Validating TOSCA Application Topologies

Antonio Brogi, Antonio Di Tommaso and Jacopo Soldani
Department of Computer Science, University of Pisa, Pisa, Italy

Keywords: TOSCA, Validation, Cloud Application Topologies.

Abstract: The Topology and Orchestration Specification for Cloud Applications (TOSCA) is a standardised metamodel that permits specifying cloud applications, and automating their deployment and management. TOSCA permits describing the structure of an application as a topology graph, which can then be exploited by TOSCA-compliant cloud platforms to automate the deployment of the components forming an application. In this paper we formalise the conditions that must hold for checking the validity of TOSCA application topologies, and we present an open-source prototype of validator (called SOMMELIER) that implements such conditions.

1 INTRODUCTION

Cloud computing is revolutionising IT by enabling on-demand network access to shared pools of configurable computing resources. Current cloud technologies however suffer from a lack of standardisation, with different providers offering similar resources in a different manner (Armbrust et al., 2010). Hence, how to deploy and flexibly manage complex composite applications over heterogeneous cloud platforms is one of the main challenges that have emerged after the cloud revolution (Brogi et al., 2014a).

OASIS recently released a YAML\(^1\) version (OASIS, 2016) of TOSCA, the Topology and Orchestration Specification for Cloud Applications. The TOSCA Simple Profile in YAML provides a YAML-based modelling language that permits specifying portable cloud applications, and to automate their deployment and management.

TOSCA permits describing the structure of a cloud application as a typed, directed topology graph. The nodes in a topology graph represent application components (e.g., a web-based frontend, a web server, a backend database, a database management system), and each of them can be also be associated with the requirements, capabilities, and management operations of the corresponding component. The arcs in a topology graph instead model relationships among application components (e.g., the frontend is deployed on the server and connected to the database, which is in turn hosted on the database management system) by associating the requirements of a node with capabilities featured by other nodes.

\(^1\)YAML Ain’t Markup Language (http://yaml.org).

TOSCA applications can be automatically deployed on TOSCA-compliant cloud platforms (Binz et al., 2013). The processing of a TOSCA application is declarative, and strictly depends on the inter-node relationships in its topology: Initially, (i) all nodes without requirements on other nodes are deployed. Then, (ii) the nodes whose requirements are actually satisfied (by capabilities offered by the nodes that have been deployed) are deployed, and their outgoing relationships are properly processed. Step (ii) is repeated until all the nodes in the application topology have been deployed (OASIS, 2013b).

Consider, for instance, the application topology in Fig. 1. The topology nodes are a frontend, a server, a database, and a database management systems, while its relationships specify that the frontend must be deployed on the server and connected to the database, and that the database must be hosted on the management system. Step (i) would result in first deploying the server and the management system. Step (ii) would then result in inserting the database among those hosted by the management system. Finally, step (ii) would be repeated to deploy the frontend on the server, and to set up the connection from the frontend to the database.

The above (toy) example clearly illustrates how the deployment of a TOSCA application strictly de-
pends on the inter-node relationships in its topology. This is even more evident if we consider that the processing of TOSCA nodes and relationships relies on the configuration information that they specify, and which is contextualised with constraints on how to interconnect them (OASIS, 2013b). It is hence important to ensure — at design time — that application topologies are valid, and in particular that the relationships interconnecting the nodes in such topologies have been properly specified.

In this perspective, the main contributions in this paper are twofold:

(i) We formalise the conditions that must hold to have valid TOSCA application topologies. More precisely, since TOSCA permits specifying constraints on how to interconnect nodes (e.g., which capabilities can satisfy a requirement, or which types of relationships can be used to interconnect them), we systematically map such constraints into conditions that must hold to ensure the validity of a TOSCA application topology.

(ii) We present SOMMELIER, which is a prototype of validator for TOSCA application topologies. SOMMELIER checks whether the topology of a TOSCA application satisfies all interconnection constraints, and, if this is not the case, it displays all violations.

Notice that SOMMELIER can be fruitfully exploited by TOSCA application developers to automatically validate the topologies of their applications at design time, as they currently have to do it manually.

It is also worth highlighting that SOMMELIER fully integrates with the OpenStack TOSCA parser (OpenStack, 2016). As both SOMMELIER and the OpenStack TOSCA parser are open-source, they can lay the foundations for an open-source toolset for supporting TOSCA application developers from the design time till the run time (Brogi et al., 2014b).

In this perspective, the output of SOMMELIER can be fruitfully exploited for improving the functionalities of TOSCA editors (e.g., the graphical editor in Alien4Cloud (Alien4Cloud, 2016)). SOMMELIER can indeed be useful for validating TOSCA application topologies while they are being developed.

The rest of this paper is organised as follows. In Sect. 2 we provide the necessary background on TOSCA, and we discuss related work. In Sect. 3 we formalise the conditions for validating TOSCA application topologies by systematically analysing the component interconnection constraints that can be expressed in TOSCA. In Sect. 4 we present SOMMELIER, and we show how it permits validating TOSCA application topologies. Finally, in Sect. 5 we draw some concluding remarks.

2 BACKGROUND AND RELATED WORK

2.1 Background: TOSCA

TOSCA (OASIS, 2016) is an OASIS standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. TOSCA provides a YAML-based, machine-readable modelling language that permits describing cloud applications. The obtained application specifications can then be processed to automate the deployment and management of the specified applications.

TOSCA permits specifying a cloud application as a service template, which is in turn composed by a topology template, and by the types needed to build such a topology template (Fig. 2). The topology template is a typed directed graph that describes the topological structure of the composite cloud application. Its nodes (called node templates) model the application components, while its edges (called relationship templates) model the relationships occurring among such components. The topology template may also contain typed policies, which permit specifying non-functional information about the node templates they target.

The node templates and relationship templates are typed by means of node types and relationship types, respectively. A node type defines the observable properties and attributes of a component, its possible requirements, the

![Figure 2: TOSCA service template (OASIS, 2016).](image-url)
capabilities it may offer to satisfy other components’ requirements, and the interfaces through which it offers its management operations. Capabilities are also typed (by means of capability_types), to permit specifying their properties, attributes, and valid_source_types (viz., the node types that can be satisfied by such capabilities).

A relationship type instead describes the observable properties and attributes of a relationship occurring between two application components, and the interfaces through which it offers its management operations. A relationship type can also put constraints on its valid_target_types (viz., the capability types that can be targeted by a relationship whose type is that under definition).

TOSCA type system supports inheritance. A node type can be defined by extending another, thus permitting the former to inherit the latter’s properties, attributes, requirements, interfaces, and operations. Analogously, a relationship type or a capability type can extend another to inherit all its features.

TOSCA also prescribes the format (called CSAR — Cloud Service ARchive) to archive application specifications along with the installable and executable files needed to properly instantiate the specified applications. This is because the modelling language illustrated above only allows developers to specify the application topology and its management and to give it in a .tosca document. Such document must be packaged together with the artefacts implementing its components so as to make all such artefacts available to the execution environment.\footnote{A more detailed, self-contained introduction to TOSCA can be found in (Brogi et al., 2014b).

2}

### 2.2 Related Work

Software systems are assuming a key role in everyday life, and guaranteeing that such systems satisfy their functional and non-functional requirements is becoming imperative and increasingly difficult (Marchetti, 2014). Validation techniques are hence crucial, as their purpose is precisely to evaluate software systems and to determine whether they satisfy specified requirements (Geraci, 1991).

The need for validation is also recognised by TOSCA (OASIS, 2016). Indeed, TOSCA permits specifying constraints that must be satisfied while interconnecting application components to form the topology of a cloud application. However, the current support for checking such constraints is limited, which makes the process of validating TOSCA applications cumbersome and time-consuming.

The OpenStack TOSCA Parser (OpenStack, 2016) can be used to check whether TOSCA application specifications are syntactically correct. More precisely, the OpenStack TOSCA Parser permits checking that all TOSCA elements have been specified in the proper section (e.g., node types in the node_types section, node templates in the node_templates section, etc.), and that node templates and relationship templates are specified coherently with the structure given by the corresponding types (e.g., the capabilities assigned in a node template are also defined in the corresponding node type, integer properties contain integer values, etc.). However, despite the OpenStack TOSCA Parser checks that the actual value assigned to a template’s field is of the proper type, it does not perform any check about the “meaningfulness” of such value. For instance, it checks whether the node field of a requirement assignment contains the name of a node template, but it does not check whether such node template satisfies the interconnection constraints specified in the corresponding requirement definition. Our objective is instead to permit checking such a kind of constraints.

Similar considerations apply to other TOSCA parsers currently available, e.g., the brooklyn-tosca parser (CloudSoft, 2016), or the TOSCA parser in Alien4Cloud (Alien4Cloud, 2016).

TOSCA is also available in an XML-based version (OASIS, 2013a), which still permits specifying the topology of a composite cloud application, as well as the constraints to interconnect the components forming such topology. TOSCA XML also permits specifying management plans, in the form of workflows orchestrating the management operations offered by the application components.

TOSCA XML applications can be edited with the Winery graphical environment (Kopp et al., 2013), and executed with the OpenTOSCA engine (Binz et al., 2013). However, despite Winery and OpenTOSCA are equipped with parsers checking whether the XML sources of applications are syntactically correct, there is no support for validating the interconnections forming application topologies.

(Hirmer et al., 2014) propose an approach for automatically completing TOSCA XML application topologies. The approach is based on some of the interconnection constraints that can be expressed in TOSCA XML, which are exploited to decide which available components can satisfy a dangling requirement of a component. The approach of (Hirmer et al., 2014) is a first approach exploiting the interconnection constraints that can be expressed in TOSCA XML. The aim of (Hirmer et al., 2014) is however different from ours, as they rely on TOSCA XML,
and since they consider only some of the interconnection constraints that can be expressed in TOSCA XML (viz., those constraining the relationship types and capability types that can be validly used to fulfill a requirement). Our aim is instead to systematically map all constraints that can be expressed in TOSCA to formal conditions, hence enabling a full validation of TOSCA application topologies.

On the other hand, it is possible to validate management plans in TOSCA XML. Management protocols (Brogi et al., 2015; Brogi et al., 2016) permit specifying the management behaviour of the components forming an application, which can then be combined (according to the application topology) to obtain the management behaviour of the whole application. Such a behaviour permits automating various useful analyses (e.g., determining the validity and effects of a management plan, or which plans reach a given goal). However, despite topologies are crucial to determine the management behaviour of applications, they are assumed to be valid, because of the lack of support for validating application topologies.

In summary, despite TOSCA is designed to permit specifying constraints on how to interconnect application components to form application topologies, the support for checking such constraints is currently lacking. Existing parsers can be used to check whether TOSCA applications are syntactically correct, but currently there are no tools that permit validating TOSCA application topologies. Our objective is to address this lack, by systematically mapping the interconnection constraints that can be expressed in TOSCA to formal conditions, and by exploiting such conditions to devise a first support for validating TOSCA application topologies.

Finally, it is worth mentioning that our approach shares its baselines with most of existing approaches for checking the validity of component-based systems at design time, e.g., (Speck et al., 2002; Caporuscio et al., 2004; Wu et al., 2003), or for automatically obtaining valid component-based systems, e.g., (Autili et al., 2007; Pelliccione et al., 2008). Given a specification of a component-based system (a TOSCA application specification, in our case), we try to enforce that all interconnections between components are properly settled (which in our case means to check whether the topology of an application satisfies all the specified interconnection constraints). The main difference between such approaches and ours is given by the context: (Speck et al., 2002; Caporuscio et al., 2004; Wu et al., 2003; Autili et al., 2007; Pelliccione et al., 2008) focus on systems that have to offer a certain functionality by validly composing the functionalities offered by their components. We instead focus on ensuring that the dependencies between the components forming a TOSCA application are properly specified, as such information is the basis for orchestrating the management of a TOSCA application.

3 VALIDATING TOPOLOGIES

The objective of this paper is to permit validating the topology of a TOSCA application (viz., its topology_template). As shown in Sect. 2.1, the topology_template is a typed directed graph whose nodes model the application components, and whose edges model the relationships occurring between such components. Each relationship specifies that a requirement of the source node must be actually satisfied by (a capability of) the target node.

As we anticipated in Sect. 1, in order to validate the topology of a TOSCA application, we must check whether all inter-component relationships have been properly specified. For each relationship, we must validate all the TOSCA elements concerning its source, the relationship itself, and its target. In this respect, please note that the source of a relationship is always a requirement of a node, that a relationship is actually a relationship template, and that the target of a relationship is either a node or a capability of a node (OASIS, 2016).

We hereafter systematically map the interconnection constraints expressed while defining TOSCA requirements, relationships, and capabilities to formal conditions. We separately discuss the conditions to validate the interconnection constraints specified in the sources of relationships (Sect. 3.1), in relationships (Sect. 3.2), and in the targets of relationships (Sect. 3.3). We then combine all such conditions to define the notion of validity for a TOSCA application topology (Sect. 3.4).

3.1 Validating Sources of Relationships

We hereby illustrate how to validate the source of each relationship appearing in an application topology. As the source of a relationship is always a node’s requirement (OASIS, 2016), we must check that each relationship outgoing from a requirement is compatible with the requirement definition itself.

After systematically reading the whole TOSCA specification (OASIS, 2016), we have extracted the portions that permit specifying interconnection constraints. We hereafter recall such portions, and we show how to directly map them to formal conditions that must hold to have valid TOSCA application topologies.
In the following, we recall how to declare a requirement in TOSCA, and what is its meaning according to the TOSCA specification. We then single out the checks that must be performed to ensure that no relationship is violating the constraints given by a requirement declaration.

### How to define a Requirement in TOSCA

We hereby recall how to define a requirement in a TOSCA node type, and how to instantiate a requirement in a TOSCA node template.

A node type $n_{\text{temp}}$ defines the set of named requirements that can be exposed by a node template of such type. Each requirement definition can be given within the requirements section of $n_{\text{temp}}$ according to the (a) simple or (b) extended grammars in Figure 3. In both cases, a developer can define the requirement name ($req\_name$), and she can provide constraints for its satisfaction.

The (a) simple grammar requires to indicate the name ($cap\_type\_name$) of a valid capability type that can fulfill the requirement under definition. The (b) extended grammar also permit indicating three optional constraints:

- **node** permits indicating the name ($node\_type\_name$) of a valid node type that contains a capability definition that can be used to fulfill the requirement under definition.
- **relationship** permits providing the name of a relationship type that can be validly used to build a relationship template when fulfilling the requirement.
- **occurrences** permits instructing the minimum ($min\_occ$) and maximum ($max\_occ$) occurrences of the requirement under definition in a node template of the corresponding node type.$^4$

$^4$Since the focus of this paper is on validating inter-component dependencies in TOSCA application topologies, and since occurrences is not giving any constraint concerning inter-component dependencies, we shall not for-
cloud provider to select a type-compatible relationship template to relate the source and target nodes at run-time).

- **node_filter** permits defining additional constraints that TOSCA-compliant cloud providers will use to select a type-compatible target node that can fulfill the requirement under assignment at run-time.

### 3.1.2 How to validate a TOSCA Requirement

We hereby single out the conditions that must hold to validate sources of relationships at design-time⁶. In doing so, we exploit the following shorthand notation.

**Notation.** *In the following, we write:*

- `name(•)` and `type(•)` to denote the name and type associated to a TOSCA element, respectively.
- `t' ≥ t` to denote that `t'` extends `t` or is equal to `t`.
- `e.f` to denote the field `f` of the TOSCA element `e` (which is ⊥ if `f` is not defined in `e`), and
- `R(•)` and `C(•)` to denote the requirements and capabilities assigned by a node template or defined in a node type.

Given a TOSCA application topology, the sources of its inter-component relationships are valid if each of its node templates satisfies all the aforementioned conditions.

**Condition 1.** Let `ntempl` be a node template of type `n_type`. Then:

\[
\forall r_a \in R(ntempl), \exists r_d \in R(n_type):
\]

1. `name(r_a) = name(r_d) \land
2. `r_d.node \neq \perp \Rightarrow type(r_a.node) \geq r_d.node \land
3. `r_a.capability \neq \perp \Rightarrow type(r_a.capability) \geq r_d.capability \land
4. `r_a.capability = \perp \Rightarrow \exists c \in C(r_a.node): type(c) \geq r_d.capability \land
5. `r_d.relationship \neq \perp \land r_a.relationship \neq \perp \Rightarrow type(r_a.relationship) \geq r_d.relationship

**3.2 Validating Relationships**

We hereby illustrate how to validate the relationships occurring between the nodes appearing in an application topology. As such relationships are expressed by typed relationship templates (OASIS, 2016), we must check that each relationship template is instantiated without violating any of the constraints given in the corresponding relationship type.

In the following, we recall how to declare a relationship type in TOSCA, and which is its meaning according to the TOSCA specification. We then illustrate the conditions that must hold to ensure that no relationship template is violating the constraints given by the corresponding relationship type.

---

⁶We hence exclude all constraints that permit defining how to automatically complete the topology of a TOSCA application (by selecting type-compatible node templates and relationship templates to satisfy pending requirements — e.g., `node: node_type_name, relationship: rel_type_name, and capability: cap_type_name` in Figure 4(b)). Our validation approach can however be useful for double-checking automatically completed TOSCA application topologies (as well as for helping the automatic completion itself).

⁷Given that `t` and `t'` are TOSCA types, `t'` extends `t` if `t'` is (directly or indirectly) derived_from `t`. Firstly, we must check that, for each requirement assigned `r_a` in a node template `ntempl` (of type `n_type`), there exists a corresponding requirement definition `r_d` in `n_type` (Condition 1..1).

We can then check that no requirement assignment `r_a` is violating the constraints indicated by the corresponding requirement definition `r_d`:

- The (optional) field `node` of a requirement definition `r_d` permits indicating the name of a valid node type that can be used to fulfill a requirement. Hence, if `node` is specified in `r_d`, the type of the node template targeted by each corresponding requirement assignment `r_a` has to extend to or be equal to that indicated in `node` (Condition 1..2.)

- A requirement definition `r_d` indicates (either inline or with the keyword `capability`) the name of a valid capability type that can fulfill the requirement. This is ensured if each corresponding requirement assignment `r_a` directly targets a capability whose type extends or is equal to that indicated in the requirement definition (Condition 1..3). Alternatively, if a requirement assignment `r_a` is only indicating the target node template (but not pointing to any of its capabilities), we must check whether such node template is offering at least one type-compatible capability (Condition 1..4).

- Finally, the (optional) field `relationship` of a requirement definition `r_d` permits indicating the name of a relationship type that can be validly used to build a relationship template when fulfilling the requirement. Hence, if `relationship` is specified in `r_d`, the type of a relationship template outgoing from a corresponding requirement assignment `r_a` (if any) extends or is equal to that indicated in `relationship` (Condition 1..5).
3.2.1 How to define a Relationship in TOSCA

A relationship template instantiate a relationship type to specify the actual occurrence of a manageable relationship between two node templates in an application topology. A relationship type can be defined with the grammar in Figure 5, which permits specifying the name (rel_type.name) of a relationship type, as well as the features and constraints characterising such relationship type.

```
rel_type.name: derived_from: parent.rel_type.name
version: version.number
description: rel.description
properties: property_definitions
attributes: attribute_definitions
interfaces: interface_definitions
valid_target_types: [ cap.type_names ]
```

Figure 5: Grammar for relationship types (OASIS, 2016).

- The (optional) field derived_from permits indicating the name (parent.rel_type.name) of the parent relationship type. The relationship type under definition inherits all the features the parent relationship type, and it can override some of them (OASIS, 2013b).
- For instance, if the relationship type under definition does not specify a new list of valid_target_types, then it takes that of the parent relationship type. Otherwise, the parent’s list of valid_target_types is overridden by that specified in the relationship type under definition.
- version and description are optional fields that permit versioning and describing (in natural language) a relationship type.
- The (optional) fields properties and attributes permit describing the structure of the desired and actual state of a relationship, respectively. Concrete state values will be then given in the relationship templates instantiating the relationship type under definition.
- interfaces is a (optional) field that permits declaring the management operations that can be offered by relationship templates whose relationship type is that under definition.
- valid_target_types is a (optional) list of one or more names (cap_type.names) of capability types that are valid targets for the relationship type under definition.

8 All TOSCA relationship types should be derived (directly or indirectly) from the tosca.relationships.Root relationship type (OASIS, 2016).

3.2.2 How to validate a TOSCA Relationship

Since a relationship template instantiates a relationship type in an application topology, it has to not violate the constraints given when defining the corresponding relationship type.

Notice that the only constraints that can be given while defining a relationship type are those concerning the valid_target_types. The latter can either be specified inline or inherited from the parent type.

**Notation.** In the following, we write $T(\cdot)$ to denote the set of capability types that are valid targets for a relationship type. Given a relationship type $reltype$:

- If $reltype.valid_target_types \neq \bot$, then $T(reltype)$ is the set containing all elements in $reltype.valid_target_types$,
- otherwise, if $reltype.derived_from \neq \bot$, then $T(reltype) = T(reltype.derived_from)$,
- otherwise, $T(reltype)$ is the set containing all capability types (meaning that all capability types are valid targets for $reltype$).

Given a TOSCA application topology, its relationship templates are valid if they satisfy the following condition.

**Condition 2.** Let $reltemp$ be a relationship template of type $reltype$. If there exists a requirement assignment $ra$ that is source of $reltemp$, then:

1) $ra.capability \neq \bot \Rightarrow \exists ctype \in T(reltype): type(ra.capability) \geq ctype \wedge$
2) $ra.capability = \bot \Rightarrow \exists cnode \in C(ra.node):$
   $\exists ctype \in T(reltype): type(cnode) \geq ctype$

Namely, if a relationship template $reltemp$ is targeting a specific capability, then the type of such capability has to be compatible with at least one of those indicated in valid_target_types (Condition 2.1).

If the source of a relationship template $reltemp$ is instead only indicating the target node template (but not pointing to any of its capabilities), we must check whether such node template is offering at least one type-compatible capability (Condition 2.2).

3.3 Validating Targets of Relationships

While the source of a relationship is always a requirement, the target of a relationship can be either a capability or a node template. In the latter case, the node
template must offer at least a capability capable of satisfying the source requirement (OASIS, 2016).

In the following we recall how to specify a capability in TOSCA, and which is its meaning according to the TOSCA specification. We then illustrate the conditions that must hold to ensure that no relationship is violating the constraints given by its target.

### 3.3.1 How to define a Capability in TOSCA

A capability type is a reusable entity that describes a kind of capability that a node type can declare to expose (OASIS, 2016). A capability type can be defined according to the grammar in Figure 6, which permits specifying the name (`cap_type_name`) of a capability type, as well as with the features and constraints characterising such capability type.

```plaintext
| cap_type_name: |
| derived_from: parent_cap_type_name |
| version: version_number |
| description: capability_description |
| properties: property_definitions |
| attributes: attribute_definitions |
| valid_source_types: [ node_type_names ] |
```

Figure 6: Grammar for capability types (OASIS, 2016).

- The (optional) field `derived_from` permits indicating the name (`parent_cap_type_name`) of the parent capability type. The capability type under definition inherits all the features the parent capability type, and it can override some of them (OASIS, 2013b).

For instance, if the capability type under definition does not specify a new list of `valid_source_types`, then it takes that of the parent capability type. Otherwise, the parent’s list of `valid_source_types` is overridden by that specified in the capability type under definition.

- `version` and `description` are optional fields that permit versioning and describing (in natural language) a capability type.

- The (optional) fields `properties` and `attributes` permit describing the structure of the desired and actual state of a capability, respectively. Concrete state values will be then given by the node templates instantiating capabilities whose type is that under definition.

- `valid_source_types` is a (optional) list of one or more names (`node_type_names`) of node types that are valid sources for relationships whose target capability is of the type under definition.

Capability types are then referred by node types to define the set of named capabilities that can be exposed by a node template of such type. Each capability definition can be given within the `capabilities` section of a node type (Figure 7), according to the (a) simple or (b) extended grammars in Figure 7. In both cases, a developer can define the capability name (`cap_name`), and she can specify the type (`cap_type_name`) of the capability under definition.

```plaintext
| cap_name: cap_type_name |
| type: cap_type_name |
| description: capability_description |
| properties: property_definitions |
| attributes: attribute_definitions |
| valid_source_types: [ node_type_names ] |
```

(a) Figure 7: (a) Simple and (b) extended grammars for capability definitions (OASIS, 2016).

```plaintext
| cap_name: |
| properties: property_assignments |
| attributes: attribute_assignments |
```

(b) Figure 8: Grammar for capability assignments (OASIS, 2016).

A node template `n_temp` instantiates a node type `n_type` in a TOSCA application topology, as well as all the capabilities it defines. A node template can also assign concrete values to the properties and attributes of such capabilities, to provide additional information concerning their desired and actual state, respectively. Such assignments can be given through so-called capability assignments within the capabilities section of `n_temp` (Figure 8).

### 3.3.2 How to validate a TOSCA Capability

While defining a capability in TOSCA, the only constraints that can be given are those concerning the `valid_source_types`. Such constraints can be specified within the specification of a capability type, or within the capability definitions of a node type. The constraints given while defining a capability type can...
Table 1: Conditions to validate inter-component relationships in TOSCA application topologies.

| Validating sources of relationships (Condition 1.) | Let $n_{temp}$ be a node template of type $n_{type}$. Then, $\forall r_a \in R(n_{temp}), \exists r_d \in R(n_{type})$:
|--------------------------------------------------|---------------------------------------------------------------|
| (1.1) name$(r_a) = \text{name}(r_d) \land$       | (1.2) $r_d$ node $\not\equiv \exists \rightarrow \text{type}(r_d \text{node}) \geq r_d$ node $\land$
| (1.3) $r_a$ capability $\not\equiv \exists \rightarrow \text{type}(r_a \text{capability}) \geq r_d$ capability $\land$
| (1.4) $r_d$ capability $\not\equiv \exists \rightarrow \exists c \in C(r_a \text{node}), \text{type}(c) \geq r_d$ capability $\land$
| (1.5) $r_d$ relationship $\not\equiv \exists \rightarrow r_d$ relationship $\not\equiv \exists \rightarrow \text{type}(r_d \text{relationship}) \geq r_d$ relationship |

| Validating relationships (Condition 2.) | Let $r_{temp}$ be a relationship template, and let $r_{type}$ be its relationship type. If there exists a requirement assignment $r_a$ that is source of $r_{temp}$, then:
|--------------------------------------------------|---------------------------------------------------------------|
| (2.1) $r_a$ capability $\not\equiv \exists c_{type} \in T(r_{type}) : \text{type}(r_a \text{capability}) \geq c_{type} \land$
| (2.2) $r_a$ capability $\not\equiv \exists c_{type} \in C(r_a \text{node}), c_{type} \in T(r_{type}) : \text{type}(c_{type}) \geq c_{type}$ |

| Validating targets of relationships (Condition 3.) | Let $c_a$ be a capability assignment, whose corresponding type and definition are $c_{type}$ and $c_d$. For each node template $n_{temp}$ having a requirement assignment $r_a$ such that $r_a$ capability $\equiv c_{a}$:
|--------------------------------------------------|---------------------------------------------------------------|
| (3.1) $\exists n_{type} \in S(c_{type}) : \text{type}(n_{temp}) \geq n_{type} \land$
| (3.2) $\exists n_{type} \in S(c_{d}) : \text{type}(n_{temp}) \geq n_{type}$ |

either be specified inline or they can be inherited from the parent capability type.

Notation. In the following, we write $S(\cdot)$ to denote the set of node types that are valid sources for a capability type. Given a capability type $c_{type}$:

- If $c_{type}$ valid source types $\equiv \perp$, then $S(c_{type})$ is the set containing all elements in $c_{type}$ valid source types,
- otherwise, if $c_{type}$ derived from $\equiv \perp$, then $S(c_{type}) = S(c_{type}$ derived from),
- otherwise, $S(c_{type})$ is the set containing all node types (meaning that all node types are valid targets for $c_{type}$).

Given a capability assignment $c_{a}$, in a TOSCA application topology there might be multiple node templates (having requirement assignments) that are sources of relationship templates targeting $c_{a}$. All such node templates must not violate any of the constraints imposed by $c_{a}$:

- Neither those indicated by its capability type $c_{type}$ (Condition 3.1),
- nor those specified within the corresponding capability definition $c_{d}$ (Condition 3.2).

Condition 3. Let $c_a$ be a capability assignment, whose corresponding type and definition are $c_{type}$ and $c_{d}$. For each node template $n_{temp}$ having a requirement assignment $r_a$ such that $r_a$ capability $\equiv c_{a}$:

1) $\exists n_{type} \in S(c_{type}) : \text{type}(n_{temp}) \geq n_{type} \land$
2) $\exists n_{type} \in S(c_{d}) : \text{type}(n_{temp}) \geq n_{type}$

3.4 Valid TOSCA Application Topologies

In Sects. 3.1, 3.2, and 3.3 we have illustrated the conditions that permit validating sources, instances, and targets of TOSCA relationships, respectively. The following definition gathers all such conditions to permit validating all inter-component relationships appearing in a TOSCA application topology.

Definition 1. A TOSCA application topology is valid if all its node templates and relationship templates satisfy Conditions 1., 2., and 3.

For the convenience of readers, Conditions 1., 2., and 3. are recapped in Table 1.

![Open-source toolchain for validating TOSCA application topologies.](image-url)
4 THE SOMMELIER PROTOTYPE

We hereby illustrate SOMMELIER, an open-source prototype for validating TOSCA application topologies (as we discussed in the previous section). The prototype of SOMMELIER is implemented in Python, and it is fully integrated with the OpenStack TOSCA parser (OpenStack, 2016).

Figure 9 illustrates how SOMMELIER performs the validation of a TOSCA application topology:

1 SOMMELIER inputs a CSAR package or a .tosca document containing the TOSCA application topology to be validated.

As the current prototype of SOMMELIER is a command-line tool, the CSAR or .tosca files can be passed to SOMMELIER with a dedicated command line option.

2 The input is forwarded by SOMMELIER to the OpenStack Parser.

3 The OpenStack Parser checks whether the TOSCA application is syntactically correct, i.e., it checks whether all TOSCA elements have been specified in the proper section (e.g., node types in the node_types section) and that each template respects the structure given by the corresponding type (e.g., the capabilities assigned in a node template are also defined in the corresponding node type). If this is the case, the OpenStack Parser generates a representation of the TOSCA application in Python, according to its object-model (OpenStack, 2016).

4 The Python objects representing the TOSCA application are returned to SOMMELIER.

5 SOMMELIER validates the topology of the TOSCA application by checking whether all (the Python objects representing) its nodes and relationships satisfy all conditions whose abstract structure is illustrated in Figure 10. Namely, SOMMELIER associates each requirement of each node template with a list containing all validation errors affecting the relationship outgoing from such requirement. Each error is in turn represented as a list, whose first element is a numeric code (e.g., 1.2 if the requirement is violating Condition 1..2), and whose remaining elements may be used to provide further information about the error (e.g., the type and name of the target node, which is not satisfying Condition 1..2).

6 The results of the validation are displayed by SOMMELIER. The current prototype of SOMMELIER either states that the service template’s topology is valid, or it lists all the detected validation errors (by also explaining why such errors occurred).

Example. We now illustrate how to use SOMMELIER to validate existing TOSCA application topologies. As SOMMELIER is a command-line tool, running an instance of SOMMELIER just requires to type the following command in a command shell:

```bash
$ python sommelier.py --template-file=../tosca-parser/toscaparser/tests/data/topology_template/tosca_elk.yaml
```

The application topology is valid.

(a)

```bash
$ python sommelier.py --template-file=../tosca-parser/toscaparser/tests/data/topology_template/transactionsystems.yaml
```

node_template1_name: {
  req1_name: [
    [ error1_code, error1_info ],
    [ error2_code, error2_info ],
    ...
  ],
  req2_name: [ ... ],
  ...
}

node_template2_name: { ... }

(b)

Figure 9: Illustrates how SOMMELIER performs validation of a TOSCA application topology.

Figure 10: Structure of the Python dictionary containing the results of the validation.

Figure 11: Example of runs of SOMMELIER.
$ python sommelier.py
    --template-file=template-file-path
(where template-file-path is the absolute path
where the TOSCA file to be validated is stored).

Concrete examples of run of SOMMELIER are
shown in Figure 11.

In the figure, the application specification that
are passed to SOMMELIER are taken among those
available in the GitHub repository of the OpenStack
TOSCA parser. Figure 11.(a) shows that the TOSCA
application topology specified in tosca_elk.yaml\textsuperscript{14} is
valid.

Figure 11.(b) instead shows that the TOSCA
application topology specified in transactionssubsys-
tem.yaml\textsuperscript{15} is not valid, because the relationship out-
going from requirement host of the node template app
is violating Condition 1..2 (since it targets the node
template websrv, whose type is not compatible with
those indicated in the requirement definition corre-
sponding to host). Such a kind of inconsistencies
would be really hard to be manually detected, with-
out the availability of a tool like SOMMELIER.

5 CONCLUDING REMARKS

TOSCA provides a standardised language that per-
mits specifying portable cloud applications, and to au-
tomate their deployment and management. As the au-
tomated deployment of TOSCA applications is based
on their topologies (OASIS, 2013b), it is fundamental
to ensure that such topologies are valid.

TOSCA permits specifying the constraints that
must be satisfied while interconnecting application
components to form the topology of a cloud applica-
tion. In this paper:

(i) We have systematically mapped such con-
straints to formal conditions that must hold to
ensure the validity of a TOSCA application
topology. The obtained conditions are recapped
in Table 1.

(ii) We have also presented SOMMELIER, an open-
source prototype of validator for TOSCA appli-
cation topologies.

SOMMELIER implements all the aforementioned val-
idation conditions (as one can readily check by run-
ning the unit tests available in its repository). SOM-
MELIER can hence be fruitfully exploited by TOSCA
application developers to automatically validate the
topologies of their applications.

SOMMELIER is fully integrated with the Open-
Stack TOSCA parser (OpenStack, 2016). As they are
both open-source, they can lay the foundations for an
open-source toolset for supporting TOSCA applica-
tion developers. This is in line with the research direc-
tions indicated in (Brogi et al., 2014b), and in partic-
ular with the development of tools to support TOSCA
developers from the design time till the run time.

It is worth noting that SOMMELIER can be
fruitfully exploited for improving the functionalities
of existing graphical editors (such as that in the
Alien4Cloud platform (Alien4Cloud, 2016), for in-
stance). SOMMELIER can indeed be useful for vali-
dating TOSCA application topologies while they are
being developed. Its output could also be exploited
for suggesting how to fix errors (e.g., which capabil-
ity can actually satisfy a misplaced requirement) or to
drive the development itself (e.g., warning the user if
she is connecting a requirement to a wrong capability
—or impeding her to do so). As part of our future
work, we plan to integrate SOMMELIER with a graph-
ical editor, with the long-term objective of providing
a full-fledged support for the development of TOSCA
applications (Brogi et al., 2014b).

It is also worth highlighting that the elements that
can be expressed in TOSCA YAML (OASIS, 2016)
are a subset of those that can be defined in TOSCA
XML (OASIS, 2013a), as the former is a simplified
profile of the latter. We plan to investigate whether
the validation conditions contained in this paper are
enough to validate also the topologies of TOSCA
XML applications, and to (extend and) adapt them to
work with TOSCA XML, if necessary. This would
permit providing the OpenTOSCA open-source envi-
nronment (Kopp et al., 2013; Binz et al., 2013) also
with validation capabilities. It would also permit re-
fining the automated topology completion approach
by Hirmer et al. (Hirmer et al., 2014), by ensuring the
validity of obtained topologies.

Finally, as TOSCA permits describing not only the
 technological requirements of application com-
ponents, but also their non-functional requirements
(e.g., scalability constraints, in terms of minimum and
maximum amount of occurrences of a node replica,
or QoS requirements specified in policies applied to

\textsuperscript{14}The tosca_elk.yaml file has been de-
veloped within the OpenStack TOSCA parser
community, and it is publicly available at
https://raw.githubusercontent.com/openstack/tosca-
parser/master/toscaparser/tests/data/tosca_elk.yaml (last
accessed on November 13th, 2016).

\textsuperscript{15}The transactionssubsystem.yaml file has
been developed within the OpenStack TOSCA
parser community, and it is publicly available at
https://raw.githubusercontent.com/openstack/tosca-
parser/master/toscaparser/tests/data/topology_template/
transactionssubsystem.yaml (last accessed on November
13th, 2016).
nodes), TOSCA applications should also be validated also from a non-functional perspective. We plan to systematically map the non-functional constraints of that can be expressed in TOSCA to formal conditions that will permit carrying out such a validation.

ACKNOWLEDGEMENTS

Work partly supported by the project Through the fog (PRA_2016_64) funded by the University of Pisa.

REFERENCES


