TOSCA-MART: A Method for Adapting and Reusing Cloud Applications

Jacopo Soldani[1], Tobias Binz[2], Uwe Breitenbücher[2], Frank Leymann[2], and Antonio Brogi[1]

[1]Department of Computer Science, University of Pisa
{*surname@di.unipi.it*}

[2]Institute of Architecture of Application Systems, University of Stuttgart
{*name.surname@iaas.uni-stuttgart.de*}

March 31, 2015

TOSCA-MART: A Method for Adapting and Reusing Cloud Applications[*]

Jacopo Soldani[1], Tobias Binz[2], Uwe Breitenbücher[2], Frank Leymann[2], and Antonio Brogi[1]

[1]Department of Computer Science, University of Pisa
{*surname@di.unipi.it*}

[2]Institute of Architecture of Application Systems, University of Stuttgart
{*name.surname@iaas.uni-stuttgart.de*}

Abstract

To fully appreciate cloud computing powers, design and development of cloud applications should be eased and supported. The OASIS TOSCA standard enables developers to design and develop cloud applications by specifying their topologies as orchestrations of typed nodes and relationships. However, building such application topologies often results in reinventing the wheel multiple times when similar solutions are manually created for different applications by different developers having the same requirements. Thus, the reusability of existing TOSCA solutions is crucial to ease and support design and development processes. In this paper, we tackle this issue. We introduce TOSCA-MART, a method that enables deriving valid implementations for custom components from a repository of complete and validated cloud applications. The method enables developers to specify individual components in their application topologies, and illustrates how to match, adapt, and reuse existing (fragments of) applications to implement these components while fulfilling all their compliance requirements. We also characterize and validate TOSCA-MART by means of a prototypical implementation based on an open source toolchain and a case study.

1 Introduction

Cloud computing recently gained a lot of attention due to its economical and technical benefits. However, current cloud technologies suffer from a lack of standardization, with various providers offering similar resources in a different manner [2]. Furthermore, reusing software artifacts in different cloud applications is a serious challenge due to technical as well as conceptual interoperability problems. As a result, to provision cloud applications on heterogeneous providers by fulfilling their individual requirements, developers are often asked to model and configure the whole middleware and infrastructure layers from scratch. This requires deep technical expertise, and results in error-prone development processes which significantly increase the cost of cloud application development,

[*]The research leading to these results has been supported by the European project Sea-Clouds (EU-FP7-ICT-610531) and by the BMWi project CloudCycle (01MD11023).

deployment and management. To tackle these issues, OASIS recently released the *Topology and Orchestration Specification for Cloud Applications* (TOSCA) [25], a standard to describe cloud applications in a portable and interoperable way. Based on application topologies (which define the structure of an application, as well as the concrete artifacts implementing its components) and on executable management plans, TOSCA-based applications can be deployed on, maintained in, and migrated across TOSCA-compliant cloud environments (e.g., OpenTOSCA [6]). Furthermore, TOSCA supports the reuse of individual application components by providing a type system, which enables the creation of modular building blocks to be reused for developing new applications [7, 24]. Although this eases modelling topologies, combining appropriate components and defining effective configurations for these combinations is still an open issue and mainly done in an ad-hoc manner. In addition, enterprises often define concrete compliance requirements that must be fulfilled by the applications, which makes the development of proper topologies even harder. Thus, we need a mean to enable reusing not only single components, but also complete topologies that are proven to be effective thanks to their employment in already existing enterprise applications [15].

In this perspective, we introduce TOSCA-MART (*TOSCA-based Method for Adapting and Reusing application Topologies*), a method which allows to concretely implement application components with certain requirements by adapting and reusing fragments of existing application topologies. More precisely, TOSCA-MART allows developers to define custom TOSCA application components by declaring the offerings and requirements they need to be properly operated. These features are then matched against those provided by each of the topologies available in a repository of existing cloud applications, so as to determine the topology fragments which are able to provide the desired features. Afterwards, TOSCA-MART automatically selects the "best" fragments among the matched ones and adapts them by creating new TOSCA specifications which fulfil the desired requirements. In this way, TOSCA-MART is able to discover complete topologies as well as middleware and infrastructure fragments to host new applications. Thus, instead of modeling complete topologies, application developers can define only the offerings and requirements (needed to deploy and manage their solutions) in terms of abstract components. TOSCA-MART will then automatically implement them, thus significantly decreasing the effort and cost needed for developing cloud applications.

The implementations are obtained by adapting the determined TOSCA topology fragments to *exactly* match desired components [13]. This suffices to reuse such fragments to deploy cloud solutions that rely on the desired components [14]. This is thanks to the powerful way in which TOSCA supports the processing of cloud application specifications. TOSCA permit to pack in a CSAR (*Cloud Service ARchive*) file an application specification together with the actual installable/executable files to be deployed/run 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 [24]. Therefore, in order to adapt a fragment of a specification to deploy an application that relies on a desired component c, it suffices to adapt such fragment into a new specification that matches c — without having to generate any implementation of the specified adaptation.

The rest of this paper is structured as follows: Sect. 2 overviews TOSCA and illustrates a first way to match TOSCA cloud applications. Sect. 3 illustrates a scenario which motivates the need for our approach. Sects. 4 and 5 detail and characterize the TOSCA-MART method, respectively. Sect. 6 evaluates TOSCA-MART by means of its prototype implementation. Finally, Sects. 7 and 8 discuss related work and draw some concluding remarks.

2 Background and Fundamentals

In this section we provide the fundamental notions needed to present the TOSCA-MART approach. Namely, we first overview the *Topology and Orchestration Specification for Cloud Applications* (TOSCA), and then we illustrate how to properly match the features of cloud applications described with such a specification.

2.1 TOSCA

TOSCA [25] is an emerging standard whose main goals are (i) to enable the specification of portable cloud applications and (ii) the automation of their deployment and management. In this perspective, TOSCA provides an XML-based modeling language which allows to formalize the structure of a cloud application as a typed topology graph, and the deployment/management tasks as plans. More precisely, each cloud application is represented as a `ServiceTemplate` (Fig. 1), which consists of an application topology (called `TopologyTemplate`) and a set of management `Plans`. The `TopologyTemplate` is essentially a typed

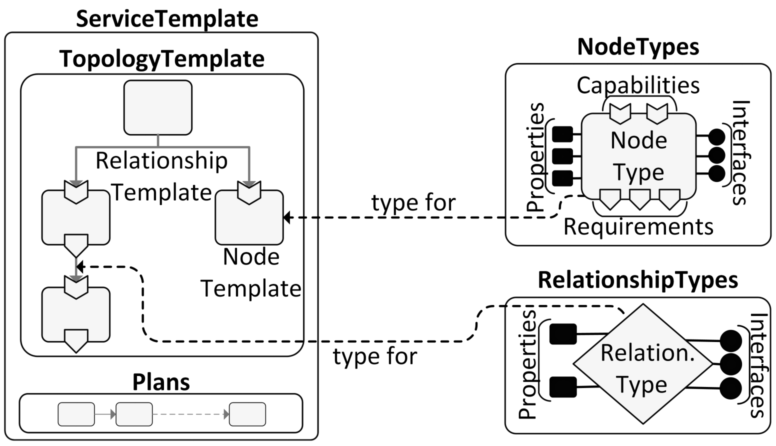


Fig. 1: TOSCA `ServiceTemplate`.

directed graph whose purpose is to describe the topological structure of the composite cloud application. Its nodes (called `NodeTemplates`) are the application components, while its edges (called `RelationshipTemplates`) are the relations between these application components. The connotation of such components and relations is defined by typing the aforementioned `NodeTemplates` and `RelationshipTemplates` by means of `NodeTypes` and `RelationshipTypes`, respectively. A `NodeType` defines (i) the observable properties of an application component, (ii) the management operations it offers, (iii) the requirements needed to properly operate it, and (iv) the capabilities it offers to satisfy other components' requirements. Syntactically speaking, properties are described with `PropertiesDefinitions`, operations with `Interface` and `Operation` elements, requirements with `RequirementDefinitions` (of certain `RequirementTypes`), capabilities with `CapabilityDefinitions` (of certain `CapabilityTypes`). `Plans` enable the description of application deployment and/or management aspects. Each `Plan` is a workflow that orchestrates the operations offered by the application components (i.e., `NodeTemplates`) to address (parts of) the management of the whole cloud application. Properties, capabilities, requirements and operations externally exposed by a `ServiceTemplate` can be described in its `BoundaryDefinitions`.

2.2 Matching cloud applications in TOSCA

In our previous work [13], we formally defined when a `ServiceTemplate` *exactly* or *plug-in* matches a desired `NodeType`. A `ServiceTemplate` S *exactly* matches a `NodeType` N (viz., $S \equiv N$) if the capabilities, requirements, properties, policies and interfaces exposed by S exactly match those of N, namely: (i) the requirements, capabilities and properties of S and N have the same name and type, and they are in a one-to-one correspondence, (ii) the policies exposed by S are applicable to N, and (iii) the interfaces of S and N have the same name, contain the same operations, and are in a one-to-one correspondence.

On the other hand, we say that a `ServiceTemplate` S *plug-in* matches a `NodeType` N (viz., $S \simeq N$) if, intuitively speaking, the former "requires less" and "offers more" than the latter. Namely, (i) for each requirement r of S there exists a requirement of N which has the same name as r and whose type is a sub-type of r's type, (ii) for each capability c of N there exists a capability of S which has the same name as c and whose type is a super-type of c's type, (iii) for each property p of N there exists a property of S which has the same name as p and whose (XML) type is a sub-type of p's type, (iv) the policies exposed by S are applicable to N, and (v) for each interface operation o of N there exists an operation of S which exactly matches o.

Consider for instance the `NodeType` N and the `ServiceTemplates` $S1$, $S2$ and $S3$ in Fig. 2[1], where C is a capability of type $CType$, R is a requirement of type $RType$, $p1$, $p2$ and $p3$ are string properties, $i1$, $i2$ and $i3$ are interfaces, and $o1$, $o2$, $o3$, $o4$, and $o5$ are operations. Assume also that the services $S1$, $S2$ and $S3$ do not have policies, and that same-named operations have the same input/output parameters. It is easy to see that $S1$ exactly matches N (i.e., $S1 \equiv N$) since $S1$ has the same capabilities, requirements, properties, and interfaces of N, while the same does not hold for $S2$ and $S3$ (i.e., $S2 \not\equiv N$ and $S3 \not\equiv N$). However, $S2$ plug-in matches N (i.e., $S2 \simeq N$) because $S2$ and N expose the same requirements and capabilities, and $S2$ features "more" properties and interface operations than N. On the other hand, $S3$ does not plug-in match N (i.e., $S3 \not\simeq N$) since their property names differ.

[1] For readability reasons, and since internal topologies are not considered by the our previous matchmaking approach, Fig. 2 focuses only on the boundaries of the available `ServiceTemplates`.

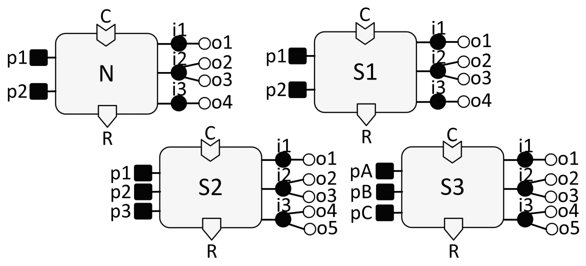


Fig. 2: Example of `NodeType` and `ServiceTemplate`s.

3 Motivating scenario

A typical issue that arises after finishing the development of Web-based applications is where to host them. Typically, multiple cloud providers are appropriate as many focus on such kind of applications (e.g., Amazon's and Google's PaaS and IaaS offerings). Thus, finding an appropriate provider (i.e., finding a *suitable* topology that describes the provisioning on this provider) is a time-consuming challenge. For instance, suppose that a Web application developer needs to host a PHP application on a cloud environment, along with a MySQL database containing application data. Currently, the developer is required to select the appropriate cloud provider and to explicitly describe the provisioning of her PHP application on this provider. Furthermore, in case she decides to move her application to another provider, this may require to re-describe (and re-implement) the deployment and management of the whole solution (even from scratch). It would be much better to abstractly describe the desired hosting environment and to provide such description as input to a tool which automatically derives a topology implementing the environment needed to deploy and manage the PHP application.

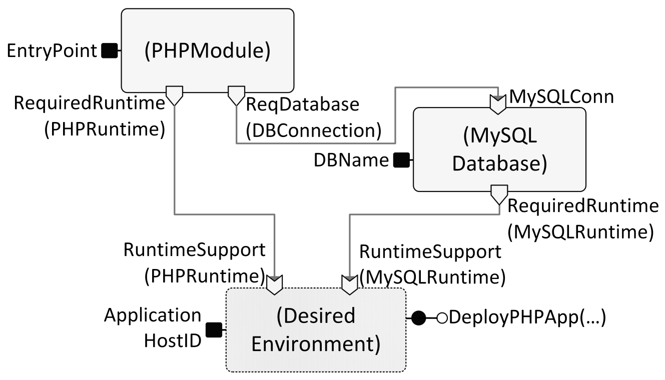


Fig. 3: Motivating scenario.

In TOSCA, this can be done as shown in Fig. 3. The environment required to host the PHP and MySQL modules is represented by a node whose type is `DesiredEnvironment`. This node is used by the application developer to describe the capabilities needed to host her application, to specify that the desired environment must provide an operation for deploying PHP applications, and to instruct that the environment's application ID must be available as property. Based on this simple `NodeType`, TOSCA-MART can derive a concrete implementation by searching among existing and validated cloud applications (as we will see in the next section).

4 The TOSCA-MART method

In this section, we illustrate TOSCA-MART as a solution to derive possible implementations of desired `NodeTypes` from a repository of validated cloud applications. We first overview the method as a whole, and then illustrate its steps separately.

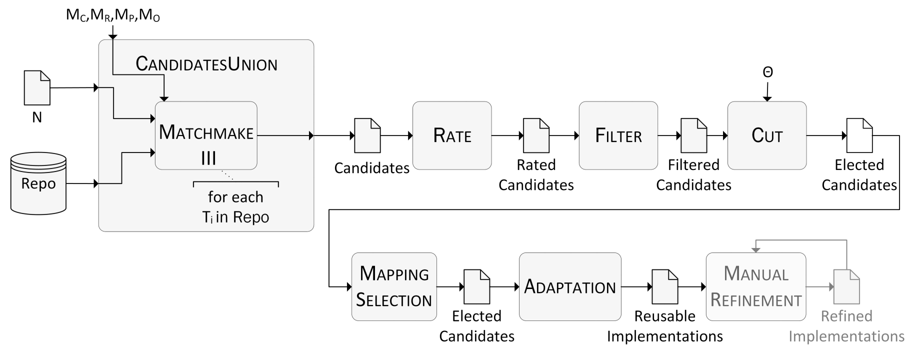


Fig. 4: The TOSCA-MART matchmaking and adaptation method.

4.1 Overview

Our goal is to derive an implementation for a target `NodeType` N by excerpting it from an repository *Repo* of cloud applications. Hence, N[2] and *Repo* must be input of the TOSCA-MART method illustrated in Fig. 4 (see Sect. 4.2). Once N and *Repo* are available, each application topology $T_i \in \mathit{Repo}$ is compared with N by employing the MATCHMAKE procedure. As a partial result, we obtain the set $\mathit{Candidates}_{T_i}$, whose elements are

$$\langle T_i, C, \{m_1, m_2, ..., m_n\}\rangle$$

(where C is a candidate fragment of the topology T_i, i.e., a fragment of T_i whose elements offers all the features declared in N, and m_i is a potential mapping between the features in N and those in C). Then, all the $\mathit{Candidates}_{T_i}$ are unified to obtain the set *Candidates*, containing all the candidate topology fragments (see Sect. 4.3). Due to the potentially huge number of already available topologies and to the possibility of having multiple candidates for each of these topologies, the set *Candidates* may become huge. Thus, providing the user with all these candidates is not appropriate. We reduce the number of

[2] In the following, we assume that N is defined in such a way that needed features are not redundant (e.g., it is not possible to match more than one capability of N with one of the available capabilities).

available candidates by employing three subsequent steps (see Sect. 4.4). First, Rate computes a score for each candidate using the rating function r. As a result, the set *Candidates* is transformed in the set *RatedCandidates*, whose elements are $\langle T_i, C, r(C), \{m_1, m_2, ..., m_n\}\rangle$. Second, Filter reduces the number of *RatedCandidates* by removing duplicates[3] (i.e., candidates that have the same topology fragment C, the same rating $r(C)$ and the same sets of potential mappings, independently from the topology T_i they come from). Finally, Cut reduces the number of candidates according to a threshold Θ. More precisely, the set *FilteredCandidates* is reduced to the set *ElectedCandidates*, which contains only the "best" Θ candidates (according to r). Afterwards, each of the *ElectedCandidates* has to be adapted to properly implement the target N. First, in order to avoid the user to select mappings on her own, we need to select the most proper mapping among the available ones. This is the purpose of the MappingSelection step, which can be implemented in various ways (e.g., ontologies, heuristics, compliance rules, etc.). Once the mappings are selected, each of the *ElectedCandidates* is adapted by resolving the unsatisfied dependencies of the selected components, and by enclosing the candidate fragments into standalone application specifications which implement the target `NodeType` N. All these specifications compose the set *ReusableImplementations*, which is the output of the TOSCA-MART method (see Sect. 4.5). Finally, an optional ManualRefinement step may be done to allow the cloud application developer to manually modify the outputted `NodeType` implementations, if they are not designed as desired.

4.2 Repository of application topologies

The repository *Repo* of application topologies is the knowledge base from which the TOSCA-MART method extracts the implementations for the target `NodeType` N. Thus, a prerequisite for this work is to have a large set of diverse topologies to be included in *Repo*. We include *application models*, which describe components, structure and configuration of applications. TOSCA application models can be retrieved from modelling environments (e.g, Winery [22]), as well as from configuration management systems. For instance, [27] shows how descriptions used by configuration management systems, such as *Chef*, can be wrapped into TOSCA application models. We can also include applications already operated in organizations (i.e., *application instances*). Despite such instances are usually not available as topology models, we can cope with them by employing Enterprise Topology Graphs (ETG) [10]. An ETG is a technically-detailed instance model that represents a snapshot of one or multiple applications, including all components, configuration and their relations. [8] shows how to semi-automatically create complete and technically detailed ETGs from existing enterprise applications. These ETGs can then be transformed into TOSCA topologies (as shown in [9]) so as to include them into *Repo*.

For instance, the repository *Repo* can be populated with the concrete application topologies illustrated in Fig. 5, namely with three instantiable topologies implementing (a) a *Moodle* application, (b) a *Wiki* application, and (c) a *SendSMS* web service. In the following, we will show how the TOSCA-MART

[3] Duplicates are maintained because they do not significantly impact on the approach's complexity and they allow the definition of smarter rating functions (e.g., by enabling to count how many times a topology fragment is recurring).

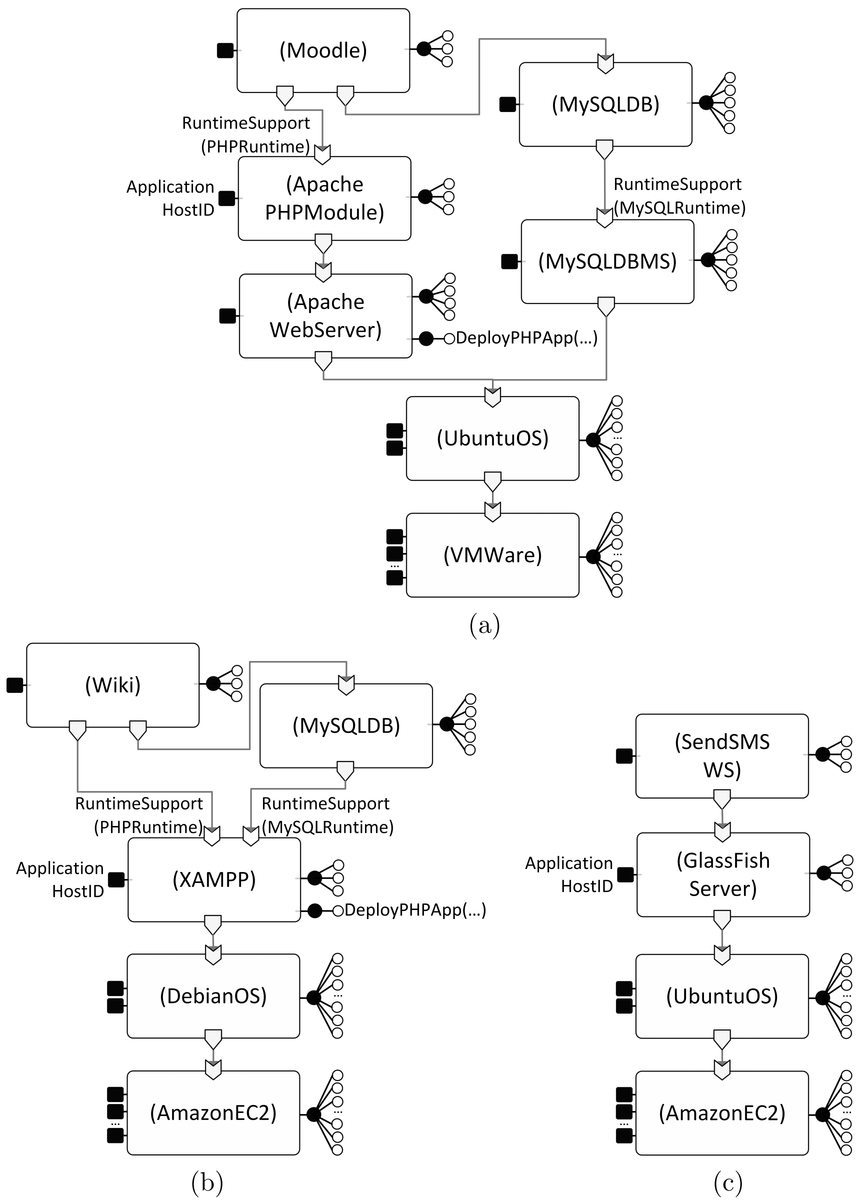


Fig. 5: Three examples of application topologies that can be included in *Repo*.

method leverage of this repository of instantiable topologies to derive an implementation for the `DesiredEnvironment` illustrated in Fig. 3. It is easy to see that the former two topologies are offering the desired features (i.e., the *Moodle* application offers all the features via the `ApachePHPModule`, `ApacheWebServer` and `MySQLDBMS` components, while the *Wiki* application offers them in a more integrated fashion via the `XAMPP` server). Thus, TOSCA-MART has to detect that both can be reused to implement the desired node, and (by supposing $\Theta = 1$) it also has to return only the adaptation of that having the highest rating. On the other hand, the topology implementing the *SendSMS* web service is offering only one of the desired features (i.e., the `ApplicationHostID` property via the `GlassFishServer`). Thus, TOSCA-MART has to discard it during the matchmaking phase (since it cannot be reused to implement the `DesiredEnvironment`).

4.3 Finding the candidate topology fragments

In order to find the candidate topology fragments, we apply the procedure MATCHMAKE to each topology $T_i \in \mathit{Repo}$, as shown in Fig. 4. This allows us to detect the sets $\mathit{Candidates}_{T_i}$, which are then unified by CANDIDATESUNION to obtain *Candidates*, the set of all candidate topology fragments. The MATCH-

```
MATCHMAKE($N, T, M_C, M_R, M_P, M_O$)
1  $mCaps = $ MATCHCAPS($M_C, \{\}, \mathsf{Caps}(N), \mathsf{Caps}(T), \{\}$);
2  if(($\mathsf{Caps}(N) \neq \{\} \wedge mCaps = \{\}$) return $\{\}$;
3  $mReqs = $ MATCHREQS($M_R, \{\}, \mathsf{Reqs}(N), \mathsf{Reqs}(T), \{\}$);
4  if($\mathsf{Reqs}(N) \neq \{\} \wedge mReqs = \{\}$) return $\{\}$;
5  $mProps = $ MATCHPROPS($M_P, \{\}, \mathsf{Props}(N), \mathsf{Props}(T), \{\}$);
6  if($\mathsf{Props}(N) \neq \{\} \wedge mProps = \{\}$) return $\{\}$;
7  $mOps = $ MATCHOPS($M_O, \{\}, \mathsf{Ops}(N), \mathsf{Ops}(T), \{\}$);
8  if($\mathsf{Ops}(N) \neq \{\} \wedge mOps = \{\}$) return $\{\}$;
9  $candidates = \{\}$
10 $mappings = mCaps \times mReqs \times mProps \times mOps$;
11 for each $m \in mappings$ {
12   $C = $ COLOUR($T, m$);
13   if $\exists \langle T, C', mappings_{C'} \rangle \in candidates : C = C'$
14     $mappings_{C'} = mappings_{C'} \cup \{m\}$;
15   else
16     $candidates = candidates \cup \{\langle T, C, \{m\} \rangle\}$;
17 }
18 return $candidates$;
```

Fig. 6: MATCHMAKE procedure.

MAKE procedure is listed in Fig. 6. Given the `NodeType` N and the topology T, it employs the procedure MATCHCAPS to check whether all the capabilities declared in N (viz., $\mathsf{Caps}(N)$) can be matched by those offered by the components of T (viz., $\mathsf{Caps}(T)$), according to the matchmaking operator M_C (e.g., M_C may be $\equiv_C$ or $\simeq_C$ — see Sect. 2.2). The detected capability mappings are stored in *mCaps* (line 1). Afterwards, MATCHMAKE checks whether all the required capabilities have been matched. If not, it ends by returning the empty set, which means that no fragments of T can match the target N (line 2). Analogously, the procedures MATCHREQS, MATCHPROPS, and MATCHOPS are employed to determine *mReqs*, *mProps*, and *mOps*, respectively (lines 3-8). Once the sets of potential mappings are available, MATCHMAKE starts computing the set of candidate topology fragments (line 9). First, all the possible combinations of *mappings* are created (line 10). Then, for each mapping m, COLOUR[4] determines the fragment C of T which exposes the features referred in m, and the

[4] Due to its straightforward behaviour, we omit the presentation of COLOUR. Essentially, it "colours" the elements of the topology which offer the matched features, and returns the set of coloured elements.

candidate $\langle T, C, .\rangle$ is added or updated in the set of *candidates* (lines 11-17). Finally, the set of *candidates* is returned (line 18).

As illustrated above, the MATCHMAKE procedure employs the MATCHCAPS, MATCHREQS, MATCHPROPS, and MATCHOPS to detect the subsets of available capabilities, requirements, properties and operations which match the set of desired ones. MATCHCAPS (Fig. 7) is a recursive procedure which inputs the

```
MATCHCAPS($M, matched, needed, available$)
1  if($\forall c_N \in needed,\ required(c_N) = 0$) return $\{matched\}$;
2  if($available = \{\}$) return $\{\}$;
3  select $Capability\ c_A$ from $available$;
4  $available' = available - \{c_A\}$;
5  $solutions$ = MATCHCAPS($M, matched, needed, available'$)
6  if($\exists\ CapabilityDefinition\ c_N \in needed$ :
         $c_A M c_N \wedge required(c_N) > 0$) {
7    $needed' = needed - \{c_N\}$,
8    $c_N{}' = c_N$;
9    $required(c_N{}') = required(c_N) - 1$;
10   if($required(c_N{}') > 0$)
11     $needed' = needed' \cup \{c_N{}'\}$
12   $solutions'$ = MATCHCAPS($M, matched \cup \{(c_N, c_A)\}$,
         $needed', available', \{\}$);
13   $solutions = solutions \cup solutions'$;
14 }
15 return $solutions$;
```

Fig. 7: MATCHCAPS procedure.

parameters M, *matched*, *needed*, and *available*. M is the matchmaking operator to be employed when comparing available capabilities with respect to the needed ones (e.g., $\equiv_C$, $\simeq_C$), while *matched*, *needed*, *available*, and *solutions* are the parameters used to maintain the state of the recursive computation (viz., *matched* contains the set of matchings discovered by the current instance of MATCHCAPS, *needed* contains the set of capability definitions which still need to be matched, and *available* contains the set of available capabilities). MATCHCAPS starts by checking whether there are no more *required*[5] capabilities in *needed*. If so, it returns *matched* since it contains a potential mapping between available and desired capabilities (line 1). It then check whether there no more *available* capabilities, which means that no mapping can be detected (line 2). If not, a capability c_A is removed from *available* (line 3-4), and the *solutions* without mappings to c_A are computed (line 5). Then, if c_A matches a needed capability c_N, a new instance of MATCHCAPS (with the sets *matched* and *needed* properly updated) is started so as to determine the solutions which comprise the mapping between c_N and c_A (line 6-12). The computed *solutions'* are then incorporated in the set *solutions* determined by the current instance (line 13). Finally, the set of computed *solutions* is returned (line 15). The other procedures (i.e., MATCHREQS, MATCHPROPS, and MATCHOPS) are analogous.

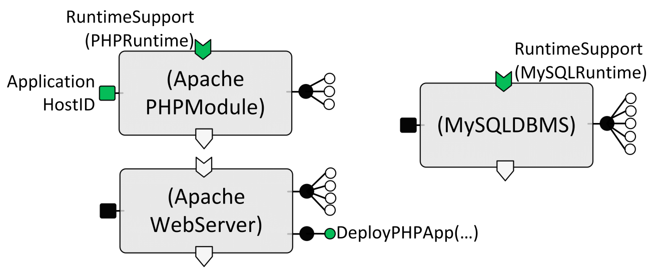
We are now able to matchmake N with respect to a single topology T of our

[5] *required* is defined as follows: If not explicitly assigned (as in line 10), $required(x)$ returns a default value. Such value is $x.lowerBound$ when x is a capabililty/requirement definition. Otherwise, it is 1.

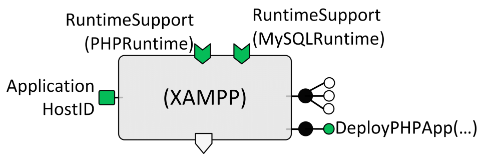
```
CandidatesUnion($N$, $Repo$, $M_C$, $M_R$, $M_P$, $M_O$)
1  $candidates = \{\}$;
2  for each $T_i \in Repo$ {
3    $newCandidates$ = Matchmake($N, T, M_C, M_R, M_P, M_O$);
4    $candidates = candidates \cup newCandidates$;
5  }
6  return $candidates$;
```

Fig. 8: CandidatesUnion procedure.

repository *Repo*. In order to matchmake N with the entire repository, we just have to iteratively apply Matchmake to all the topologies $T \in Repo$ and to unify the discovered candidates (Fig. 8). For instance, by iterating the Matchmake algorithm over the repository of applications in Fig. 5, we end up with the candidates illustrated in Fig. 9. The candidate (a) is composed by the software components of the *Moodle* application that offer the desired features, namely `ApachePHPModule`, `ApacheWebServer` and `MySQLDBMS`, while (b) is composed only by the `XAMPP` server belonging to the *Wiki* application, since it offers all the desired features in a more integrated fashion. On the other hand, the topology implementing the *SendSMS* web service is removed from consideration, since Matchmake fails at the very beginning (because none of the nodes appearing in *SendSMS*'s topology is offering the required `PHPHost` capability).



(a)



(b)

Fig. 9: Determined *Candidates*.

4.4 Election of the "best" candidate(s)

As we already mentioned, the number of detected candidates may be huge, and providing the user with all these candidates is not appropriate. We reduce them by employing the Rate, Filter and Cut steps (Fig. 4) presented in this section.

```
RATE($candidates, r$)
1  $ratedCandidates = \{\}$;
2  for each $\langle T, C, mappings_C \rangle \in candidates$ {
3    $r_C = r(C, mappings_C, candidates)$;
4    $ratedCandidates = ratedCandidates \cup$
        $\{\langle T, C, r_C, mappings_C \rangle\}$;
   }
5  return $ratedCandidates$;
```

Fig. 10: Rate procedure.

Rate (Fig. 10) inputs a set of *candidates* and a rating function r. It then constructs and outputs the set of *ratedCandidates* by applying r to each of the candidate topology fragments (lines 2-5). Please note that we do not prescribe which rating function r to employ, since this depends on what the user wants to privilege. For instance, in our reference example we may look for the most integrated solutions, i.e., we may try to minimize the amount of components appearing in a candidate:

$$r(C) = 1/|C|$$

(where C is a candidate, and $|C|$ is the number of components it contains)[6]. The candidates in Figs. 9.(a) and 9.(b) will then be rated 1/3 and 1, respectively. Accordingly, TOSCA-MART will privilege the latter with respect to the former.

Once the ratings are available, we can remove the "duplicates", namely the candidates having the same topology fragment C, the same rating r_C, and the same set of possible mappings $mappings_C$, independently from the topology T they come from. Please note, that both r_C and $mappings_C$ depend on the candidate topology fragment *elems*, since the former is a function of C, and the latter is the set of possible mappings between the features of N and those of the elements in C. Thus, we can consider duplicates those candidates having the same topology fragment C. This means that, in order to remove the duplicates from the output of Rate (by also optimizing the performances of the method), we can merge Rate and Filter into RateAndFilter (Fig. 11), so as to add candidates to the output only if they are not already there (lines 4-5). The procedure is also modified in such a way that its output is a list (instead of a set — line 2), whose elements are sorted in descending order according to the value of r_C (line 5).

Finally, Cut is implemented by cutting the list *ratedCandidates* so as to maintain only the first Θ elements (Fig. 12). The value of Θ depends on the usage context. For fully-automated approach, $\Theta = 1$ instructs TOSCA-MART

[6] There are many other possible rating functions. For instance, r could privilege not only the fragments having fewest components, but also the most frequent ones (by exploiting the amount of duplicates a candidate has).

$$r(C, candidates) = 1/|C| + \frac{duplicates(C, candidates)}{|candidates|},$$

where $duplicates(C, candidates)$ computes the number of duplicates of C among the *candidates*.

```
RateAndFilter($candidates, r$)
1  $ratedCandidates = []$;
2  for each $\langle T, C, mappings_C\rangle \in candidates$
4    if($\nexists\langle T', C', r_{C'}, mappings_{C'}\rangle \in ratedCandidates :$
       $C = C'$) {
5      $r_C = r(C, mappings_C, candidates)$;
6      add$_{sorted}$ $\langle T, C, r_C, mappings_C\rangle$ to $ratedCandidates$;
7  }
8  return $ratedCandidates$;
```

Fig. 11: RateAndFilter procedure.

to proceed with the highest rated candidate. For instance, with $\Theta = 1$, our reference example (Fig. 9) proceeds by electing (b) as the candidate to be adapted (since its rating is 1, while (a) has a rating of 1/3).

```
Cut($ratedCandidates, \Theta$)
1  for $i = |ratedCandidates|$ to $\Theta + 1$
2    remove $ratedCandidates[i]$ from $ratedCandidates$;
3  return $ratedCandidates$;
```

Fig. 12: Cut procedure.

4.5 Adaptation of the elected candidate(s)

The *ElectedCandidates* have to be adapted so as to become concrete implementations of the `NodeType` N. This is the purpose of the MappingSelection and Adaptation steps. For each candidate $\langle T, C, r_C, mappings_C\rangle$ in *ElectedCandidates*, MappingSelection determines the mapping $m_C \in mappings_C$, that makes the candidate work as implementation of N. Despite there is no chance to ensure that the selected mapping is the one the user desires, we can approach the problem in a heuristic way, by selecting the mapping m_C which maps each feature to the "uppermost" available and compatible one. As a result, each candidate $\langle T, C, r_C, mappings_C\rangle$ is transformed into $\langle T, C, r_C, m_C\rangle$, where m is the selected mapping to be employed. The selected mapping is then employed by the Adaptation procedure to transform the available TOSCA definitions in a standalone implementation of N. First, the unsatisfied dependencies of the elements in *elems* must be resolved. This is done by applying the following rules until the set C is no more modified by their operation:

A1) For each application component in C, its outgoing relationships must be added to C, if not already present. This rule does not affect the outgoing relationships whose sources are requirements that have been matched with those of the target node N.

A2) For each relationship in C, the components representing its source and target must be added to C, if not already present.

Once all (unsatisfied) dependencies have been processed, the actual adaptation can take place. The adaptation is analogous to the one proposed in [13, 14]: (i) we create a new `ServiceTemplate` which contains the application components and relationships stored in C, (ii) we define its `BoundaryDefinitions` by exposing only the features declared by N, and (iii) we employ m_C to map these features to the corresponding ones exhibited by the elements of the topology. The resulting `ServiceTemplate` *exactly* matches the desired `NodeType` N, and thus can be employed to concretely implement and substitute N.

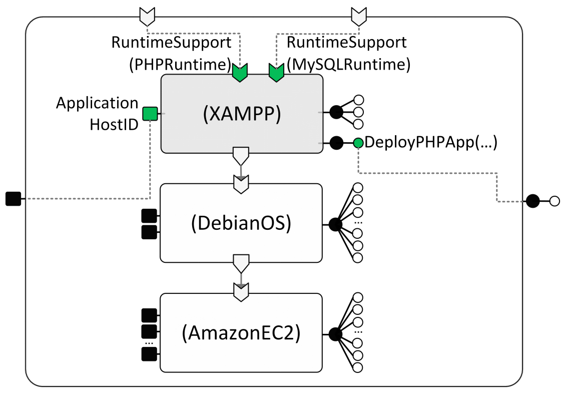


Fig. 13: Implementation derived by TOSCA-MART.

In our reference example, TOSCA-MART follows this approach to adapt the candidate fragment in Fig. 9(b). Namely, MappingSelection selects the only available mapping and Adaptation starts extending the fragment according to A1 and A2. First, A1 causes the introduction of the `XAMPP` node's outgoing relationships. Then, A2 causes the introduction the `DebianOS` operating system component. Similarly, Adaptation introduces the relationship starting from `DebianOS`, as well as the `AmazonEC2` virtual machine, and this makes the candidate fragment be no more modifiable by A1 and A2. Adaptation then employs this fragment as the topology of a new `ServiceTemplate`, say `DesiredEnvironmentImplementation`. It then defines the boundaries of the new `ServiceTemplate` according to the selected mapping (Fig. 13). As a result, `DesiredEnvironmentImplementation` exactly matches the `Desired-Environment` target `NodeType` (Fig. 3), thus being a valid implementation for such a `NodeType` [15].

4.6 TOSCA-MART

All the aforementioned procedures must be orchestrated so as to operate the TOSCA-MART method illustrated in Fig. 4. This can be easily done by implementing the procedure in Fig. 14. First, we need to invoke CandidatesUnion so as to derive all the *candidates* which can be excerpted from the topologies in *Repo* (line 2). These *candidates* are rated and filtered by employing the RateAndFilter procedure (line 3). The obtained *filteredCandidates* are then reduced by Cut to the "best" Θ ones (line 3), whose mapping is subsequently determined by MappingSelection (line 4). Finally, the resulting *mappedCandidates* are given to ADAPTATION so as to generate the *reusable-Implementations* to be returned by TOSCA-MART (lines 5-6).

```
TOSCA-MART$(N, T, r, \Theta, M_C, M_R, M_P, M_O)$
1  $candidates$ = CANDIDATESUNION$(N, T, M_C, M_R, M_P, M_O)$;
2  $filteredCandidates$ = RATEANDFILTER$(candidates, r)$;
3  $electedCandidates$ = CUT$(filteredCandidates, \Theta)$;
4  $mappedCandidates$ =
      MAPPINGSELECTION$(electedCandidates)$;
5  $reusableImplementations$ =
      ADAPTATION$(mappedCandidates)$;
6  return $reusableImplementations$;
```

Fig. 14: TOSCA-MART orchestrating procedure.

5 Properties of the approach

In this section we informally illustrate the termination, soundness, and time complexity of TOSCA-MART. In the following, please remember that the definitions of the topologies $T_i \in Repo$, as well as that of the target `NodeType` N, are necessarily finite.

5.1 Termination

According to Fig. 14, the termination of TOSCA-MART directly follows from that of its steps. First, we need to ensure that MATCHMAKE (Fig. 6) eventually terminates. Consider MATCHCAPS (Fig. 7), which recurs on the set of *available* capabilities until it becomes empty. Each recursive invocation is performed after the removal of a capability from the set *available*. Thus, since *available* is initially finite, it eventually becomes empty, causing the termination of MATCHCAPS (which returns a finite set of *solutions*). Since the same holds for MATCHREQS, MATCHPROPS, and MATCHOPS, to prove the termination of MATCHMAKE, we just need to ensure that lines 8-14 eventually terminate. The set *mappings* is computed as the product of finite sets, and thus both its computation and cardinality are finite. This, along with the fact that COLOUR can at most "colour" the whole finite topology, implies that the loop at lines 9-14 eventually terminates. Thus, MATCHMAKE eventually terminates.

The termination of CANDIDATESUNION, RATEANDFILTER, and CUT (Figs. 8, 11, and 12) obviously follows from the fact that MATCHMAKE produces a finite set of *Candidates*. Thus, we only have to prove the termination of MAPPING-SELECTION and ADAPTATION (Sect. 4.5) MAPPINGSELECTION selects one of the potential mappings, for each of the candidates. Since the set of candidates and those containing their mappings are finite, we can conclude that MAPPING-SELECTION eventually terminates. This (along with the fact that the generation of the adapted `ServiceTemplate`s obviously terminates) implies that to prove the termination of ADAPTATION we just need to ensure that A1 and A2 eventually become no more applicable (see Sect. 4.5). This can happen only if the size of the fragment can eventually be no more increased by their operation, which is true because the fragment is upperbounded by the topology it comes from. It follows the termination of ADAPTATION, and thus that of TOSCA-MART.

5.2 Soundness

We want to ensure that TOSCA-MART returns Θ standalone `ServiceTemplate`s which exactly match the target `NodeType` N (see Sect. 2.2), by properly adapting the Θ candidates having the highest ratings.

First, we have to ensure that CANDIDATESUNION (Fig. 8) computes all possible candidates (with all possible mappings) which can be excerpted from the cloud application topologies in *Repo*. This directly follows from the fact that MATCHMAKE computes all the possible mappings (and thus all the possible candidates) which can be derived from a single topology: Suppose (by contradiction) that MATCHMAKE misses one of these mappings. This can happen only if (at least) one among the procedures MATCHCAPS, MATCHREQS, MATCHPROPS, and MATCHOPS misses a mapping between a *needed* and an *available* feature. Suppose (without loss of generality) that MATCHCAPS misses a mapping between a *needed* capability definition c_N and a matching *available* capability c_A. This can happen only if the pair (c_A, c_N) is never added to a *matched* set, which in turns requires c_A to be never analyzed (otherwise, since c_A matches c_N, line 12 would add (c_A, c_N) to a set of *matched* pairs). Nevertheless, according to the recursive definition of MATCHCAPS (Fig. 7), c_A is eventually analyzed, and thus we come to a contradiction which allows us to deduce what we wanted to prove.

Then, we have to ensure that RATEANDFILTER and CUT remove the duplicates and reduce the set of available candidates to the Θ highest rated ones. This can be easily deduced from their definition (Figs. 11 and 12). RATEANDFILTER avoids the insertion of duplicate candidates through the check at line 4 and outputs the list of rated Candidates sorted in descending order, according to r. Afterwards, CUT removes all the candidates whose index is higher than Θ, thus maintaining only the Θ candidates having the highest ratings.

For each of the Θ candidates, MAPPINGSELECTION selects a mapping among those available. Thus, after this step, ADAPTATION is provided with the Θ candidates with the highest value of r, and each of them only contains one mapping.

Finally, ADAPTATION has to ensure that each of the Θ candidates is transformed into a `ServiceTemplate` which (a_1) is standalone and (a_2) exactly matches N. (a_1) means that all the dependencies of the elements composing a candidate are satisfied, which can be easily proven by relying on the eventual non-applicability of the rules A1 and A2. (a_2) is ensured by the fact that the boundaries of each returned `ServiceTemplate` are built by including all the features declared by N (and by employing the selected mapping to map such features onto the internal ones which have been matched). Since the output of ADAPTATION is also the output of TOSCA-MART, it follows what we wanted to prove.

5.3 Time complexity

The time complexity of TOSCA-MART is given by the maximum among the complexities of its step. Consider MATCHCAPS, and let $a = |available|$ and $n = |needed|$. In the base case, MATCHCAPS goes through the set of *needed* features, and thus its complexity can be approximated with $O(n)$. Otherwise, its complexity is dominated by the two recursive calls (whose a is decreased by 1) and by the union of the disjoint sets *solutions* and *solutions*$'$. This, along with the fact that the size of *solutions* and *solutions*$'$ can be upperbounded by the size 2^{an} of the power set of the cartesian product $\mathit{available} \times \mathit{needed}$, allows us to derive the following recurrence relation[7]:

$$T(a) = \begin{cases} O(n) & \text{if } a = 0 \\ 2T(a-1) + O(an) & \text{if } a > 0 \end{cases}$$

By induction on t, it is possible to prove that the solution of the above relation is $T(a) = O(2^a n)$. This, along with the fact that initially $a = \max_{T \in \mathit{Repo}} |\mathsf{Caps}(T)|$ and $n = \mathsf{Caps}(N)$, allows us to conclude that

$$T(\textsc{MatchCaps}) = O(2^{\mathsf{Caps}(T)}\mathsf{Caps}(N)).$$

Furthermore, since each recursive invocation having the set $\mathit{available} = \{\}$ can at most generate one mapping, we deduce that the maximum number of mappings can be $2^{\mathsf{Caps(T)}}$ (each of which contains $\sum_{x \in \mathsf{Caps}(N)} \mathit{required}(x)$ mappings). Similarly:

$$T(\textsc{MatchReqs}) = O(2^{\mathsf{Reqs}(T)}\mathsf{Reqs}(N)),$$
$$T(\textsc{MatchProps}) = O(2^{\mathsf{Props}(T)}\mathsf{Props}(N)),$$
$$T(\textsc{MatchOps}) = O(2^{\mathsf{Ops}(T)}\mathsf{Ops}(N)),$$

and the produced mappings can be upperbounded with $2^{\mathsf{Reqs(T)}}$, $2^{\mathsf{Props(T)}}$, and $2^{\mathsf{Ops(T)}}$, respectively.

Consider now Matchmake, which is dominated either by the matching procedures (lines 1-4) or by the generation of the *candidates* (lines 9-14). By properly combining the above computed quantities of mappings, we can conclude that the set *mappings* can contain (at most) 2^t mappings, each consisting of n pairs, where:

$$t = |\mathsf{Caps}(T) \cup \mathsf{Reqs}(T) \cup \mathsf{Props}(T) \cup \mathsf{Ops}(T)|$$
$$n = \Sigma_{x \in \mathsf{Caps}(N) \cup \mathsf{Reqs}(N) \cup \mathsf{Props}(N) \cup \mathsf{Ops}(N)} \mathit{required}(x)$$

Thus, *candidates* can be generated with a time complexity of $O(2^t n)$, which is higher than those of the matchmaking procedures. It follows that

$$T(\textsc{Matchmake}) = O(2^t n).$$

However, in practice we have $n \ll t$, and this allows us to approximate $T(\textsc{Matchmake})$ as follows.

$$T(\textsc{Matchmake}) = O(2^t).$$

From $T(\textsc{Matchmake})$, we can deduce the complexity of CandidatesUnion. Since the latter performs the union of disjoint sets, we can approximate the complexity of CandidatesUnion as that which comes out by operating Matchmake against each topology in *Repo*, namely

$$T(\textsc{CandidatesUnion}) = O(r2^t).$$

[7] According to [17], the union of two disjoint sets having size 2^s leads to a worst case complexity of $O(s)$.

where $r = |Repo|$. Furthermore, since all the remaining activities are dominated by set operations performed against *candidates* (which can be viewed as a different representation of the above counted mappings), the steps from RateAndFilter afterwards can lead to a complexity which is at most $O(r2^t n)$. Thus, the worst-case complexity of TOSCA-MART is

$$T(\text{TOSCA-MART}) = O(r2^t).$$

6 Prototype and Performances

To illustrate the feasibility of TOSCA-MART, we implemented a Java prototype integrated into the open source *OpenTOSCA* ecosystem [6, 12, 22]. As

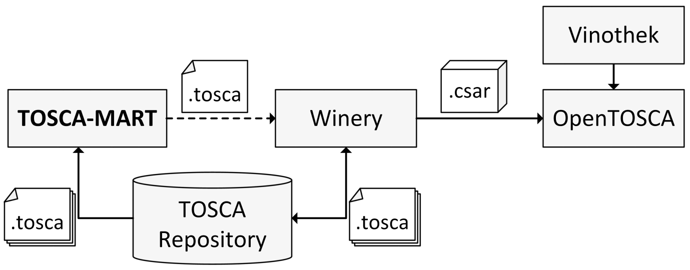


Fig. 15: TOSCA-MART in the OpenTOSCA ecosystem.

shown in Fig. 15, our prototype processes TOSCA specifications taken from a shared *TOSCA Repository*. It then produces new specifications which can then be imported into *Winery* [22], the graphical TOSCA modeling tool of the aforementioned open source ecosystem. As we already mentioned in the introduction, the creation of new application specifications does suffice to enact the actual reuse of the matched fragments. Indeed, by employing Winery to replace the available specifications with the adapted ones, the existing TOSCA application packages (i.e., *Cloud Service ARchive*s) can be processed as if they were only implementing the to-be-reused fragment. This is because each CSAR is processed by TOSCA containers (like OpenTOSCA [6]) according to the cloud application specification it contains [24]. From the above, it follows that the adapted CSARs can also be provided to users through the Vinothek self-service portal [12].

We comparatively assessed the TOSCA-MART prototype with respect to a GREEDY implementation of our previous matchmaking and adaptation approach [13, 14]. With respect to TOSCA-MART, GREEDY access randomly the shared repository and returns the first specification matched according to the definitions we recalled in Sect. 2.2. We employed both prototypes to automatically generate an implementation of `DesiredEnvironment` (see Sect. 3) by relying on a plastic repository containing 279 validated TOSCA solutions. Among them, 135 application can offer the desired features, and only 27 can gain the highest rating. In the following, we illustrate the time performances of both prototypes[8].

[8] All tests were repeated 300 times on a Windows 8.1 machine having an AMD A6-5400K APU (3.60GHz) and 4 GBs of RAM.

6.1 Time Performances

We evaluated the time performances of TOSCA-MART and GREEDY with respect to the size r of the available repository, the maximum amount t of features available in a topology, and the amount n of features described in the desired `NodeType`. In order to test the prototypes under the same input conditions, we varied r by repeating multiple times the applications appearing in the repository, and t by making each application composed by multiple copies of its starting topology.

As expected, the completion time of TOSCA-MART grew linearly with respect to growth of r (Fig. 16.(a)). The expectations were respected also when varying n, since the completion time was independent from n itself (Fig. 16.(b)). This is because we had $n \ll t$, and this allows us to approximate $T(\text{TOSCA-MART})$ with $O(r2^t)$, which is independent from n (see Sect. 5.3). On the other hand, when we increased t, the completion time was not growing exponentially as expected. It instead grew linearly (Fig. 16.(c)). This is because, by properly implementing the matchmaking operators, the situations in which MATCHCAPS, MATCHREQS, MATCHPROPS, and MATCHOPS performed two recursive calls became negligible with respect to those in which they performed one recursive call. It follows that their complexity, as well as the number of detected mappings, can be approximated with $O(t)$, which in turn implies that $T(\text{TOSCA-MART}) = O(rt)$. From the above, it follows that when (i) $n \ll t$ and (ii) the repository and matchmaking operators are such that matchings is negligible with respect to that of non-matchings (see Fig. 7, line 5), we have that the time complexity of TOSCA-MART can be approximated with

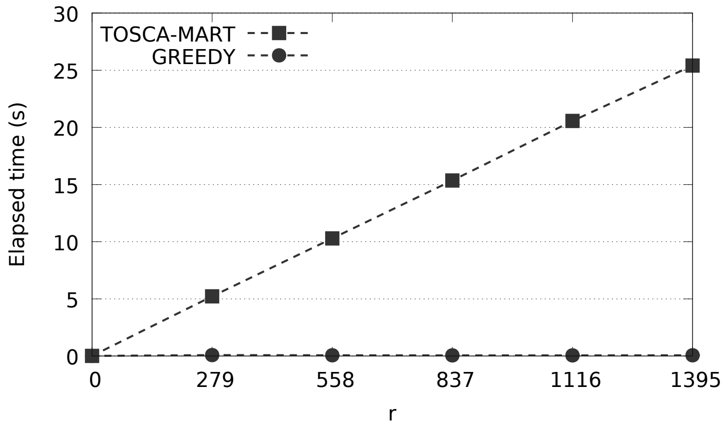
$$T(\text{TOSCA-MART}) = O(rt).$$

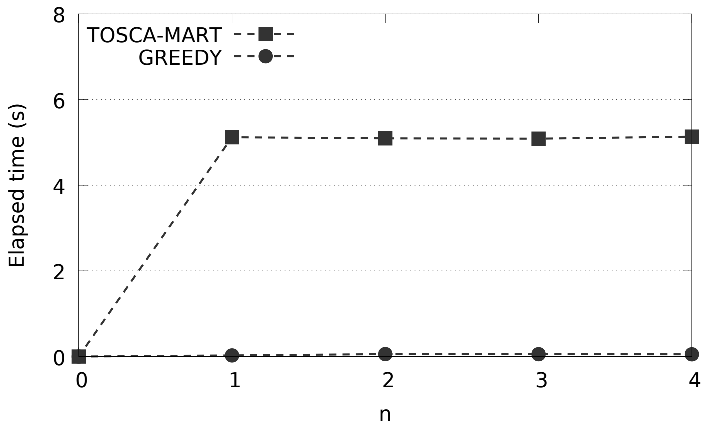
Please note that, in practice, these conditions are most probably true: (i) the features declared on a component are much fewer than those available in complex applications, and (ii) each complex application is composed by heterogeneous components offering different features, thus causing a negligible amount of matches among the performed checks — if the employed matchmaking operators are not dummy.

As shown in Fig. 16, we also compared the time performances of TOSCA-MART with respect to those registered by the GREEDY (in the luckiest case of having all applications exposing all available features on their boundaries). As expected, TOSCA-MART always required a completion time much higher than that of GREEDY, since the former always analyze all available applications, while the latter returns the first match. This is the price for providing the user with the topology fragments that best match the desired nodes, instead of providing the first match as a whole. However, it is worth noting that the development of complex application topologies is a process requiring days to be performed. Despite our solution requires a few seconds to complete, it allows cloud application developers to drastically reduce the time and effort they currently devote to the implementation of their cloud solutions.

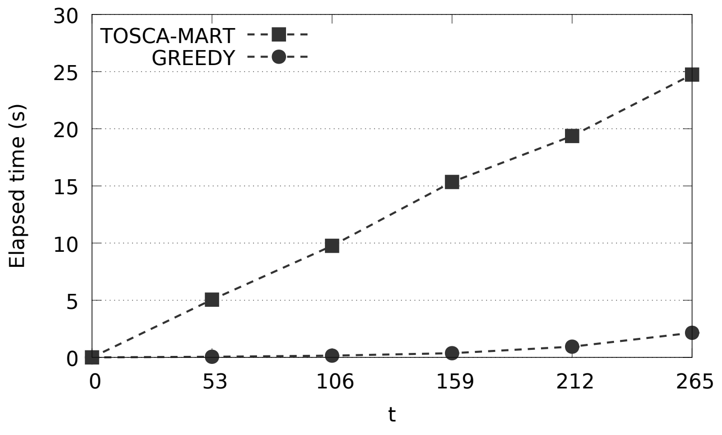
7 Related work

The development of systematic approaches to adapt and reuse existing software is widely recognized as one of the crucial problems in software engineering





(a) (b)



(c)

Fig. 16: Time performances.

[16, 28]. In spite of the increasing availability of cloud solutions, currently platform-specific code often needs to be manually modified to reuse existing solutions in cloud-based applications. This is obviously an expensive and error-prone activity, as pointed out in [26], 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 applications. For instance, [23] and [20] propose two approaches to transform platform-agnostic source code of applications into platform-specific applications, provided that they developed according to model-driven methodologies. 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 non open-source, third-party services. [19] proposes a framework which allows developers to develop cloud-based 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 that of [19] in three ways. First, only some cloud platforms are targeted by [19], while our method can be applied on any (TOSCA-compliant) platform. Moreover, in their approach the reuse of a cloud service requires invoking the middleware layer, while in our method adaptation is performed only once. Finally, [19] always requires to write source code, while our method 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-based services. In contrast, our method proposes a way to adapt existing cloud applications, by relying on TOSCA [25] as the standard for cloud interoperability, and to support an easy reuse of third-party services.

In this perspective, we are in line with the research directions individuated by [15], and in particular with the easy reuse of existing TOSCA specifications. We already proposed a way to instantiate desired `NodeType`s by reusing entire `ServiceTemplate`s in [13, 14]. TOSCA-MART goes a step further, by allowing to reuse not only entire application topologies, but also fragments of such topologies. Other approaches may be those proposed by [3, 4], and [1]. [3, 4] provide a method to deploy and provision SOA solution, in which patterns are used to structure and constraint composite applications, without binding to specific resources, and without specifying provisioning actions. [1] propose a way to detect the optimal deployment for a cloud-based solution, by employing a highly abstracted topology. Nevertheless, these approaches require an application developer to structure the topology of the IT solution to provision, and based on this structure the provisioning can be detected. In contrast, TOSCA-MART gives developers all the freedom in choosing whether to structure the whole topology or to abstract all the needed components as a single (and standalone) module. The latter solution obviously eases the development of cloud applications, thus flatting the learning curve needed to provision them. Furthermore, [3, 4], and [1] focus only on the provisioning of cloud application, while TOSCA-MART gives developers the means to also describe the configuration operations needed to manage their applications.

Finally, it is worth noting that the novelty of TOSCA-MART does not reside in the type of adaptation techniques applied to `ServiceTemplate`s. Indeed, our method exploits well-know adaptation patterns (e.g., [18, 5]) to adapt TOSCA templates. The novelty of TOSCA-MART is rather that, in contrast with traditional adaptation approaches (e.g., [11, 21]), no additional code must be developed to reuse existing cloud-based services. We exploit the possibilities provided by TOSCA of mapping exposed features onto internal ones, and of entirely delegating the management of such mappings to TOSCA containers [24].

8 Conclusions

Migrating applications across different clouds often results in remodeling and reconfiguring the whole middleware and infrastructure layers from scratch. This requires technical expertise and results in error-prone development processes, thus increasing the costs for developing, deploying and managing cloud applications. In this paper, we reduced these costs by introducing TOSCA-MART, a method which allows developers to abstractly describe application components, and to automatically excerpt their implementations from a repository of existing cloud application topologies. Since TOSCA-MART is able to discover complete topologies, as well as middleware and infrastructure fragments to host new applications, developers do not have to model complete topologies any more. Instead, they only have to pay the effort of defining (in terms of the to-be-implemented components) the compliance requirements which must be satisfied to properly run and manage their applications.

Such an approach would be really useful also in the emerging field of *docker*[9]. Docker is an open platforms allowing developers and system administrators to build, ship, and run distributed applications. More precisely, docker enables composite applications to be manually assembled from existing components in a way pretty much similar to that of TOSCA. Thus, by modeling docker components through TOSCA `NodeType`s and docker applications through TOSCA `ServiceTemplate`s, our approach can be applied as-is to detect also fragments of existing docker solutions.

Finally, it is worth stressing that the development of complex applications, as well as of the topologies needed to provision and manage them, is a process requiring days to be performed. This means that TOSCA-MART helps developers to significantly reduce the time needed for building their solutions, even if it may require an exponential time complexity. As we highlighted in Sect. 6, this exponential complexity holds in theory, while in practice TOSCA-MART will most probably exhibit a linear time complexity. By employing optimization techniques (e.g., parallelization, pre-fetching, etc.), its completion time can be further decreased, and by returning partial results (like in flight booking services), the user may not be annoyed by the seconds spent in computing the results. In addition, it is important to stress that, thanks to the powerful way in which TOSCA supports the processing of cloud application specifications, the proposed adaptation of matched TOSCA topology fragments does suffice to reuse such fragments to deploy cloud solutions that rely on the desired components [14]. Namely, in order to adapt an application fragment to deploy a solution that relies on a desired component c, it suffices to adapt such fragment into a new specification that matches c — without having to generate any im-

[9] `https://www.docker.com`

plementation of the specified adaptation.

The extensions of this work we are going to immediately investigate are twofold. First, (i) we shall understand whether there exists specific use case in which it is required to explicitly rate the mappings, so as to provide the user with the set of most promising mappings for each candidate (instead of selecting only one). If so, we aim at extending our approach by treating the MAPPING-SELECTION in a way which is similar to that actually devoted to the selection of candidates (i.e., by employing subsequent steps analogous to RATEANDFILTER and CUT). Second, (ii) since our method only focuses on the topologies of TOSCA applications, it produces implementations which can only be *declaratively* processed [24]. In order to cope also with *imperative processing*, we aim at extending TOSCA-MART by also considering management plans when excerpting the implementation of an abstract component from a repository of validated application topologies.

References

[1] Vasilios Andrikopoulos, Santiago Gomez Saez, Frank Leymann, and Johannes Wettinger. Optimal distribution of applications in the cloud. In *Proceedings of the 26th International Conference on Advanced Information Systems Engineering (CAiSE'14)*, 2014.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commununications of the ACM*, 53(4):50–58, 2010.

[3] William Arnold, Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, and Alexander Totok. Pattern based SOA deployment. In Bernd J. Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Proceedings of the 5th International Conference on Service-Oriented Computing (ICSOC'07)*, volume 4749 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2007.

[4] William Arnold, Tamar Eilam, Michael H. Kalantar, Alexander V. Konstantinou, and Alexander Totok. Automatic realization of SOA deployment patterns in distributed environments. In Athman Bouguettaya, Ingolf Krüger, and Tiziana Margaria, editors, *Proceedings of the 6th International Conference on Service-Oriented Computing (ICSOC'08)*, volume 5364 of *Lecture Notes in Computer Science (LNCS)*, pages 162–179. Springer, 2008.

[5] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli. Towards an Engineering Approach to Component Adaptation. In *Architecting Systems with Trustworthy Components*, volume 3938 of *Lecture Notes in Computer Science (LNCS)*, pages 193–215. Springer, 2006.

[6] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner. OpenTOSCA – a runtime for TOSCA-based cloud applications. In *Service-Oriented Computing*, volume 8274 of *Lecture Notes in Computer Science (LNCS)*, pages 692–695. Springer, 2013.

[7] T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann. TOSCA: Portable automated deployment and management of cloud applications. In Athman

Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Advanced Web Services*, pages 527–549. Springer, 2014.

[8] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Automated Discovery and Maintenance of Enterprise Topology Graphs. In *Proceedings of the 6th IEEE International Conference on Service Oriented Computing and Applications (SOCA'13)*, pages 126–134. IEEE, 2013.

[9] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. Migration of enterprise applications to the cloud. *Information Technology*, 56(3):106–111, May 2014.

[10] Tobias Binz, Christoph Fehling, Frank Leymann, Alexander Nowak, and David Schumm. Formalizing the Cloud through Enterprise Topology Graphs. In *Proceedings of the 5th International Conference on Cloud Computing (CLOUD'12)*, pages 742–749. IEEE, 2012.

[11] A. Bracciali, A. Brogi, and C. Canal. A formal approach to component adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.

[12] Uwe Breitenbücher, Tobias Binz, Oliver Kopp, and Frank Leymann. Vinothek - A Self-Service Portal for TOSCA. In *Proceedings of the 6th Central European Workshop on Services and their Composition (ZEUS'14)*, volume 1140 of *CEUR Workshop Proceedings*, pages 69–72. CEUR-WS.org, 2014.

[13] A. Brogi and J. Soldani. Matching cloud services with TOSCA. In *Advances in Service-Oriented and Cloud Computing*, volume 393 of *Communication in Computer and Information Science (CCIS)*, pages 218–232. Springer, 2013.

[14] Antonio Brogi and Jacopo Soldani. Reusing cloud-based services with TOSCA. In Erhard Plödereder, Lars Grunske, Eric Schneider, and Dominik Ull, editors, *44. Jahrestagung der Gesellschaft für Informatik, INFORMATIK 2014*, volume 232 of *LNI*, pages 235–246. GI, 2014.

[15] Antonio Brogi, Jacopo Soldani, and PengWei Wang. TOSCA in a nutshell: Promises and perspectives. In *Proceedings of the 3rd European Conference on Service-Oriented and Cloud Computing (ESOCC'14)*, 2014.

[16] William B. Frakes and Kyo Kang. Software reuse research: Status and future. *IEEE Transactions on Software Engineering*, 31(7):529–536, 2005.

[17] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *ACM Computing Surveys*, 23(3):319–344, 1991.

[18] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[19] J. Guillén, J. Miranda, J. M. Murillo, and C. Canal. A service-oriented framework for developing cross cloud migratable software. *Journal of Systems and Software*, 86(9):2294 – 2308, 2013.

[20] M. Hamdaqa, T. Livogiannis, and L. Tahvildari. A reference model for developing cloud applications. In *Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER'11)*. SciTePress, 2011.

[21] W. Kongdenfha, H. R. Motahari-Nezhad, B. Benatallah, F. Casati, and R. Saint-Paul. Mismatch patterns and adaptation aspects: A foundation for rapid development of web service adapters. *IEEE Transactions on Services Computing*, 2(2):94–107, 2009.

[22] Oliver Kopp, Tobias Binz, Uwe Breitenbücher, and Frank Leymann. Winery - A Modeling Tool for TOSCA-based Cloud Applications. In *Proceedings of the 11th International Conference on Service-Oriented Computing (ICSOC'13)*, volume 8274 of *Lecture Notes in Computer Science (LNCS)*, pages 700–704. Springer, 2013.

[23] B. Martino, D. Petcu, R. Cossu, P. Goncalves, T. Mahr, and M. Loichate. Building a mosaic of clouds. In *Euro-Par 2010 Parallel Processing Workshops*, volume 6586 of *Lecture Notes in Computer Science (LNCS)*, pages 571–578. Springer, 2011.

[24] 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.

[25] OASIS. TOSCA 1.0 (Topology and Orchestration Specification for Cloud Applications), Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`, 2013.

[26] V. Tran, J. Keung, A. Liu, and A. 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.

[27] Johannes Wettinger, Michael Behrendt, Tobias Binz, Uwe Breitenbücher, Gerd Breiter, Frank Leymann, Simon Moser, Isabell Schwertle, and Thomas Spatzier. Integrating Configuration Management with Model-Driven Cloud Management Based on TOSCA. In *Proceedings of the 3rd International Conference on Cloud Computing and Services Science (CLOSER'13)*. SciTePress, 2013.

[28] X. Xiong and Z. Weishi. The Current State of Software Component Adaptation. In *First International Conference on Semantics, Knowledge and Grid, 2005. SKG '05.*, pages 103–103, 2005.