TOSCA in a Nutshell: Promises and Perspectives*

Antonio Brogi, Jacopo Soldani, and PengWei Wang

Department of Computer Science, University of Pisa, Italy

Abstract. How to deploy and flexibly manage complex multi-service applications in the cloud is one of the emerging problems in the cloud era. The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) [1] aims at contributing to solve this problem by providing a language to describe and manage complex cloud applications in a portable, vendor-agnostic way. The objective of this paper is twofold: To provide a compact and easy-to-access introduction to TOSCA, and to discuss possible research directions for TOSCA.

1 Introduction

Cloud computing is revolutionizing IT by enabling a convenient, on-demand and scalable network access to shared pools of configurable computing resources. However, current cloud technologies suffer from a lack of standardization, with different providers offering similar resources in a different manner [2]. As a result, cloud developers tend to remain locked in a specific platform environment because it is practically unfeasible for them, due to high complexity and cost, to migrate their applications to a different platform. According to [3], in order to enable the creation of portable cloud applications, the application's components, their relations and management should be modeled in a standardized, machine-readable format. This will also allow the automation of the deployment and management of the modeled application [4].

In this perspective, OASIS recently released version 1.0 of TOSCA, the Topology and Orchestration Specification for Cloud Applications [1]. TOSCA proposes an XML-based modeling language which permits to specify an application's structure as a typed topology graph, and the management tasks as plans. More precisely, TOSCA aims at addressing the following three issues in cloud application management [3]: (O1) automated application deployment and management, (O2) portability of application descriptions and their management, and (O3) interoperability and reusability of components.

Interested readers can browse various documents to get acquainted with TOSCA. The official specification [1] and the primer [5] provide a comprehensive presentation of TOSCA, while several research papers (like [3], [6], and [7]) provide a short recap of the main features of TOSCA. Moreover, recent research

* Work partly supported by project EU-FP7-ICT-610531 SeaClouds (www.seaclouds-project.eu).

M. Villari et al. (Eds.) : ESOCC 2014, LNCS 8745, pp. 171–186, 2014.

papers (e.g., [8], [9], [10], [11], [12], [13], and [14]) are proposing various extensions of TOSCA. One of the motivations of this paper is that we believe that the availability of an updated, compact, and easy-to-access description of TOSCA may contribute to the dissemination of this OASIS specification.

In this paper:

(i) We try to provide a compact, easy-to-access description of TOSCA. We reorganize the available information about TOSCA in a compressed overview which outlines the goals of the specification, illustrates the TOSCA modeling language, positions TOSCA with respect to other cloud interoperability standard proposals and describes how TOSCA specifications are processed.

(ii) We analyze TOSCA with the aim of discussing some research perspectives which are leveraged by TOSCA itself. Namely, we discuss (D1) reuse of available specifications, (D2) enhanced and full-fledged implementations of so-called *TOSCA containers*, (D3) implementation of TOSCA tools, (D4) integration of TOSCA with existing standard proposals, and (D5) comparative assessment of TOSCA.

The rest of the paper is organized as follows. Sect. 2 presents an easy-to-access description of TOSCA. Sect. 3 analyzes TOSCA with the aim of highlighting its possible extensions and improvements, while Sect. 4 discusses some research perspectives. Finally, Sects. 5 and 6 discuss related work and draw some concluding remarks, respectively.

2 Overview of TOSCA

As previously mentioned, TOSCA [1] is an emerging standard whose main goal is to enable the creation of portable cloud applications and the automation of their deployment and management. In order to achieve this goal, TOSCA focuses on the following three sub-goals [3].

(O1) Automated Application Deployment and Management. TOSCA aims at providing a language to express how to automatically deploy and manage complex cloud applications.

This objective is achieved by requiring developers to define an abstract topology of a complex application and to create plans describing its deployment and management [4], [3] (see Sect. 2.1).

(O2) Portability of Application Descriptions and Their Management. TOSCA aims at addressing the portability of application descriptions and their management (but not the actual portability of the applications themselves) [3].

To this end, TOSCA provides a standardized way to describe the topology of multi-component applications (see Sect. 2.1). It also addresses management portability by relying on the portability of workflow languages used to describe deployment and management plans [6].

(O3) Interoperability and Reusability of Components. TOSCA aims at describing the components of complex cloud applications in an interoperable and reusable way.

Interoperability is the capability for multiple components "*to interact using well-defined messages and protocols*" [1] so that they can be combined independently of the vendor(s) supplying them. TOSCA abstracts from messages and protocols details, and it permits to describe the dependencies between application components (see Sect. 2.1).

Furthermore, TOSCA enables defining, assembling, and packaging the building blocks of an application in a completely self-contained manner (see Sect. 2.2), thus providing a standardized way to reuse them in different applications [3].

Fig. 1 tries to position TOSCA with respect to some other cloud interoperability[1] standards and specifications, namely CAMP [16], CIMI [17], EMML [18], OCCI [19], Open-CSA [20], OVF [21], SOA-ML [22], and USDL [23]. The three numbered sections of the pie represent the aforementioned three main goals of TOSCA, and the position of each label is intended to summarize "how much" the goals of an initiative overlap with TOSCA goals[2]. More precisely, to indicate that a standard is targeting one of the goals, its label covers the corresponding section of the pie. For instance, CAMP aims at addressing both O2 and O3. Furthermore, if a label is not completely contained in the pie, this means that the corresponding standard only partially addresses the covered goals. Consider for instance OCCI. It provides an standardized IaaS interface which can be employed to automatize application deployment and management. Nevertheless, automation is not its real goal and thus OCCI is represented as partially covering the section O1 and partially out of the pie.

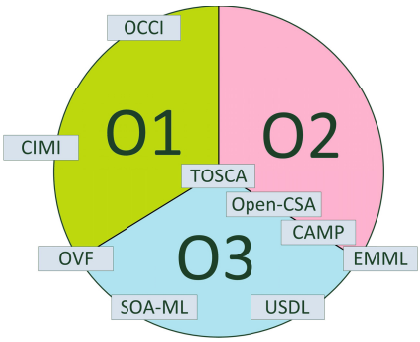

Fig. 1. Positioning TOSCA

2.1 TOSCA Modeling Language

To achieve the aforementioned goals, TOSCA provides an XML-based modeling language, whose purpose is to allow formalizing the structure of each cloud application as a typed topology graph, and the management tasks as plans [3].

An application is represented as a `ServiceTemplate` (Fig. 2), which is in turn composed by a `TopologyTemplate` and (optionally) by some management `Plans` [1].

[1] A more thorough discussion on the relations between TOSCA and other cloud interoperability initiatives can be found in [15].

[2] Note that all mentioned initiatives target cloud interoperability, while only some of them also target the interoperability of application *components* (viz., O3).

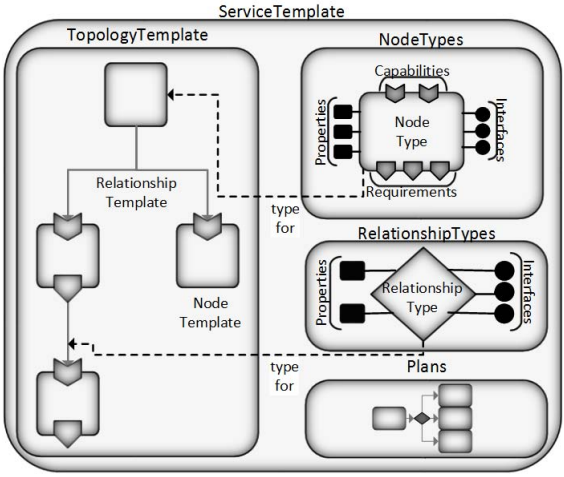

Fig. 2. TOSCA `ServiceTemplate`

Generic type and type implementation definitions (which will be discussed later) are also contained in the XML document defining the `ServiceTemplate` as they are referred to by the templates appearing in the topology [5].

In the following we illustrate the TOSCA modeling language with reference to the `SugarCRM` application example (whose complete description can be found in the TOSCA primer [5]), which exemplifies a complex cloud application designed for enabling businesses to manage the relationships with their customer.

Topology of an Application. The topology of a multi-component application is represented by means of `TopologyTemplate`s. A `TopologyTemplate` is essentially a typed graph whose nodes are the application components, and whose edges are the relations between these application components [1]. Syntactically speaking, the application components and their relations are represented by

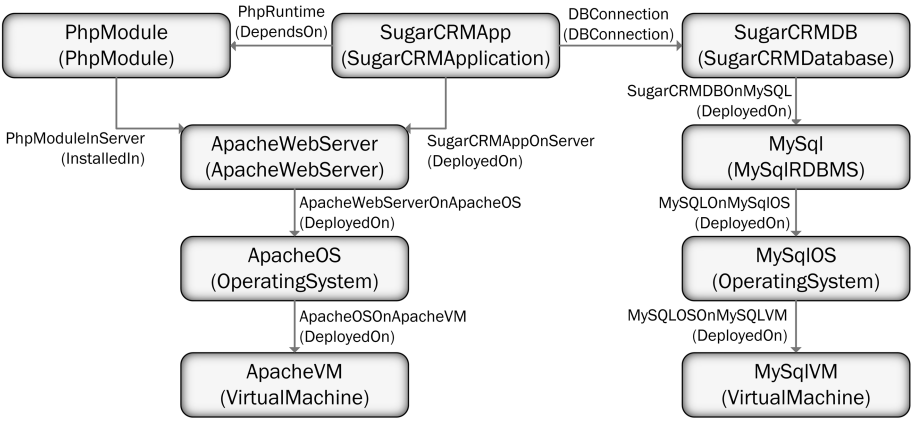

Fig. 3. Example of `TopologyTemplate`

means of typed NodeTemplates and RelationshipTemplates, respectively. A concrete example of an application topology is shown in Fig. 3, which illustrates the NodeTemplates and RelationshipTemplates composing the topology of the SugarCRM application. Fig. 3 also indicates the corresponding NodeTypes and RelationshipTypes between parentheses.

Application Components. As shown in Figs. 2 and 3, each application component appears in the topology as a NodeTemplate, and each NodeTemplate is in turn typed. This is because the purpose of NodeTemplates is to define the application-specific features of components (e.g., actual property values, QoS, etc.), while the purpose of the corresponding types is to describe the structure of the features to be specified.

The structure of the features exposed by an application component is defined by means of NodeTypes [10]. More precisely, a NodeType specifies the structure of the observable properties of an application component, the management operations it offers, the possible states of its instances, the requirements needed to properly operate it, and 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), and capabilities with CapabilityDefinitions (of certain CapabilityTypes). An example of a NodeType is shown in Fig. 4, which illustrates the structure of the properties, requirements and interfaces exposed by the SugarCRMApp component.

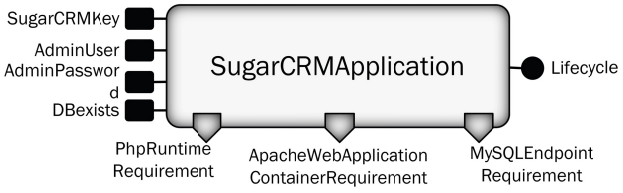

Fig. 4. Example of NodeType

Note that NodeTypes do not specify which are the artifacts required to instantiate and operate application components, since that is the purpose of NodeTypeImplementations. Each NodeTypeImplementation refers to the NodeType whose implementation is under definition and specifies its DeploymentArtifacts and ImplementationArtifacts [1]. The former are the contents (viz., ArtifactTypes and ArtifactTemplates) needed to materialize instances of application components, while the latter are those which implement management operations offered by application components [6].

Relations between Application Components. Complex multi-service applications require not only to model their components, but also the relations between them [5]. As for components, relations can be modeled by means of RelationshipTypes, RelationshipTypeImplementations, and RelationshipTemplates [1].

A RelationshipType defines the structure of a generic relationship between a ValidSource (i.e., a NodeType or a node's RequirementType) and a ValidTarget (i.e., a NodeType or a node's CapabilityType). It also allows to describe the operations which can be performed on the source and on the target of the

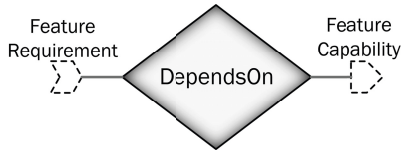

Fig. 5. Example of `RelationshipType`

relationship (via `SourceInterfaces` and `TargetInterfaces`, respectively), its observable properties, and the possible states of its instances. For instance, Fig. 5 illustrates the `DependsOn RelationshipType`, whose `ValidSource` is a `FeatureRequirement` exposed by an application component, and whose `ValidTarget` is a `FeatureCapability` offered by another application component. Such a `RelationshipType` is only one of those modeling the relations between the component of the `SugarCRM` application example.

Each `RelationshipType` requires to be connected with the artifacts implementing the operations it offers. This is the purpose of `RelationshipTypeImplementations` [1], each of which refers to a `RelationshipType` and specifies its `ImplementationArtifacts`. More precisely, a `RelationshipTypeImplementation` links each operation offered by a `NodeType` with the `ArtifactTypes` and `ArtifactTemplates` implementing it.

As for nodes, types and type implementations only describe relations in a generic way [5]. Once placed in the topological description of a certain application, they become application-specific and thus require to be described by means of `RelationshipTemplates` (to describe application-specific features).

Artifacts. An *artifact* represents the content needed to realize a deployment and/or management operation of an application component [5]. TOSCA allows artifacts to represent contents of any type (e.g., script, executable program, installable image, configuration file, library, etc.). This requires to describe artifacts along with the metadata needed to properly access them. The structure of such metadata is described by means of `ArtifactTypes`, while links to concrete artifacts (and values of invariant metadata) that can be specified by employing `ArtifactTemplates` [1].

Management Plans. `Plans` enable the description of application deployment and/or management aspects [14]. Each `Plan` is a workflow combining the operations offered by the nodes in the topology [1]. TOSCA prescribes to use workflows to describe `Plans` (so as to leverage of their suitability to handle errors, exceptions and human interactions [6]), but it does not mandate the use of specific workflow language [3]. Furthermore, `Plans` are distinguished on the basis of their `planType`. There are only two predefined types of plans: the `BuildPlan` type models plans which initially create a new instance of a service template, while the `TerminationPlan` type is for plans used to terminate the existence of a service instance [1].

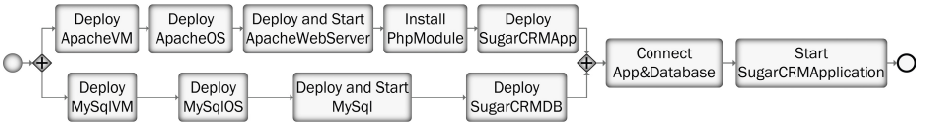

Fig. 6. Example of `Plan`

A concrete example of a TOSCA `Plan` is shown in Fig. 6, which illustrates a possible (BPMN) `BuildPlan` for the `SugarCRM` application example.

Application "Boundaries". A `ServiceTemplate` can also describe the functional and non-functional features it exposes externally. More precisely, the (optional) `BoundaryDefinitions` element allows to specify the properties, capabilities, requirements and operations of internal components which are externally visible. It also allows to expose management plans as operations and to describe the non-functional properties of the complex application.

Non-functional Features of an Application (Component). TOSCA employs policies to describe non-functional behavior and/or quality-of-service (QoS) that an application and its components can declare to expose [3]. Similar to the other entities in the TOSCA standard, a policy has an abstract `PolicyType` definition and is instantiated by defining a `PolicyTemplate`. While the `PolicyType` describes the structure and required parameters of a policy, the `PolicyTemplate` is used to define a specific policy instance [1].

`ServiceTemplate`s (via `BoundaryDefinitions`), `NodeTemplate`s, and `RelationshipTemplate`s can then declare their non-functional features by referring the `PolicyType` and/or `PolicyTemplate` describing them [13].

2.2 Packaging and Processing of Application Specifications

TOSCA also prescribes the format to archive application specifications along with the installable and executable files needed to properly instantiate the specified applications. This is because the modeling language illustrated in the previous section only allows developers to specify the application topology and its management and to give it in a `Definition.tosca` document. Such document must be packaged together with the artifacts implementing its components so as to make all such artifacts available to the execution environment.

Packaging of Application Specifications. The TOSCA specification defines an archive format called CSAR (*Cloud Serivce ARchive*) to package application specification together with concrete implementation and deployment artifacts [1]. A CSAR is a (compressed) zip file containing at least the `Definitions` and `TOSCA-Metadata` directories.

The `Definitions` directory contains one or more `Definitions.tosca` documents. These documents contain the TOSCA definitions describing the cloud application. More precisely, exactly one of them must contain the `ServiceTemplate` defining the structure and behavior of the whole cloud application, while the others can be devoted to supporting definitions (so as to modularize the application specification). Additionally, CSARs can also be devoted to contain TOSCA definitions to be reused in other contexts. For instance, a CSAR might be used to provide a set of `NodeType`s (with their corresponding implementations) to be employed as building blocks while specifying new cloud applications.

A `TOSCA-Metadata` directory contains the `TOSCA.meta` file. Its purpose is to describe metadata about the other files in the CSAR by means of blocks, which

in turn consist of a set of name-value pairs. More precisely, the first block of the `TOSCA.meta` file provides metadata about the CSAR itself (e.g., version, creator, etc.), while each other block points to a file in the CSAR and describes its metadata.

Processing of Application Packages. An application specification is packaged (along with the concrete artifacts implementing its components) in a CSAR archive with the purpose of deploying it on TOSCA-compliant cloud platforms. A cloud platform is TOSCA-compliant if it offers a TOSCA container (e.g., OpenTOSCA [8]) which is an engine able to process CSAR archives, and thus to deploy and operate the applications they contain.

TOSCA containers can deploy applications by processing the CSAR archives in two different ways [5]. On one hand, *imperative processing* takes the CSAR and deploys the application according to the workflow defined as a `BuildPlan` in the corresponding `ServiceTemplate` (e.g., the `BuildPlan` shown in Fig. 6). On the other hand, *declarative processing* deploys the application by trying to automatically excerpt a deployment plan from the application's `TopologyTemplate`. In the latter case, the CSAR engine (a) first deploys the nodes without requirements on other nodes, and then (b) until all nodes have been deployed, it searches the nodes whose requirements are satisfied (by the capabilities of the already deployed nodes) and deploys them. For instance, if we consider the topology in Fig. 3, the *declarative processing* works as follows. First, it deploys the node templates `ApacheVM` and `MySqlVM` since they have no dependencies on other nodes. Second, it deploys `ApacheOS` and `MySqlOS` since the node templates they depend on have been deployed. Then, it proceeds in repeating steps analogous to the second one until all the node templates in the topology have been deployed.

TOSCA containers not only have to support application at deployment time, but also at run time. They are indeed in charge of ensuring that the implementation artifacts (corresponding to management operations) are available [14]. They should also be able to properly operate such artifacts as well as the management plans provided by the application specification [3].

3 Analysis of the TOSCA Approach

In the previous section, we illustrated how TOSCA permits to describe the topology and management behaviour of multi-component cloud applications. It allows application developers to describe their solutions by clearly separating topology and management concerns. In this section we analyze the TOSCA approach for describing an application's topology and management, with respect to its declared main goals.

3.1 Topology Aspects

One of the main advantages of TOSCA is its suitability to (easily) represent the structure of (even complex) cloud applications. Each multi-component application is indeed modeled as a graph, in which typed nodes correspond to the

application's components, and typed relationships represent the dependencies between these components.

The availability of an abstract topology description is necessary to achieve the goal of automating the deployment of applications [4]. The topology description (along with the artifacts connected to each component) indeed allows TOSCA containers to automatically excerpt the declarative plans needed to deploy the specified application [5]. The automated management also benefits from the topology description. Imperative management plans can indeed be implemented by orchestrating the operations offered by the nodes in the topology.

Furthermore, the topology description is portable [6]. Despite application developers have all the freedom in choosing the types of the elements composing a topology, this is understandable to every container (provided that also the type definitions are available to the same container). This is because TOSCA, with the aim of giving flexibility to application developers in deciding the types to be used, only prescribes how to *structure* the definition [1].

Finally, TOSCA enhances reuse. Each TOSCA definition may indeed be referred by more than one specification (in order to be reused) [5]. Consider, for instance, the definition of a server component. By defining `ServerType`, we can specify (abstractly) its observable properties, capabilities, requirements, and management operations. The `ServerType` can then be referred by `ServerTemplate1`, ..., `ServerTemplateN`, which are different templates whose structure has been defined only once. The same holds for `ServerType` implementations. Different providers can offer different `ServerTypeImplementation`s, each of which implements this `ServerType` according to the provider's running environment. Furthermore, type definitions can be refined through derivation [3]: If one needs an Apache server component, then she can reuse the definitions in `ServerType` by extending them into an `ApacheServerType`.

The above mentioned features come at the price of defining a bunch of (XML) TOSCA elements. For instance, to define the above mentioned server component, an application developer must specify a `ServerType`, a `ServerTemplate`, and a `ServerTypeImplementation`. The latter in turns needs the definition of a set of `ArtifactType`s and `ArtifactTemplate`s corresponding to the set of artifacts implementing the `ServerType`. Since all the previously mentioned definitions are required for a single component of an application, it is not difficult to imagine how many definitions are required for a complex, multi-component application. The heaviness of the specification can be however mitigated mainly by leveraging reuse of TOSCA definitions, and by employing graphical tools (like Winery [12]) while defining new application components.

3.2 Management Aspects

TOSCA enables the automated application deployment and management by capturing the knowledge of the application developers via the modeling of their management proven best practices [3]. More precisely, application developers can model their application management at two different levels of abstraction. `DeploymentArtifact`s and `ImplementationArtifact`s are used to implement

deployment and management operations of a single application component, while `Plans` allow to express higher level management tasks [14]. For instance, an artifact may implement the pausing of an application module, while a plan may pause the multi-component application (by employing such artifact).

Artifacts can be implemented in whatever programming language the application developers like. Analogously, application developers have all the freedom in choosing the workflow languages to model (both declarative and imperative) plans. Ideally, the employed workflow languages should satisfy the following requirements, as BPMN4TOSCA does [14]: (i) they should provide ways to access and modify properties of nodes and relationships, (ii) they should enable management plans to access TOSCA topology model, (iii) they should ease the selection of management operations offered by nodes, and (iv) they should support an easy and comfortable way to execute scripts on nodes.

The freedom given to application developers makes TOSCA really flexible. On the other hand, to ensure "portability of applications and their management", TOSCA containers must be able to process the set of artifacts and plans needed to execute the management operations and to instantiate component instances [1]. In other words, TOSCA containers must pay the cost of supporting a bunch of languages and of being able to bind management of analogously defined operations to different kinds of artifacts.

3.3 Other Aspects

Besides topology and management aspects, TOSCA also allows application developers to specify the non-functional properties of their applications. Non-functional properties are expressed in TOSCA by means of policies, which in turn can be written with whatever policy language an application developer likes. This empowers the flexibility of TOSCA, but at the price of requiring TOSCA containers to support a bunch of policy languages.

Furthermore, the purpose of policies is to declare which non-functional properties an application offers. Thus, to specify what an application requires, application developers are asked to employ policies in a somewhat counter-intuitive way (by mixing what the application offers and what it requires). We argue that to split policies in *non-functional capabilities* and *non-functional requirements*, similar to functional requirements and capabilities could be a better alternative.

The flexibility of TOSCA is even more visible in the possibility of deploying CSAR archives both *imperatively* and *declaratively*. This gives freedom to application developers, by allowing them to either explicitly specify how to deploy their applications, or to ask containers to excerpt deployment plans from the application topology. This freedom comes at the price of requiring TOSCA containers to support both ways of processing.

In summary, TOSCA achieves its goals — automated application deployment and management, portability of application descriptions and their management, and interoperability and reusability of components — by also trying to be as much flexible as possible. On one hand, such a flexibility gives application developers freedom in choosing languages and types to be used while specifying

their applications. On the other hand, it obviously requires TOSCA containers to support a bunch of languages, and this complicates the development (and potentially also the operation) of TOSCA containers.

4 Research Directions

In the previous section we discussed the TOSCA approach for describing topology and behaviour of cloud applications. In this section we exploit such an analysis to try to identify a set of possible interesting research directions.

(D1) Fostering the Reuse of TOSCA Specifications. Cloud applications can share some management infrastructure. For instance, web applications (independently of their purposes) share an underlying topology whose top component is the web server needed to run them. If the underlying topology (and the related management) is already somehow available, it can be included in the specification and then suitably configured. In this way, the time and complexity required for application specification could be considerably decreased. It is thus interesting to identify reusable (fragments of) specifications so as to speed-up the development of new ones.

(D2) Enhanced and Full-fledged Implementations of TOSCA Containers. TOSCA aims to achieve its objectives by remaining as much flexible as possible. This also means to not prescribe (i) how to select whether to process a CSAR archive either declaratively or imperatively —if both are possible—, (ii) how to decide which build plan is to be invoked to imperatively processed when more than one are available, and (iii) how to select the proper type implementation when multiple are present. While issue (i) can easily be fixed by extending the TOSCA specification issues (ii) and (iii) may not be satisfactorily solved by simply extending the TOSCA specification, since they involve the development of proper selection criteria, and the implementation of mechanisms and tools which operate these criteria. Thus, it may be worth investigating issues (ii) and (iii) so as to gain smart and effective solutions.

(D3) Implementation of TOSCA Tools. Another interesting research direction is obviously the development of tools capable of working with TOSCA specifications (e.g., visual editors, analyzers, etc.) which can contribute to a widespread adoption of TOSCA.

(D4) Integration of TOSCA with Existing Standard Proposals. Another interesting direction is to investigate how TOSCA can be integrated with other initiatives. For instance, it is interesting to understand whether and how TOSCA can be integrated with CAMP, another emerging standard targeting the management of cloud applications. It is also interesting to understand which of the existing workflow modeling languages (e.g., BPMN, WS-BPEL, etc.) may be more suited for writing TOSCA plans.

(D5) Comparative Assessment of TOSCA. Since TOSCA is emerging, it still has to be accepted as the *de-facto* standard for the management orchestra-

tion of cloud applications. It is thus really interesting to devote further investigation to comparatively assess TOSCA with respect to other proposals that permit to specify cloud applications (e.g., CAMP). Such an assessment may be performed in terms of the expressive power of the language, the heaviness of the specifications, and the exploitability of the specification for analysis and verification.

We shall now expand the discussion regarding the above mentioned research directions. Due to space limitations, we will mainly focus on (D1), which is the scope of our immediate future work.

The reuse of TOSCA specifications can be fostered from two different perspectives: (i) (flexible) matching of available topology fragments with required node types, and (ii) identification of common management patterns. In this way, application developers become able to model their application without taking care of the underlying infrastructure. Once the application is modeled, they can indeed look for TOSCA nodes corresponding to PaaS offerings, select the most suited one (possibly on the basis of desired QoS), and then just include it as a single node in their application specification.

Informally speaking, (i) consists of determining a fragment of an available application specification that can become a standalone TOSCA service to be included in place of a desired node type while specifying new cloud applications. This may be done only from a functional perspective, or also by including non-functional features of desired nodes and available applications.

In case of (ii), starting from a bunch of cloud application specifications, it may be interesting to identify recurring substructures (modeling the same node types) and to export them as *management patterns*. The identified management patterns could then be merged with other patterns and definitions so as to build-up whole applications. This requires to solve two main issues, namely: how to merge the topologies, and how to merge the deployment and management plans. The former issue has already been studied [9], but the provided solution is no longer applicable (since it thoroughly employs `GroupTemplate`s, which are no longer supported by TOSCA). So, there is a need for new solutions that can either be based on the available approach [9] or not. The merging of plans has not yet been studied in the TOSCA context, but it is strongly related to the research work on web service composition. Some available solutions can then be employed in order to solve this new, TOSCA-related issue.

The above discussion about the reuse of available definitions implicitly assumes the ability to detect the TOSCA definitions corresponding to needed components. However, it is worth noting that TOSCA models application components from a management perspective, while application developers search them from an operational viewpoint. For instance, an application developer needing a web server searches a middleware component able to run web applications, rather than a component which offers the server-related management features. Thus, a mechanism to map an application developer's operational needs to TOSCA management definitions is an interesting research perspective.

Such a mechanism would allow to build-up a repository which lets application developers satisfy their (operational) needs with available TOSCA (management) definitions. TOSCA will benefit from such a repository, since the availability of easy-to-find, reusable definitions will strongly simplify the specification of applications in TOSCA, and thus exponentially decrease the time needed to do it. Furthermore, a repository of official definitions also empowers the portability of application specifications because TOSCA containers should support all of them.

Portability and reusability of application specifications are even more effective if the repository addresses the issue of having application components offering the same management features with similar (but different) names. A solution may be to provide a super-type standardizing the name of common features to be implemented by all derived definitions (maybe according to emerging API standardization like CAMP [16]), so that containers can uniformly understand them. Another solution may be to make the repository able to match available specification with respect to needed ones, and to suitably adapt them [10].

5 Related Work

At the time of writing, TOSCA [1] is a hot research topic. This is witnessed by the amount of research work which has already been produced, despite the young age of TOSCA.

On the one hand, some research efforts are targeted at illustrating what TOSCA is and how to use it. The primer [5] illustrates how TOSCA should be employed to specify complex applications and their management. More precisely, it identifies the three possible usage roles (viz., *application architect*, *type architect*, and *artifact developer*) and shows how they should employ TOSCA. The primer also discusses how CSAR archives are declaratively and imperatively processed. Binz et al. [3] outline the main goals of TOSCA and then discuss how it achieves them. The discussion starts with a very high-level overview of TOSCA, and then proceeds by illustrating how TOSCA achieves its goals. Lipton [7] and Binz et al. [6] overview TOSCA a bit more in detail, with the aim at highlighting the portability of TOSCA specifications, thus showing how TOSCA avoids the cloud vendor lock-in problem. Each of the aforementioned efforts discusses general aspects of TOSCA, either focusing on the modeling language or on other aspects like the processing of specifications or its goals (and sub-goals). In this paper, we tried to reorganize the aforementioned available information in a compact, easy-to-access description which comprises both the TOSCA modeling language and the other aspects.

On the other hand, several researches are related to TOSCA, but do not target at illustrating TOSCA itself. These researches can be considered in line with the research directions individuated in this paper. Brogi et al. [10] aim at instantiating desired node types by reusing existing service templates, and thus define four types of matching between service templates and node types (and show how to adapt service templates, if needed). Since [10] illustrates how to

match and adapt available TOSCA definitions, this can be considered in line with (D1). Binz et al. [9] are also strongly related with (D1), since they show how to improve resource sharing by merging the topologies of available cloud applications. OpenTOSCA [8] and Winery [12] are a container and a visual editor for TOSCA, respectively. Thus, OpenTOSCA is related to (D2), while Winery is in line with (D3). Kopp et al. [14] and Cardoso et al. [11] work in the direction (D4), by trying to integrate TOSCA with BPMN and USDL, respectively. Finally, Waizenegger et al. [13] illustrate two possible mechanisms for automatically processing policies expressed according to TOSCA, which are in line with (D3), since they can be easily implemented as a TOSCA tool.

6 Conclusions

As mentioned in the Introduction, interested readers can browse various documents to get acquainted with TOSCA. In this paper, we reorganized the available information so as to provide a compact, easy-to-access description of TOSCA which may speed-up the learning process of this promising OASIS specification, thus leveraging its widespread acceptance.

We have also discussed how TOSCA achieves its goals — automated application deployment and management, portability of application descriptions and their management, and interoperability and reusability of components — by also trying to be as flexible as it can. We also discussed how a reduction of such a flexibility (e.g, by reducing the number of supported plan/artifact languages) may empower the portability of application descriptions across different TOSCA containers.

In this paper, we also individuated some research perspectives, namely: (D1) reuse of available specifications, (D2) enhanced and full-fledged implementations of so-called *TOSCA containers*, (D3) implementation of TOSCA-about tools, (D4) integration of TOSCA with existing standard proposals, and (D5) comparative assessment of TOSCA. (D1) and (D5) are scope of our future work.

As a final remark, it is worth highlighting that TOSCA is not the *de-facto* standard for the interoperable specification of cloud applications. Its widespread adoption depends not only on its potential, but also on commercial and economical decisions. In this perspective, TOSCA may leverage of the set of big companies (e.g., Alcatel-Lucent, CA Technologies, Fujitsu, Huawei, IBM, SAP) which are active members of the OASIS TOSCA WG[3].

References

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

[3] The full list of OASIS TOSCA WG members can be found at `https://www.oasis-open.org/committees/membership.php?wg_abbrev=tosca`.

2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53, 50–58 (2010)
3. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable automated deployment and management of cloud applications. In: Bouguettaya, A., Sheng, Q.Z., Daniel, F. (eds.) Advanced Web Services, pp. 527–549. Springer, New York (2014)
4. Wettinger, J., Andrikopoulos, V., Strauch, S., Leymann, F.: Enabling dynamic deployment of cloud applications using a modular and extensible PaaS environment. In: 2013 IEEE Sixth International Conference on Cloud Computing (CLOUD), pp. 478–485 (2013)
5. OASIS: Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer Version 1.0 (2013), http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf
6. Binz, T., Breiter, G., Leymann, F., Spatzier, T.: Portable Cloud Services Using TOSCA. IEEE Internet Computing 16, 80–85 (2012)
7. Lipton, P.: Escaping Vendor Lock-in with TOSCA, an emerging Cloud Standard for Portability. CA Technology Exchange 4, 49–55 (2013)
8. Binz, T., Breitenbücher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA – a runtime for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 692–695. Springer, Heidelberg (2013)
9. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F., Weiss, A.: Improve Resource-Sharing through Functionality-Preserving Merge of Cloud Application Topologies. In: Desprez, F., Ferguson, D., Hadar, E., Leymann, F., Jarke, M., Helfert, M. (eds.) Proceedings of the 3rd International Conference on Cloud Computing and Service Science, CLOSER 2013, Aachen, Germany, May 8-10, 8 pages. SciTePress (2013)
10. Canal, C., Villari, M. (eds.): ESOCC 2013. CCIS, vol. 393, pp. 218–232. Springer, Heidelberg (2013)
11. Cardoso, J., Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: Cloud computing automation: Integrating USDL and TOSCA. In: Salinesi, C., Norrie, M.C., Pastor, Ó. (eds.) CAiSE 2013. LNCS, vol. 7908, pp. 1–16. Springer, Heidelberg (2013)
12. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – A Modeling Tool for TOSCA-Based Cloud Applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 700–704. Springer, Heidelberg (2013)
13. Waizenegger, T., et al.: Policy4TOSCA: A policy-aware cloud service provisioning approach to enable secure cloud computing. In: Meersman, R., Panetto, H., Dillon, T., Eder, J., Bellahsene, Z., Ritter, N., De Leenheer, P., Dou, D. (eds.) ODBASE 2013. LNCS, vol. 8185, pp. 360–376. Springer, Heidelberg (2013)
14. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: BPMN4TOSCA: A domain-specific language to model management plans for composite applications. In: Mendling, J., Weidlich, M. (eds.) BPMN 2012. LNBIP, vol. 125, pp. 38–52. Springer, Heidelberg (2012)
15. Pahl, C., Zhang, L., Fowley, F.: Interoperability standards for cloud architecture. In: Desprez, F., Ferguson, D., Hadar, E., Leymann, F., Jarke, M., Helfert, M. (eds.) CLOSER. SciTePress (2013)
16. OASIS: Cloud Application Management for Platforms (CAMP) Version 1.1 (2014), http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf

17. DMTF: Cloud Infrastructure Management Interface, CIMI (2013), http://www.dmtf.org/sites/default/files/standards/documents/DSP0264_1.0.0.pdf
18. Open Mashup Alliance: Enterprise Mashup Markup Language, EMML (2011), https://en.wikipedia.org/wiki/EMML
19. Open Grid Forum: Open Cloud Computing Interface, OCCI (2013), http://occi-wg.org/about/specification/
20. OASIS: Open Component Service Architectures, Open-CSA (2007), http://www.oasis-opencsa.org/specifications
21. DMTF: Open Virtualization Format, OVF (2014), http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_2.1.0.pdf
22. OMG: Service Oriented Architecture Modeling Language, SOA-ML (2012), http://www.omg.org/spec/SoaML/1.0.1/
23. W3C: Unified Service Description Language, USDL (2011), http://www.w3.org/2005/Incubator/usdl/XGR-usdl-20111027/