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.](image_url)

The TopologyTemplate is a typed directed graph describing the structure of a composite cloud application. Its nodes (NodeTemplates) model the application components, while its edges (RelationshipTemplates) model the relations among those components. NodeTemplates and RelationshipTemplates are typed by means of NodeType and RelationshipType, 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. RelationshipTypes 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 RequirementTypes), capabilities by CapabilityDefinitions (of certain CapabilityTypes), properties by PropertiesDefinition, and operations by Interfaces and Operations. Requirements, capabilities, properties and operations externally exposed by a ServiceTemplate can be described in its BoundaryDefinitions.

NodeTemplates and RelationshipTemplates can also declare QoS information by exposing Policy elements, which in turn must be typed by referring PolicyTypes. A PolicyType defines the structure of the QoS information (as well as the NodeTypes 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 Operations 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 (≡) a NodeType. Then, we formally define the plug-in matching (≃), which relaxes the exact one (viz., ≡⊂≃) in order to identify larger sets of ServiceTemplates 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.

Definition 1. A ServiceTemplate S exactly matches a NodeType N (S ≡ N) iff:

1. Reqs(S) ≡R Reqs(N) and
2. Caps(S) ≡C Caps(N) and
3. Pols(S) ≡PO Pols(N) and
4. Props(S) ≡PR Props(N) and
5. Ints(S) ≡I Ints(N).

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:
- name(x) denotes the name of x, where x can be a requirement, capability, property, interface, operation, or parameter of N or S.
- type(x) denotes the type of x, where x can be a requirement, capability, policy, property, or parameter of N or S.
- XMLtype(x) denotes the XML type of x, where x can be Props(N) or 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:
Reqs(S) ≡R Reqs(N) iff
∀rS ∈ Reqs(S) ∃!rN ∈ Reqs(N) : name(rS) = name(rN) ∧ type(rS) = type(rN), and

---

1Strictly 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.
\[ \forall r_N \in \text{Reqs}(N) \exists r_S \in \text{Reqs}(S) : \text{name}(r_N) = \text{name}(r_S) \land \text{type}(r_N) = \text{type}(r_S). \]

\[ \text{Caps}(S) \equiv C \text{Caps}(N) \text{ iff } \forall c_S \in \text{Caps}(S) \exists c_N \in \text{Caps}(N) : \text{name}(c_S) = \text{name}(c_N) \land \text{type}(c_S) = \text{type}(c_N), \text{ and } \forall c_N \in \text{Caps}(N) \exists c_S \in \text{Caps}(S) : \text{name}(c_S) = \text{name}(c_N) \land \text{type}(c_S) = \text{type}(c_N). \]

According to [28], a \text{PolicyType} can be associated with a set of \text{NodeTypes} to which it is applicable\(^2\). To ensure exact matching, the type of each \text{Policy} of \text{S} must therefore be one of the \text{PolicyTypes} applicable to \text{N}.

**Definition 3.** Let \( N \) be a \text{NodeType} and let \( S \) be a \text{ServiceTemplate}. Then:
\[ \text{Pols}(S) \equiv_{PO} \text{Pols}(N) \text{ iff } \forall \text{pol}_S \in \text{Pols}(S) : \text{type}(\text{pol}_S) \in \text{Pols}(N). \]

Furthermore, since a \text{NodeType} only specifies the XML schema of its observable properties (while \text{ServiceTemplates} specify actual values of properties), property matching reduces to comparing XML types.

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

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 \text{NodeType} and let \( S \) be a \text{ServiceTemplate}. Then:
\[ \text{Ints}(S) \equiv I \text{Ints}(N) \text{ iff } \forall i_S \in \text{Ints}(S) \exists i_N \in \text{Ints}(N) : \text{name}(i_S) = \text{name}(i_N) \land \forall o_S \in \text{Ops}(i_S) \exists o_N \in \text{Ops}(i_N) : o_S \equiv_o o_N \text{ and } \forall o_N \in \text{Ops}(i_N) \exists o_S \in \text{Ops}(i_S) : o_N \equiv_o o_S \]

where \( \text{Ops}(_) \) denotes the set of operations of an interface and where
\[ o_x \equiv_o o_y \text{ iff } \text{name}(o_x) = \text{name}(o_y) \land \forall a \in I(o_x), b \in I(o_y) : \text{name}(a) = \text{name}(b) \land \text{type}(a) = \text{type}(b), \text{ and } \forall a \in I(o_y), b \in I(o_x) : \text{name}(a) = \text{name}(b) \land \text{type}(a) = \text{type}(b), \text{ and } \forall a \in O(o_x), b \in O(o_y) : \text{name}(a) = \text{name}(b) \land \text{type}(a) = \text{type}(b), \text{ and } \forall b \in O(o_y), a \in O(o_x) : \text{name}(a) = \text{name}(b) \land \text{type}(a) = \text{type}(b) \]

where \( I(_) \) and \( O(_) \) 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 \text{NodeTypes} \( N_1 \) and \( N_2 \) and the \text{ServiceTemplate} \( S \) of Fig. 3, where \( C \) and \( C_{\text{cap}} \) denote sets of capabilities, \( R \) and \( R_{\text{req}} \) denote sets of requirements, \( p_i \) denotes a property, \( i_j \) denotes an interface, \( o_j \) denotes an operation, and where policies and operation parameters are omitted for readability. Suppose that \( S \) exactly matches \( N_1 \) (viz., \( S \equiv N_1 \)) and that \( N_2 \) differs from \( N_1 \) since it exposes “more” requirements than \( N_1 \) and “less” capabilities, properties and operations than \( N_1 \). While, according to Defs. 1—5, \( S \) cannot exactly match \( N_2 \) (viz., \( S \not\equiv N_2 \)), a less strict definition of matching should allow \( S \) to match also \( N_2 \) (as we will discuss in the next section). \( \square \)

\(^2\)We assume that a \text{PolicyType} is applicable to all \text{NodeTypes} if not specified otherwise.
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 NodeTypes 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. $\text{Reqs}(S) \simeq_R \text{Reqs}(N)$ and
2. $\text{Caps}(S) \simeq_C \text{Caps}(N)$ and
3. $\text{Pols}(S) \equiv_{PO} \text{Pols}(N)$ and
4. $\text{Props}(S) \simeq_{PR} \text{Props}(N)$ and
5. $\text{Ints}(S) \simeq_I \text{Ints}(N)$.

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 or is equal to $t$.

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

$\text{Reqs}(S) \simeq_R \text{Reqs}(N)$ iff

$$\forall r_S \in \text{Reqs}(S) \exists r_N \in \text{Reqs}(N) : \text{name}(r_N) = \text{name}(r_S) \land \text{type}(r_N) \geq \text{type}(r_S).$$

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:

$\text{Caps}(S) \simeq_C \text{Caps}(N)$ iff

$$\forall c_N \in \text{Caps}(N) \exists c_S \in \text{Caps}(S) : \text{name}(c_S) = \text{name}(c_N) \land \text{type}(c_S) \geq \text{type}(c_N).$$

$\text{Props}(S) \simeq_{PR} \text{Props}(N)$ iff $\text{XMLtype}(\text{Props}(S)) \geq \text{XMLtype}(\text{Props}(N))$.

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

---

3More precisely, if $t$ and $t'$ are TOSCA elements then $t'$ extends $t$ if $t'$ is (directly or indirectly) 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:

\[
\text{Ints}(S) \simeq_1 \text{Ints}(N) \text{ iff } \forall i_N, o_N : i_N \in \text{Ints}(N) \land o_N \in \text{Ops}(i_N) \\
\exists i_S, o_S : i_S \in \text{Ints}(S) \land o_S \in \text{Ops}(i_S) : o_S \equiv_o o_N.
\]

It is worth noting that when a ServiceTemplate \( S \) plug-in matches a NodeType \( N \), \( S \) can be easily adapted into a new ServiceTemplate \( S' \) that exactly matches the 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 \( N \) since the latter exposes “more” requirements and “less” capabilities, properties and operations than the former. Since \( S \) exposes one property (\( p_2 \)) and one operation (\( o_4 \)) more than \( N \), we have that \( \text{Props}(S) \simeq_{PR} \text{Props}(N) \) and \( \text{Ints}(S) \simeq_{PR} \text{Ints}(N) \) by Defs. 8 and 9, respectively. Therefore, if \( R \simeq_R R_{\text{sub}} \) and \( C \simeq_C C_{\text{sup}} \) hold too, then \( S \) plug-in matches \( N \) (\( S \simeq N \)). Fig. 4.(a) illustrates how \( S \) can be adapted to exactly match \( N \).

![Figure 4: Plug-in matching examples.](image)

Consider now the NodeType \( N_3 \) in Fig. 4.(b), which differs from \( N_2 \) only since it exposes property \( p_A \) instead of property \( p_1 \). According to Def. 8, \( S \) cannot plug-in match \( N_3 \) (\( S \not\simeq N_3 \)). However, if \( p_1 \) and \( p_A \) were (syntactically) different names for the same property and if the type of \( p_1 \) were compatible with the type of \( p_A \) (i.e., \( \text{type}(p_1) \geq \text{type}(p_A) \)), then a less strict definition of matching should allow \( S \) to match also \( N_3 \) (as we will discuss in Sect. 4).
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 CapabilityTypes 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\(^5\).

\(^4\)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/.

![Figure 5: High-level management of TOSCA capabilities.](image-url)
3.3.2. Implementation of the matchmakers

Since plug-in matching generalizes exact matching (viz., $\equiv \subseteq \simeq$), we implemented the two matchings as class hierarchies. 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 $s$ matches $n$ 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 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

```java
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.

\(^6\)The (exact) matchmaking of the other TOSCA elements is analogous.
protected void matchCapabilities() {
  exactlyMatchedCapabilities = new ArrayList<Capability>();
  areCapabilitiesMatched = false;
  unmatchedCapabilities = n.getCapabilityDefinitions().getList();
  List<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).

public boolean match() {
  super.matchCapabilities();
  super.matchRequirements();
  super.matchPolicies();
  super.matchProperties();
  super.matchInterfaces();
  if(areCapabilitiesMatched && areRequirementsMatched && arePoliciesMatched &&
    arePropertiesMatched && areInterfacesMatched) return true;
  if(!areCapabilitiesMatched) matchCapabilities();
  if(!areRequirementsMatched) matchRequirements();
  if(!arePropertiesMatched) matchProperties();
  if(!areInterfacesMatched) matchInterfaces();
  return (areCapabilitiesMatched && areRequirementsMatched && arePoliciesMatched &&
          arePropertiesMatched && areInterfacesMatched);
}

Figure 8: PlugInMatchmaker.match() method.

Consider, for instance, the matchmaking of capabilities (Fig. 9). Since it is performed after the exact matching, the setup of the environment is lighter than that of

7The (plug-in) matchmaking of the other TOSCA elements is analogous.
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 \texttt{CapabilityDefinition} in \( n \) (that has not yet been matched — line 6), it checks whether there exists a \texttt{Capability} on the boundaries of \( s \) such that they plug-in match (lines 7-14). The comparison is performed by the \texttt{match} method (lines 21-29) which checks whether a \texttt{Capability} \( c \) has the same name as \texttt{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 \texttt{unmatchedCapabilities} is properly updated (line 17). If there are no unmatched capability definitions, then the \texttt{areCapabilitiesMatched} variable is set to \texttt{true} (line 18). Otherwise, the method terminates (by leaving it set to \texttt{false}).

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

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

\[
\text{ApacheServer} \equiv \text{Server} \land \text{PaaSServer} \neq \text{Server} \land \text{PaaSServer} \simeq \text{Server} \land \text{PaaSServer2} \neq \text{Server}.
\]

---

Fig. 9: \texttt{PlugInMatchmaker.matchCapabilities()} method.
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 10: A NodeType (Server) and three ServiceTemplates (ApacheServer, PaaSServer, PaaSServer2).](image)

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.](image)

### 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 that it plug-in matches the NodeType.

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;
}
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 ServiceTemplates with respect to NodeTypes. Finally, we show how to avoid the usage of ontologies by providing a methodology for adapting unmatched plug-in ServiceTemplates 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:

- (1) $\text{Reqs}(S) \sim_R \text{Reqs}(N)$
- (2) $\text{Caps}(S) \sim_C \text{Caps}(N)$
- (3) $\text{Pols}(S) \equiv_{PO} \text{Pols}(N)$
- (4) $\text{Props}(S) \sim_{PR} \text{Props}(N)$
- (5) $\text{Ints}(S) \sim_I \text{Ints}(N)$

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.\(^\text{10}\)

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

\[
\forall r_S \in \text{Reqs}(S) \exists r_N \in \text{Reqs}(N) : \text{name}(r_N) \bowtie \text{name}(r_S) \land \text{type}(r_N) \geq \text{type}(r_S).
\]

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

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

\[
\forall r_S \in \text{Reqs}(S) \exists r_N \in \text{Reqs}(N) : \text{name}(r_N) \bowtie \text{name}(r_S) \land \text{type}(r_N) \geq \text{type}(r_S).
\]

\(^{10}\)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 $N$.
\[\text{Caps}(S) \sim_C \text{Caps}(N) \text{ iff } \forall N \in \text{Caps}(N) \exists S \in \text{Caps}(S) : \text{name}(c_S) \times \text{name}(c_N) \land \text{type}(c_S) \geq \text{type}(c_N).\]

\[\text{Props}(S) \sim_R \text{Props}(N) \text{ iff } \forall p_N \in \text{Props}(N) \exists p_S \in \text{Props}(S) : \text{name}(p_S) \times \text{name}(p_N) \land \text{type}(p_S) \geq \text{type}(p_N).\]

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:
\[\text{Ints}(S) \sim_I \text{Ints}(N) \text{ iff } \forall i_N, o_N : i_N \in \text{Ints}(N) \land o_N \in \text{Ops}(i_N) \exists i_S, o_S : i_S \in \text{Ints}(S) \land o_S \in \text{Ops}(i_S) : o_S \sim_o o_N.\]

where \(o_x \sim_o o_y \text{ iff } \begin{cases} |l(a_x)| = |l(a_y)| \text{ and } \hspace{1cm} |o(a_x)| = |o(a_y)| \text{ and} \\ \forall a \in l(a_x), \exists b \in l(a_y) : \text{name}(a) \times \text{name}(b) \land \text{type}(b) \geq \text{type}(a) \text{ and} \\ \forall b \in o(a_x), \exists a \in o(a_x) : \text{name}(a) \times \text{name}(b) \land \text{type}(b) \geq \text{type}(a). \end{cases}\]

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 flexibly 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 NodeType \(N_3\) since \(S\) exposes a property \(p_1\) different from the property \(pA\) exposed by \(N_3\). It is easy to see that Def. 12 permits \(S\) to flexibly match \(N_3\) (viz., \(S \sim N_3\)) if the type of \(p_1\) extends or is equal to the type of \(pA\) and if \(p_1\) and \(pA\) — even if syntactically different — refer to the same property (viz., \(\text{name}(p_1) \times \text{name}(pA)\)). 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:

\[
\text{Start} : \emptyset \rightarrow \emptyset, \quad \text{InstallPkg} : \{\text{name}\} \rightarrow \{\text{succeeded}\}, \quad \text{and} \quad \text{Shutdown} : \emptyset \rightarrow \emptyset.
\]

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

\[
\text{Start} : \emptyset \rightarrow \emptyset, \quad \text{Shutdown} : \emptyset \rightarrow \emptyset, \quad \text{Retrieve} : \{\text{pkgName}\} \rightarrow \{\text{url}\},
\]

\[
\text{Download} : \{\text{url}\} \rightarrow \{\text{sourcePath}\}, \quad \text{and} \quad \text{Install} : \{\text{sourcePath}\} \rightarrow \{\text{installed}\},
\]

with name $\equiv$ pkgName and succeeded $\equiv$ installed. For the sake of simplicity we also assume that name($x$) $\equiv$ name($y$) implies type($x$) = type($y$).

4.2. White-box matching

A ServiceTemplate $S$ that does not flexibly match a NodeType $N$ 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 $N$ ($S \square N$) iff:
(1) $\text{Reqs}(S) \sim_R \text{Reqs}(N)$ and
(2) $\text{Caps}(S) \sqsubseteq_C \text{Caps}(N)$ and
(3) $\text{Pols}(S) \equiv_{\text{PO}} \text{Pols}(N)$ and
(4) $\text{Props}(S) \sqsubseteq_{FR} \text{Props}(N)$ and
(5) $\text{Ints}(S) \sqsubseteq_I \text{Ints}(N)$.

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 $S \rightarrow E$ 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:

$\text{Caps}(S) \sqsubseteq_C \text{Caps}(N)$ iff $\forall c_N \in \text{Caps}(N) \exists c_S :$

$$
(c_S \in \text{Caps}(S) \lor
(\exists E : S \rightarrow E \land E \text{ is NodeTemplate} \land c_S \in \text{Caps}(E)))
\land
(name(c_S) \bowtie name(c_N) \land \text{type}(c_S) \geq \text{type}(c_N)).
$$

$\text{Props}(S) \sqsubseteq_{FR} \text{Props}(N)$ iff $\forall p_N \in \text{Props}(N) \exists p_S :$

$$(p_S \in \text{Props}(S) \lor
(\exists E : S \rightarrow E \land (E \text{ is NodeTemplate or RelationshipTemplate})
\land p_S \in \text{Props}(E)))
\land
(name(p_S) \bowtie name(p_N) \land \text{type}(p_S) \geq \text{type}(p_N)).$$

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:

$\text{Ints}(S) \sqsubseteq_I \text{Ints}(N)$ iff $\forall i_N, o_N : i_N \in \text{Ints}(N) \land o_N \in \text{Ops}(i_N) :$

$$
(\exists i_S, o_S : i_S \in \text{Ints}(S) \land o_S \in \text{Ops}(i_S) \land o_S \sim_o o_N)
\lor
(\exists p : p \in \Pi(S) \land [p] \sim_o o_N)
$$

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 $\text{Ops}$, returns a set of selectedOperations $\subseteq \text{Ops}$ that can be composed into a plan featuring the input-output behaviour of a given operation $op$. The algorithm inputs a set of
FindOperations\(\{\text{Ops, op, selectedOperations, needed, available}\}\) \{  
    needed = \(\{x \mid x \in \text{needed} \land \exists y \in \text{available} : y \triangleright x\}\);  
    if needed = \(\emptyset\) then return selectedOperations;  
    else \{  
      c = \text{choose}(\text{needed});  
      needed = needed \setminus \{c\};  
      opSet = \{o \in \text{Ops} \mid \exists d \in \text{O}(o) : d \triangleright c\};  
      if opSet = \(\emptyset\) then fail;  
      else foreach o ∈ opSet do \{  
        selectedOperations = selectedOperations \cup \{o\};  
        if \text{nonMinimal}(\text{selectedOperations}, \text{op}) then fail;  
        else \{  
          available = available \cup \text{O}(o);  
          needed = needed \cup \text{I}(o);  
          \text{FindOperations}(\text{Ops, op, selectedOperations, needed, available})  
        \}  
      \}  
    \}  
\}

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 \(\text{Ops}\), the operation \(\text{op}\) to be simulated, a (initially empty) set of \(\text{selectedOperations}\), the set \(\text{needed}\) of outputs to be generated (initially the outputs \(\text{O}(\text{op})\) of \(\text{op}\)), and the set of \(\text{available}\) outputs (initially the inputs \(\text{I}(\text{op})\) of \(\text{op}\)). First, if the set of \(\text{available}\) outputs includes an output “equal to or more general than” some \(\text{needed}\) output \(z\), then \(z\) is removed from the set of \(\text{needed}\) outputs (line 1). The notation \(y \triangleright x\) stands for \(\text{name}(y) \preceq \text{name}(x)\) and \(\text{type}(y) \geq \text{type}(x)\). Then, if there are no missing outputs to be generated the current set of \(\text{selectedOperations}\) 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 \(\text{Ops}\) 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 \(\text{Ops}\) producing an output equal to or more general than \(c\), \(o\) is added to the current set of \(\text{selectedOperations}\) (line 11). If the obtained set of \(\text{selectedOperations}\) is not minimal\(^{12}\) then (the instance of) the algorithm fails (lines 12 and 13). Otherwise the set of \(\text{available}\) outputs is extended with the outputs of \(o\) (line

\(^{11}\)Execution of \text{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 \text{nonMinimal} function, which can be found in [9]. Following [9], a set \(S\) of operations can simulate the input-output behaviour of an operation \(\text{op}\) iff (1) \(\forall x \in \text{O}(\text{op}) \exists y \in \bigcup_{o \in S} \text{O}(o) : y \triangleright x\), and (2) \(\forall y \in \bigcup_{o \in S} \text{I}(o) \exists x \in \bigcup_{o \in S} \text{O}(o) \cup \text{I}(\text{op}) : x \triangleright y\). A set \(S\) of operations that can emulate an operation \(\text{op}\) is minimal iff \(\notin S' \subset S\) that can emulate \(\text{op}\).
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 \( S_{\text{tmp}} \) flexibly matches NodeType \( N \), and the adaptation described in Sect. 4.1 can be now applied to build a ServiceTemplate \( S' \) having \( S_{\text{tmp}} \) 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 NodeTypeOS 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., \( [P] \sim_o \text{InstallPkg} \)). 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:

\[
P = \text{Retrieve} \cdot \text{Download} \cdot \text{Install}
\]

![Figure 16: White-box adaptation of a ServiceTemplate.](image)

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 ServiceTemplates

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 ServiceTemplates so as to exactly match the target NodeTypes. Namely, we show how to exactly match target NodeTypes by non-intrusively adapting flexibly matched ServiceTemplates and by intrusively adapting white-box matched ServiceTemplates.

![Figure 17: Available ServiceTemplate WebAppEnvironment.](image)

In doing so, we also provide some examples showing how to adapt the ServiceTemplate WebAppEnvironment (Fig. 17) to exactly match different NodeTypes. For the sake of simplicity, we will assume that each Capability or Requirement is of an homonym CapabilityType or RequirementType (e.g., the CapabilityWebAppRTE is of WebAppRTE CapabilityType, the RequirementSWContainer 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 ServiceTemplates

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 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

---

(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 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.

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 WebApp-Runtime 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).](image)

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).](image)

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 ServiceTemplates

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 ServiceTemplates.

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-
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 \( P_{\text{Start}} \) 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.

The obtained ServiceTemplate exposes all the features exhibited by the target NodeType IntegratedWebEnvironment. 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 ServiceTemplates. 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 ServiceTemplates and NodeType, 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 ServiceTemplates with respect to NodeType. 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 NodeTypes and ServiceTemplates, 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 NodeTemplates 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


