efforts have been recently oriented to try devising systematic approaches to reuse cloud services. For instance, [16] and [21] propose two approaches to transform platform-agnostic source code of applications developed with a model-driven methodology into platform-specific applications. In contrast, our approach does not restrict to applications developed with a specific methodology, nor it requires the availability of applications' source code, and it is hence applicable also to third-party services whose source code is not available nor open.

[20] proposes a framework which allows developers to write the source code of cloud services as if they were "in-house" applications. Cloud deployment information must be provided in a separate file, and a middleware layer employs source and deployment information to generate the artifacts to be deployed on cloud platforms. We believe that our approach improves [20] in three ways. First, only some cloud platforms are targeted in [20], while our approach can be applied on any (TOSCA-compliant) platform. Moreover, in [20] the reuse of a cloud service requires invoking the middleware layer, while in our approach adaptation is performed only once. Finally, [20] always requires to write source code, while our approach only requires to edit the application specification.

In general, most existing approaches to the reuse of cloud services support a from-scratch development of cloud-agnostic applications, and do not account for the possibility of adapting existing (third-party) cloud services. To the best of our knowledge, ours is the first approach which proposes a methodological approach for adapting existing cloud applications, by relying on TOSCA [28] as the standard for cloud interoperability, and to support an easy reuse of third-party services.

Finally, it is worth noting that the novelty of our approach does not reside in the type of adaptation techniques that we employ to adapt `ServiceTemplate`s. Indeed, our methodology exploits well-know adaptation patterns (e.g., [2, 19]) to adapt TOSCA templates. The novelty of our approach is rather that, in contrast with traditional adaptation approaches (e.g., [6, 20, 23]), no additional code must be developed to reuse existing applications. This is because we exploit the possibilities *natively* provided by TOSCA of mapping exposed features onto internal ones, and of entirely delegating the management of such mappings to TOSCA engines.

More precisely, TOSCA *natively* only supports the one-to-one mapping among requirements, capabilities, and properties, and the one-to-many mapping among operations, and this is why our approach only combines operations. Going beyond one-to-one mapping among requirements, capabilities, and properties could be interesting, even if this would require to go also beyond what is natively supported by TOSCA (thus requiring developers to write the additional source code implementing the adaptation).

6. Conclusions

The results presented in this paper intend to contribute to the formal definition of TOSCA. After defining the notion of *exact matching* between TOSCA `ServiceTemplate`s and `NodeType`s, 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`s with respect to `NodeType`s. To allow exploiting the new notions of matching not only for type-checking but also for node instantiation, 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.

To demonstrate the feasibility of our approach, we have also presented a proof-of-concept implementation of the *exact* and *plug-in* matchings. Such implementation should be properly extended so as to be fruitfully integrated in a plug-in for the OpenTOSCA [4] open source environment, in order to enhance its type-checking capabilities. Furthermore, our implementation performs the matchmaking in a "verbose" way (i.e., by suitably storing information also if there is no matching). We showed how to employ the "verbose" matching results to exploit the adaptation of plug-in matched services. Since the proposed approach is based on purely syntactic choices, it can be completely automated and implemented in the above mentioned plug-in for the TOSCA implementations. As no high-level API is available yet to manage TOSCA elements [3], the implementation of such a plug-in is left for future work.

In this paper we also presented a way to manually perform the *flexible*/*white-box* matching and adaptation. As we already mentioned, this allows cloud application developers to work without equipping their services with ontologies and to avoid all ontology related problems (e.g., cross-ontology matchmaking [22], [25]). It is worth noting that the composition of operations may be employed not only for *white-box* matching but also for the *flexible* one.

The definitions of matching presented in this paper can be extended to also take into account the behaviour of cloud services. As we discussed in Sect. 5.1, an interesting direction is to employ management protocols [7, 8] to model the behaviour of `NodeType`s and `ServiceTemplate`s, and to devise (new) techniques to check whether the protocol of an available `ServiceTemplate` can "simulate" [33] that of a desired `NodeType`. This would permit extending the matching notions we proposed in this paper to take into account the behaviour of services by simply including such notions of simulation. This extension is in the scope of our immediate future work.

Besides *types*, one would also like to check actual *values* of policies and properties. This would permit verifying also the compliance of a `ServiceTemplate` with `NodeTemplate`s that instantiate a matching `NodeType` (e.g., by considering a partial ordering over the domain of a policy/property, we could be able to determine whether a desired value is compatible with an available one). This extension is also in the scope of our immediate future work.

References

[1] Rajeev Alur, Thomas A. Henzinger, Orna Kupferman, and Moshe Y. Vardi. Alternating refinement relations. In *Proceedings of the 9th International Conference on Concurrency Theory*, CONCUR '98, pages 163–178. Springer-Verlag, 1998.

[2] Steffen Becker, Antonio Brogi, Ian Gorton, Sven Overhage, Alexander Romanovsky, and Massimo Tivoli. Towards an engineering approach to component adaptation. In *Proceedings of the 2004 International Conference on Architecting Systems with Trustworthy Components*, pages 193–215. Springer-Verlag, 2006.

[3] Tobias Binz. Personal communication, 2013, November 22nd.

[4] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA – a runtime for TOSCA-based cloud applications. In *11th International Conference on Service-Oriented Computing*. Springer, 2013.

[5] Filippo Bonchi, Antonio Brogi, Sara Corfini, and Fabio Gadducci. A net-based approach to web services publication and replaceability. *Fundam. Inf.*, 94(3-4):305–330, 2009.

[6] Andrea Bracciali, Antonio Brogi, and Carlos Canal. A formal approach to component adaptation. *J. Syst. Softw.*, 74(1):45–54, 2005.

[7] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling and analysing cloud application management. In *Proceedings of the 4th European Conference on Service-Oriented and Cloud Computing (ESOCC 2015)*, LNCS. Springer, 2015. *In press.*

[8] Antonio Brogi, Andrea Canciani, Jacopo Soldani, and PengWei Wang. Modelling the behaviour of management operations in cloud-based applications. In Daniel Moldt, editor, *Proceedings of the International Workshop on Petri Nets and Software Engineering, PNSE'15*, volume 1372 of *CEUR Workshop Proceedings*, pages 191–205. CEUR-WS.org, 2015.

[9] Antonio Brogi and Sara Corfini. Behaviour-aware discovery of web service compositions. *Int. J. Web Service Res.*, 4(3):1–25, 2007.

[10] Antonio Brogi and Razvan Popescu. Service adaptation through trace inspection. *International Journal of Business Process Integration and Management*, 2(1):9–16, 2007.

[11] Antonio Brogi and Jacopo Soldani. Matching cloud services with TOSCA. In Carlos Canal and Massimo Villari, editors, *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communications in Computer and Information Science*, pages 218–232. Springer, 2013.

[12] Antonio Brogi and Jacopo Soldani. Reusing cloud-based services with TOSCA. In *INFORMATIK 2014, Lecture Notes in Informatics (LNI)*, volume 232, pages 235–246. Gesellschaft für Informatik (GI), 2014.

[13] Antonio Brogi, Jacopo Soldani, and PengWei Wang. TOSCA in a Nutshell: Promises and Perspectives. In Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau, editors, *Service-Oriented and Cloud Computing*, volume 8745 of *LNCS*, pages 171–186. Springer, 2014.

[14] Juan Carlos Corrales, Daniela Grigori, and Mokrane Bouzeghoub. BPEL processes matchmaking for service discovery. In *Proceedings of the 2006 Confederated International Conference on On the Move to Meaningful Internet Systems: CoopIS, DOA, GADA, and ODBASE - Volume Part I*, ODBASE'06/OTM'06, pages 237–254. Springer-Verlag, 2006.

[15] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120. ACM, 2001.

[16] Beniamino Di Martino, Dana Petcu, Roberto Cossu, Pedro Goncalves, Tamás Máhr, and Miguel Loichate. Building a mOSAIC of clouds. In *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, pages 571–578. Springer-Verlag, 2011.

[17] Frank Eliassen and S. Mehus. Type checking stream flow endpoints. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, Middleware '98, pages 305–320. Springer-Verlag, 1998.

[18] Rik Eshuis and Paul Grefen. Structural matching of BPEL processes. In *Proceedings of the Fifth European Conference on Web Services*, ECOWS '07, pages 171–180. IEEE Computer Society, 2007.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[20] Joaquín Guillén, Javier Miranda, Juan Manuel Murillo, and Carlos Canal. A service-oriented framework for developing cross cloud migratable software. *J. Syst. Softw.*, 86(9):2294–2308, 2013.

[21] Mohammad Hamdaqa, Tassos Livogiannis, and Ladan Tahvildari. A reference model for developing cloud applications. In Frank Leymann, Ivan Ivanov, Marten van Sinderen, and Boris Shishkov, editors, *CLOSER 2011 - Proceedings of the 1st International Conference on Cloud Computing and Services Science.*

[22] Matthias Klusch, Benedikt Fries, and Katia Sycara. OWLS-MX: A hybrid semantic web service matchmaker for OWL-S services. *Web Semant.*, 7(2):121–133, 2009.

[23] Woralak Kongdenfha, Hamid Reza Motahari-Nezhad, Boualem Benatallah, Fabio Casati, and Regis Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Trans. Serv. Comput.*, 2(2):94–107, 2009.

[24] Farzad Mahdikhani, Mahmoud Reza Hashemi, and Marjan Sirjani. QoS aspects in web services compositions. In *Service-Oriented System Engineering, 2008. SOSE'08. IEEE International Symposium on*, pages 239–244. IEEE, 2008.

[25] Jorge Martinez-Gil, Ismael Navas-Delgado, and Jose F Aldana-Montes. Maf: An ontology matching framework. *Journal of Universal Computer Science*, 18(2):194–217, 2012.

[26] Sonia Ben Mokhtar, Davy Preuveneers, Nikolaos Georgantas, Valérie Issarny, and Yolande Berbers. EASY: Efficient semantic service discovery in pervasive computing environments with qos and context support. *J. Syst. Softw.*, 81(5):785–808, 2008.

[27] Eric Newcomer and Greg Lomow. *Understanding SOA with Web services.* Addison-Wesley, 2005.

[28] OASIS. Topology and Orchestration Specification for Cloud Applications. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`, 2013.

[29] OASIS. 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`, 2013.

[30] OASIS. TOSCA XML Schema Definition. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/`

schemas/TOSCA-v1.0.xsd, 2013.

[31] Declan O'Sullivan and David Lewis. Semantically driven service interoperability for pervasive computing. In *Proceedings of the 3rd ACM International Workshop on Data Engineering for Wireless and Mobile Access*, MobiDe '03, pages 17–24. ACM, 2003.

[32] Dana Petcu, Georgiana Macariu, Silviu Panica, and Ciprian Crciun. Portable cloud applications-from theory to practice. *Future Gener. Comput. Syst.*, 29(6):1417–1430, 2013.

[33] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction.* Cambridge University Press, 2011.

[34] Van Tran, Jacky Keung, Anna Liu, and Alan Fekete. Application migration to cloud: A taxonomy of critical factors. In *Proceedings of the 2Nd International Workshop on Software Engineering for Cloud Computing*, SECLOUD '11, pages 22–28. ACM, 2011.

[35] Andreas Weiss. Merging of TOSCA cloud topology templates. Master's thesis, Institute of Architecture of Application Systems, University of Stuttgart, 2012. http://elib.uni-stuttgart.de/opus/volltexte/2012/7932/pdf/MSTR_3341.pdf.

[36] Xie Xiong and Zhang Weishi. The current state of software component adaptation. In *First International Conference on Semantics, Knowledge and Grid, 2005. SKG '05.*, pages 103–103, Nov 2005.