SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2018; **48**:11,2061–2079
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.2625

TosKer: A synergy between TOSCA and Docker for orchestrating multi-component applications

Antonio Brogi, Luca Rinaldi, and Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

SUMMARY

How to flexibly manage complex applications across heterogeneous cloud platforms is one of the main concerns in today's enterprise IT. The OASIS standard TOSCA and the Docker ecosystem are two emerging solutions trying to address this problem from different perspectives. In this paper we propose a solution that tries to synergically combine the pros of both TOSCA and of Docker. More precisely, we propose a TOSCA-based representation for specifying the software components and the Docker containers forming an application. We also present TosKer, an engine for orchestrating the management of multi-component applications based on the proposed TOSCA representation and on Docker. Finally, we illustrate how TosKer was fruitfully exploited in a concrete case study, based on a third-party application. Copyright © 2018 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Cloud; application orchestration; TOSCA; Docker

1. INTRODUCTION

Cloud computing has revolutionised IT, by allowing to run on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [5]. At the same time, the problem of automating the management of (complex) multi-component applications over heterogeneous cloud infrastructures is receiving increasing attention [17, 27]. The OASIS standard TOSCA (*Topology and Orchestration Specification for Cloud Applications* [32]) and the Docker ecosystem [19] are two emerging solutions trying to address this problem from different perspectives.

On the one hand, TOSCA [32] follows a model-driven approach. It indeed provides a YAML-based modelling language for specifying portable cloud applications, and for automating their deployment and management. TOSCA permits describing the structure of a cloud application as a typed, directed topology graph, whose nodes represent application components, and whose arcs represent dependencies among such components. Each node of a topology can also be associated with the corresponding components requirements, the operations to manage it, the capabilities it features, and the policies applied to it. Inter-node dependencies associate the requirements of a node with the capabilities featured by other nodes. An application topology can then be declaratively processed [23] to automatically deploy such application on a TOSCA-compliant cloud platform.

On the other hand, Docker [19] follows a "snapshot-based" approach. Docker is the de-facto standard for container-based virtualisation [34], and it permits packaging software components (together with all software dependencies they need to run) in Docker *images*, which are then exploited as read-only templates to create and run Docker *containers*. Docker containers can also mount external *volumes*, which ensure data persistence independently of the lifecycle of containers [29]. Docker also permits orchestrating containers, by allowing to define multi-container Docker applications [36]. Given (the images of) the containers forming a multi-container

application, the volumes they must mount, Docker compose [20] is indeed capable of automatically deploying the corresponding application.

Of course, both TOSCA and Docker have pros and cons. For instance, while TOSCA is a well-documented standard that permits orchestrating complex applications formed by heterogeneous components, application descriptions tend to become quite verbose. Docker instead is a production ready tool, with a huge repository of images (Docker Hub [21]), but it was not designed to orchestrate complex applications composed by multiple and heterogeneous components. Indeed, Docker containers are treated as "black-boxes", and they constitute the minimum deployment entity considered by currently existing approaches for deploying multi-component applications with Docker (e.g., [3, 20, 22, 38]).

The objective of this paper was to identify and develop a solution that synergically combines the pros of both TOSCA and Docker. A concrete solution is to still rely on Docker containers as a portable and lightweight mean to deploy application components on cloud platforms, by also allowing to independently manage the components and containers forming a multi-component application [33]. In this paper we propose a solution following this idea, which relies on the OASIS standard TOSCA [32] for specifying the structure and management behaviour of multi-component applications, and for orchestrating them on top of Docker containers. More precisely, our main contributions are the following:

- We propose a TOSCA-based representation for multi-component applications. The latter permits to modularly specify the requirements, capabilities, management operations, policies, and properties of the software components forming an application, along with those of the Docker containers and Docker volumes needed to run them. The proposed representation also permits indicating the relationships occurring among the components of an application (e.g., a software component is hosted on a container, a component connects to another).
- We introduce a default management behaviour of software components, Docker containers and Docker volumes. We also show how to customise the management behaviour of the software components forming a TOSCA application.
- We also present TOSKER, an engine for orchestrating the management of multi-component applications based on the proposed TOSCA representation, and on Docker.

We also show how we fruitfully exploited the proposed TOSCA-based representation and TOSKER on a concrete case study, based on a third-party multi-component application (viz., *Sock shop* [39]).

A short description of a first prototype of TOSKER was provided in [14]. This article extends [14] (i) by integrating management protocols [7] with the TOSCA-based representation for multi-component applications to permit customising the management behaviour of application components, (ii) by presenting an extended prototype of TOSKER, which now supports the orchestration of software components whose structure and management behaviour has been customised, and (iii) by including a discussion of how TOSKER has been fruitfully exploited on a concrete case study.

The rest of the paper is organised as follows. Sect. 2 provides some background on TOSCA, Docker and management protocols. Sect. 3 presents a multi-component application that will be used as running example. Sect. 4 illustrates the proposed TOSCA-based representation for multi-component applications. Sect. 5 illustrates the design of TOSKER. Sect. 6 details our prototype implementation of TOSKER. Sect. 7 presents the case study based on the *Sock shop* application. Finally, Sects. 8 and 9 discuss related work and draw some concluding remarks.

2. BACKGROUND

2.1. Docker

Docker [19] is a Linux-based platform for developing, shipping, and running applications through container-based virtualisation. Container-based virtualisation [37] exploits the kernel of the operating system of a host to run multiple isolated user-space instances, called *containers*.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

Each Docker container packages the applications to run, along with whatever software support they need (e.g., libraries, binaries, etc.). Containers are built by instantiating so-called Docker *images*, which can be seen as read-only templates providing all instructions needed for creating and configuring a container. Existing Docker images are distributed through so-called Docker *registries* (e.g., Docker Hub [21]), and new images can be built by extending existing ones.

Docker containers are volatile, and the data produced by a container is (by default) lost when the container is deleted. This is why Docker introduces *volumes*, which are specially-designated directories (shared among containers) whose purpose is to persist data, independently of the lifecycle of the containers mounting them. Docker never automatically deletes volumes when a container is removed, nor it removes volumes that are no longer referenced by any container.

Docker also allows containers to intercommunicate. It indeed permits creating virtual networks, which span from bridge networks (for single hosts), to complex overlay networks (for clusters of hosts).

2.2. TOSCA

TOSCA (*Topology and Orchestration Specification for Cloud Applications* [32]) 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 and machine-readable modelling language that permits describing cloud applications. Obtained specifications can then be processed to automate the deployment and management of the specified applications.

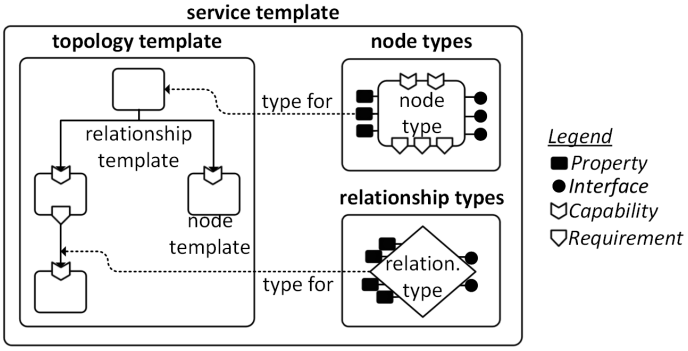

Figure 1. The TOSCA metamodel [32].

TOSCA permits specifying a cloud application as a service template, that is in turn composed by a topology template, and by the types needed to build such a topology template (Fig. 1). The topology template is a typed directed graph that describes the topological structure of a multi-component application. Its nodes (called node templates) model the application components, while its edges (called relationship templates) model the relations occurring among such components.

Node templates and relationship templates are typed by means of node types and relationship types, respectively. A node type defines the observable properties of a component, its possible requirements, the capabilities it may offer to satisfy other components' requirements, and the interfaces through which it offers its management operations. Requirements and capabilities are also typed, to permit specifying the properties characterising them. A relationship type instead describes the observable properties of a relationship occurring between two application components. As the TOSCA type system supports inheritance, a node/relationship type can be defined by extending another, thus permitting the former to inherit the latter's properties, requirements, capabilities, interfaces, and operations (if any).

Node templates and relationship templates also specify the artifacts needed to actually realise their deployment or to implement their management operations. As TOSCA allows artifacts to represent contents of any type (e.g., scripts, executables, images, configuration files, etc.), the metadata needed to properly access and process them is described by means of artifact types.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

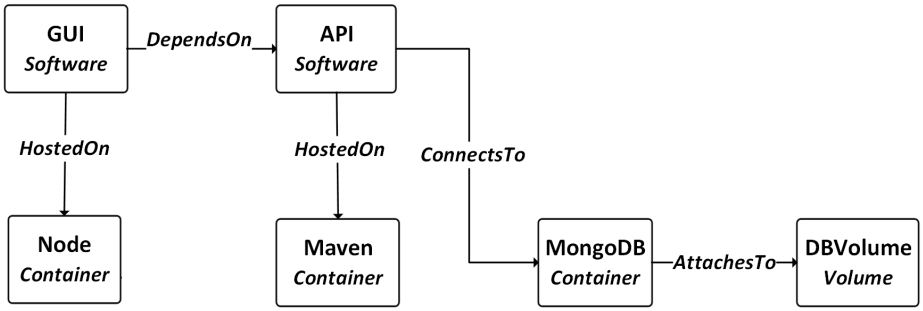

Figure 2. The architecture of the Thinking application.

TOSCA applications are then packaged and distributed in CSARs (*Cloud Service ARchives*). A CSAR is a zip archive containing an application specification along with the concrete artifacts realising the deployment and management operations of its components.

2.3. Management protocols

Management protocols [7] permit describing the management behaviour of the components forming the topology of a TOSCA application. Management protocols are finite state machines whose states and transitions are associated with conditions on the components requirements and capabilities. Intuitively speaking, the objective of those conditions is to define the consistency of component states and to constrain the executability of component operations to the satisfaction of their requirements.

The management behaviour of a TOSCA application is derived by composing the management protocols of its components, according to the application's topology. The global state of an application is indeed defined as the set containing the current state of each of the application components. A global state is considered to be "valid" only if all requirements assumed to hold by a node are connected to capabilities that are actually provided by another node (in such global state, of course). An application can only transit from one valid global state to another, by executing a management operation on a component (provided that all requirements needed to execute such operation are satisfied in the starting global state).

The derived management behaviour can then be exploited to automate various useful analyses. For instance, given a plan orchestrating the management operations of a TOSCA application to achieve some management goal, one can readily check whether such plan is valid by verifying that it can only traverse valid global states. Concrete examples of management protocols can be found in Sect. 4.2.

3. RUNNING EXAMPLE

We hereby present the *Thinking* open-source web application*, which will be used as a reference example in Sects. 4 and 5. *Thinking* allows users to share their thoughts so that all other users can read them through a web-based graphical user interface. *Thinking* is composed of three main components, namely (i) a Mongo database storing the collection of thoughts shared by end-users, (ii) a Java-based REST API to remotely access the database of shared thoughts, and (iii) a web-based GUI visualising all shared thoughts and allowing to insert new thoughts into the database. Fig. 2 illustrates a representation of the *Thinking* application:

*The source code of *Thinking* is publicly available on GitHub at `https://github.com/di-unipi-socc/thinking`.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

(i) The database is obtained by directly instantiating a *MongoDB* container, which needs to be attached to a volume where the shared thoughts will be persistently stored.

(ii) The *API* is hosted on a *Maven* Docker container, and it requires to be connected to the *MongoDB* container (for remotely accessing the database of shared thoughts).

(iii) The *GUI* is hosted on a *NodeJS* Docker container, and it depends on the availability of the *API* to properly work (as it sends GET/POST requests to the *API* to retrieve/add shared thoughts).

The *GUI* and the *API* are also provided with a set of shell scripts to implementing their lifecycle operations. Namely, they are both equipped with a set of scripts (viz., `install.sh`, `configure.sh`, `start.sh`, `stop.sh`, `uninstall.sh`) implementing the operations to *install*, *start*, *configure*, *stop*, *uninstall* them. The *API* is also equipped with the script `push_default.sh`, which can be optionally executed when the *API* is configured (but not running) to add a default set of thoughts to the *MongoDB* database. To effectively manage the *GUI* and the *API*, their management operations must be executed in a given (partial) order, and each of them can be executed only if some requirements are satisfied (e.g., the container hosting the corresponding component is up and running).

4. SPECIFYING MULTI-COMPONENT APPLICATIONS

We hereby introduce the TOSCA types that permit specifying the topology and management behaviour of a multi-component application (in Sect. 4.1 and 4.2, respectively). These types are designed to allow TOSKER to orchestrate the deployment and management of specified applications.

4.1. Node, relationship and artifact types

Multi-component applications typically integrate various and heterogeneous components [25]. We hereby define a TOSCA-based representation for such components, as well as for the Docker containers and Docker volumes that will be used to form their runtime infrastructure.

We first define three different TOSCA node types[†] (Fig. 3) to permit distinguishing the Docker containers, the Docker volumes, and the application components forming a multi-component application.

- *tosker.nodes.Container* permits representing Docker containers, by indicating whether a container requires a *connection* (to another Docker container or to an application component), whether it has a generic *dependency* on another node in the topology, or whether it needs some persistent *storage* (hence requiring to be attached to a Docker volume). *tosker.nodes.Container* also permits indicating whether a container can *host* an application component, whether it offers an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of another container/application component). To complete the description, *tosker.nodes.Container* can contain properties (*ports*, *env_variables*, *command*, and *share_data*, respectively) for specifying the port mappings, the environment variables, the command to be executed when running the corresponding Docker container, the list of files and folders to share with the host. Finally, *tosker.nodes.Container* lists the operations to manage a container (which corresponds to the basic operations offered by the Docker platform to manage Docker containers [29]).
- *tosker.nodes.Volume* permits specifying Docker volumes, and it defines a capability *attachment* to indicate that a Docker volume can satisfy the *storage* requirements of Docker

[†]The complete definition of all TOSCA types discussed in this section is publicly available on GitHub at `https://github.com/di-unipi-socc/tosker-types`.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

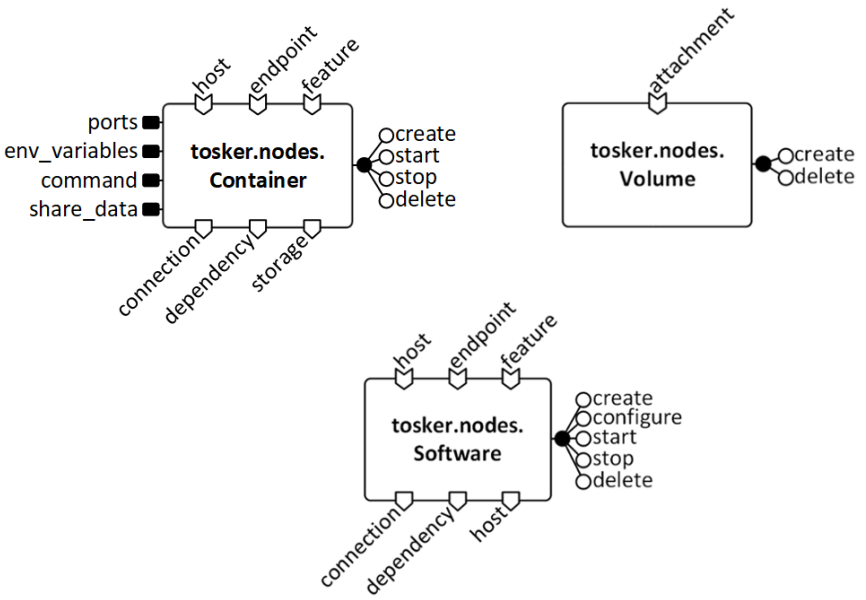

Figure 3. TOSCA node types for multi-component, Docker-based applications, viz., *tosker.nodes.Container*, *tosker.nodes.Software*, and *tosker.nodes.Volume*.

containers. It also lists the operations to manage a Docker volume (which corresponds to the operations to create and delete Docker volumes offered by the Docker platform [29]).

- *tosker.nodes.Software* permits indicating the software components forming a multi-component application. It permits specifying whether an application component requires a *connection* (to a Docker container or to another application component), whether it has a generic *dependency* on another node in the topology, and that it has to be *host*ed on a Docker container or on another component. *tosker.nodes.Software* also permits indicating whether an application component can *host* another application component, whether it provides an *endpoint* where to connect to, or whether it offers a generic *feature* (to satisfy a generic *dependency* requirement of a container/application component). Finally, *tosker.nodes.Software* lists the operations to manage an application component by exploiting the TOSCA standard lifecycle interface [32] (viz., *create*, *configure*, *start*, *stop*, *delete*).

It is worth noting that TOSCA supports inheritance [32], hence allowing to extend the TOSKER types to define new types. For instance, if a software component needs an additional requirement, offers an additional capability or provides additional management operations needed to customise its management behaviour, then it is possible to derive a new type from *tosker.nodes.Software* and to use such type to model the software component.

The interconnections and interdependencies among the nodes forming a multi-component application can be indicated by exploiting the TOSCA normative relationship types [32].

- *tosca.relationships.AttachesTo* can indeed be used to attach a Docker volume to a Docker container.
- *tosca.relationships.ConnectsTo* can indicate the network connections to establish between Docker containers and/or application components.
- *tosca.relationships.HostedOn* can be used to indicate that an application component is hosted on another component or on a Docker container (e.g., to indicate that a web service is hosted on a web server, which is in turn hosted on a Docker container).
- *tosca.relationships.DependsOn* can be used to indicate generic dependencies between the nodes of a multi-component application (e.g., to indicate that a component must be deployed before another, as the latter depends on the availability of the former to properly work).

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

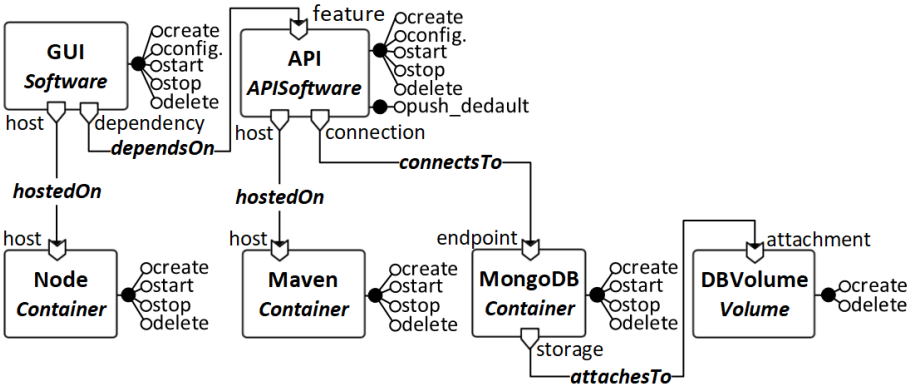

Figure 4. The TOSCA representation of the Thinking application shown in Fig. 2.

Example 1. Consider again the *Thinking* application of Sect. 3. Fig. 4 illustrates a representation of the *Thinking* application in TOSCA, where the three main components (viz., *MongoDB*, *API* and *GUI*) are modelled as follows:

(i) *MongoDB* is modelled as a node of type *tosker.nodes.Container* and it is attached to a volume (*DBVolume*) through a relationship of type *tosca.relationships.AttachesTo*.

(ii) *API* is modelled as a node of type *APISoftware* hosted on a node of type *tosker.nodes.Container* (viz., *Maven*). *APISoftware* is a customised type derived from *tosker.nodes.Software*, which adds a new interface containing the operation *push_default*. The connection from *API* to *MongoDB* is modelled as a relationship of type *tosca.relationships.ConnectsTo* (connecting the requirement *connection* of *API* to the capability *endpoint* of *MongoDB*).

(iii) *GUI* is modelled as a node of type *tosker.nodes.Software* and it is hosted on a node of type *tosker.nodes.Container* (viz., *NodeJS*). The fact that *GUI* depends on the availability of *API* is modelled with a relationship of type *tosca.relationships.DependsOn* (connecting the requirement *dependency* of *GUI* to the capability *feature* of *API*). □

Finally, also artifacts must be typed [32], as they are used to implement deployment and management operations of the nodes forming a multi-component application and they must specify the metadata needed to properly access and process them. We hence define *tosker.artifacts.Image* and *tosker.artifacts.Dockerfile* to permit indicating that an artifact is an actual image or a Dockerfile, which will then be used to create a Docker container. We also extend such artifact types by defining *tosker.artifacts.Image.Service* and *tosker.artifacts.Dockerfile.Service*, to permit distinguishing images that execute a service when started from those that "simply package" a runtime environment. We can instead rely on TOSCA normative artifact types [32] for all other kinds of artifacts linked by the nodes in a multi-container Docker application.

Example 2. Consider again the *Thinking* application of Sect. 3. The image artifact associated to the *MongoDB* container is of type *tosker.artifacts.Image.Service*, as it links to an image offering a MongoDB server when executed. The image artifacts associated to the containers *Node* and *Maven* are instead of type *tosker.artifacts.Image*, as they link to images just offering runtime environments (for NodeJS-based and Maven-based applications, respectively). The management operations of *GUI* and *API* are instead implemented by ".sh" scripts[‡]. □

[‡]The resulting TOSCA application specification is publicly available at `https://github.com/di-unipi-socc/TosKer/blob/master/data/examples/thinking-app/thinking/thinking.yaml`. A CSAR packaging such specification (together with all artifacts needed to deploy and manage the *Thinking* application) is available at `https://github.com/di-unipi-socc/TosKer/blob/master/data/examples/thinking-app/thinking.csar`.

DOI: 10.1002/spe.2625

4.2. Management protocols

TOSKER node types are associated with a default management protocol describing their default management behaviour. Fig. 5 shows the default management protocols associated to the node types *tosker.nodes.Container*, *tosker.nodes.Volume*, and *tosker.nodes.Software*, which are designed to model the standard management lifecycle of Docker containers, Docker volumes, and software components, respectively.

The default management protocol for nodes of type *tosker.nodes.Container* is shown in Fig. 5.(a). The initial state of a Docker container is *deleted*, in which the container can only perform the operation *create* to become *created*. Once *created*, a Docker container can execute the operation *delete* to return to be *deleted*, or it can become *running* by executing the operation *start*. While *running*, a Docker container can return to be *created* by executing the operation *stop*. The *running* state is the only state where a Docker container assumes all its requirements to be satisfied, and where it continues to provide all its capabilities (hence satisfying all requirements connected to them).

The default lifecycle of nodes of type *tosker.nodes.Volume* is modelled by the management protocol in Fig. 5.(b). A Docker volume is initially *deleted*, viz., it is not present yet. From this state the volume can transit to the *created* state by executing the operation *create*. In the *created* state the volume offers the capability *attachment* and it can transit back to the *deleted* state by executing the operation *delete*.

The default management protocol for nodes of type *tosker.nodes.Software* is illustrated in Fig. 5.(c). The initial state of a software component is *deleted*, from which it can become *created* by executing its management operation *create*. Once *created*, a software component can either return to be *deleted* (by executing the operation *delete*), or it can become *configured* (by executing the operation *configure*). When in state *configured*, a software component can execute the operations *delete* (which leads it back to its initial state) and *start* (which makes it become *running*). In its state *running*, a component can only execute the operation *stop* to go back to its state *configured*. Notice that *running* is the only state where a component assumes its requirement to continue to be satisfied, and where it continues to provide all its capabilities (hence satisfying all requirements connected to them).

Customised management protocols can be associated with a node to customise its management behaviour. New management protocols can indeed be defined by creating policies of type *tosker.policies.Protocol*.

The *tosker.policies.Protocol* has three main properties to define a management protocol, namely *states*, *transitions*, and *initial_state*. The property *states* is a map $state \rightarrow conds$, where *state* is the name of a state, and *conds* the conditions on requirements and capabilities associated to such state (viz., which requirements must be satisfied and which capabilities are provided). The property *transitions* is a list of transitions, each of which indicates the *source* state of a transition, its *target* state, the list of the *requirement*s needed to fire such transition, and the *operation* to be executed to enact the transition. Finally, the property *initial_state* contains the name of the initial state of a management protocol.

Example 3. A new policy of type *tosker.policies.Protocol* is added to the specification of our running example, to define the management protocol of *API*. Its protocol (Fig. 6) is essentially analogous to that of nodes of type *tosker.nodes.Software* (viz., Fig. 5.c). The only difference is the additional transition allowing to self-loop over the *configured* state by executing the operation *push_default*. In this way two possible plans to run the component can be used, one which executes the *push_default* and the other which not executes it (viz. *create*, *configure*, *start* or *create*, *configure*, *push_default*, *start*). □

Finally, it is worth noting that the possibility of extending TOSKER types (by adding new requirements, capabilities or management operations), along with that of defining customised management protocols (through policies of type *tosker.policies.Protocol*), enables the customisation of the software components orchestrated by TOSKER.

DOI: 10.1002/spe.2625

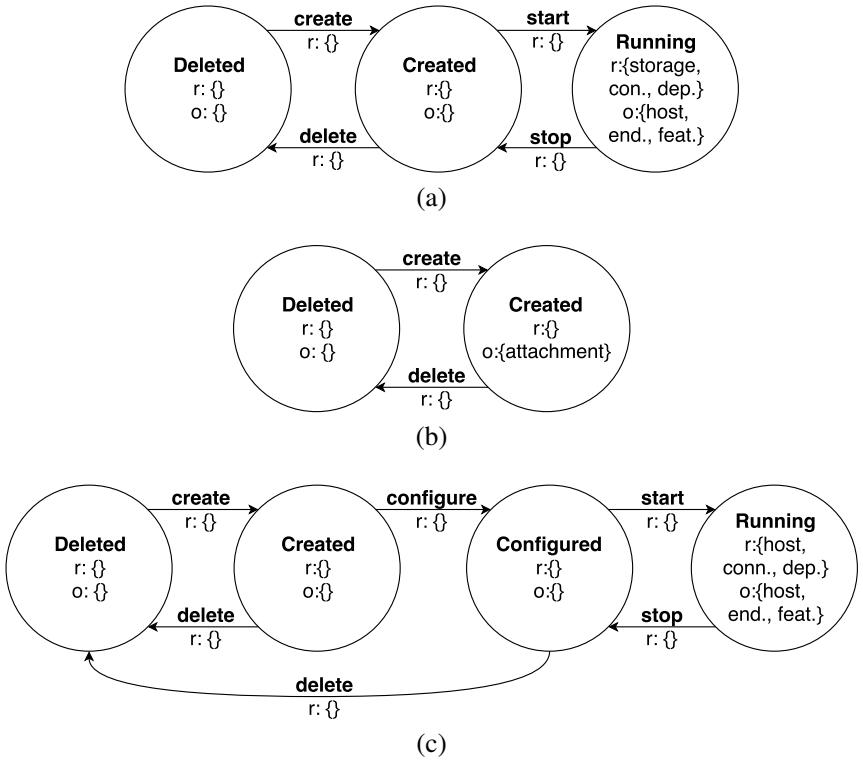

Figure 5. Default management protocol for nodes of type (a) *tosker.nodes.Container*, (b) *tosker.nodes.Volume*, and (c) *tosker.nodes.Software*, where *r* represent the requirements needed by a node in a state or to fire a transition, and where *o* represents the capabilities actually provided by a node in a state.

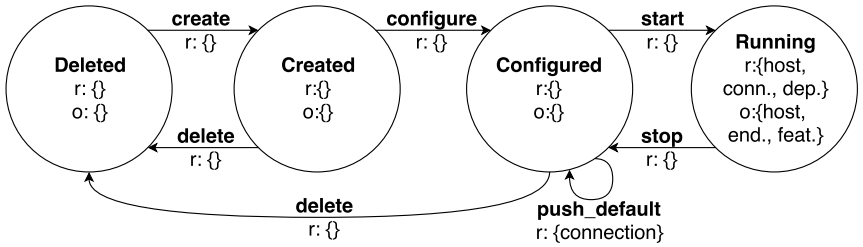

Figure 6. Customised management protocol for the component *API* of the application in our running example.

5. TOSKER ARCHITECTURE

Fig. 7 shows the architecture of TOSKER, an engine for orchestrating multi-component application based on our TOSCA-based representation and on Docker. TOSKER is designed to be modular and easily extensible. The architecture of TOSKER partitions the functionalities of TOSKER into lightweight modules that interact with each other, and new functionalities can be easily added to TOSKER by developing and plugging-in new modules.

User interface. The UI allows to feed TOSKER with the necessary input. The latter includes a CSAR (packaging the TOSCA specification of a multi-component application together with all artifacts needed to realise its management [32]), and a sequential plan listing the operations to be executed on the application components (specified as a list of triples ⟨*component, interface, operation*⟩).

Utilities. TOSKER exploits three utility modules, called TOSCA Parser, Plan Checker, and State Storage. The TOSCA Parser is a utility module for parsing a CSAR and generating an internal

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

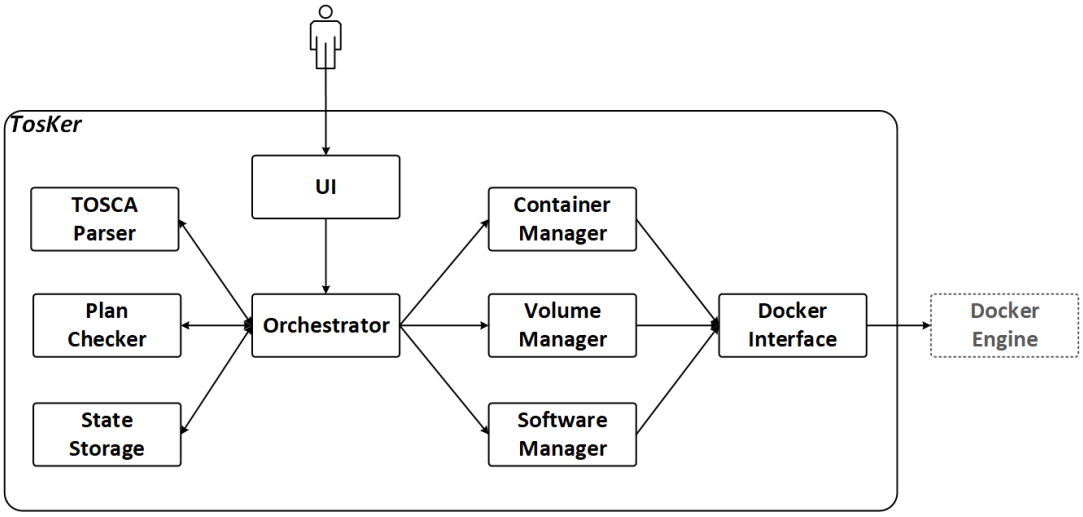

Figure 7. The architecture of TOSKER.

representation of the application it packages. Such representation will be exploited by the other modules in TOSKER to deploy and manage the corresponding application.

The Plan Checker implements the analysis based on management protocols, which permits checking whether a plan can be executed before actually executing it. More precisely, the Plan Checker is used to check whether a sequence of operations can be executed on an application in its current state. Each operation in the sequence can be executed on the corresponding component if and only if (i) there exist a corresponding transition in the component's management protocol, (ii) all the requirements needed to execute such transition are satisfied in the state where the transition has to be fired, and (iii) the execution of the transition does not result in stopping to provide capabilities needed by other components.

The State Storage is in charge of persistently storing the state of the components of the applications deployed by TOSKER. This permits remembering the states of the components among consecutive plan executions, hence allowing to enact the checks of plans.

Orchestration core. The Orchestrator is the core component of TOSKER, as it is in charge of orchestrating the management of multi-component applications. It first receives the inputs from the UI, and it exploits the TOSCA Parser to generate an internal representation of the multi-component application contained in the input CSAR.

The Orchestrator then checks the input plan on the application topology. It first loads the state of the application from the State Storage and it then checks whether the sequential plan can be executed by exploiting Plan Checker.

If the plan can be executed, then Orchestrator orchestrates the actual execution of the corresponding management operations by coordinating the Container Manager, Volume Manager and Software Manager. It indeed iterates over the sequence of management operations forming a plan, and it dispatches the actual execution of an operation on a component to the corresponding manager (e.g., to *create* a component of type *tosker.nodes.Container*, the Orchestrator dispatches the actual execution of *create* on such component to the Container Manager).

Managers. The Container Manager, Volume Manager, and Software Manager implement the actual lifecycle for components of type *tosker.nodes.Container*, *tosker.nodes.Volume*, and *tosker.nodes.Software*, respectively.

- The Container Manager is in charge of implementing the operations to *create*, *start*, *stop* and *delete* Docker containers, by also taking into account the different types of artifacts from which they are generated (viz., Docker images or Dockerfiles — see Sect. 4).

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

- The Volume Manager has to implement the operations to *create* and *delete* Docker volumes (as volumes can only be created or deleted, see Sect. 4).
- The Software Manager is in charge of implementing the operations to manage the lifecycle of the *tosker.nodes.Software*. The Software Manager supports the Standard lifecycle interface of TOSCA (which contains the operations *create*, *configure*, *start*, *stop* and *delete* a component), as well as customised interfaces (see Sect. 4).
  Notice that, as such a kind of components will be hosted on Docker containers, the actual execution of a management operation on a component requires to issue commands to its container. The Software Manager must hence copy all artifacts implementing the management operations of a software component within its container. This happens whenever a component needs to leave its initial state by executing a management operation. Copied artifacts are instead deleted whenever a component executes a management operation leading it back to its initial state.

Finally, as shown in Fig. 7, each manager implements management operations by instructing the Docker Interface on which Docker commands to execute.

Docker interface. The Docker Interface is in charge of interacting with the Docker engine installed on the host where TOSKER is running. It is used by the managers to manage Docker containers and Docker volumes, and to execute operations inside running containers.

Notice that the Docker Interface decouples TOSKER from the actual Docker engine used, meaning that it can issue commands to a classic Docker engine (as in the current implementation of TOSKER— see Sect. 6), but it could also be used to issue commands to an engine capable of distributing containers in a cluster (e.g., Docker swarm [22] or Kubernetes [38]).

6. PROTOTYPE IMPLEMENTATION OF TOSKER

We have implemented a prototype of TOSKER, which is open-source and publicly available on GitHub[§]. The prototype is written in Python[¶] and it is published on PyPI (*Python Package index*). We hereby illustrate how to use our prototype of TOSKER (Sect. 6.1) and how it actually works (Sect. 6.2, and we finally draw some concluding remarks on the implementation (Sect. 6.3).

6.1. Using TOSKER

The latest version of TOSKER can be installed on a host by simply executing the command `pip install tosker`. TOSKER can then be used as a standard Python library, or as a command line software by invoking `tosker` followed by one of the following commands:

- `exec`, which takes as input a TOSCA specification of an application and a plan, and it executes the plan on the application topology,
- `log`, which displays the log of the execution of a given management operation on a given software component,
- `ls`, which shows all the components deployed on the host and their current state,

The `exec` command has the following syntax:

```
$ tosker exec FILE --plan PLAN [INPUTS]..
```

where `FILE` is a CSAR archive or a TOSCA YAML file (containing the specification of a multi-component application), `PLAN` is a `.plan` file with a list of operations to be executed (the format is

[§]`https://github.com/di-unipi-socc/TosKer`.

[¶]The choice of Python was mainly motivated by the availability of two open-source Python libraries: *docker-py* (`https://github.com/docker/docker-py`) and *tosca-parser* (`https://github.com/openstack/tosca-parser/`). *docker-py* implements a Python interface for the Docker engine API. *tosca-parser* is instead a parser for TOSCA application specifications (developed by the OpenStack community).

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

thinking.up.plan

```
# create the DBvolume volume
dbvolume:Standard.create

# create and start the MongoDB container
mongodb:Standard.create
mongodb:Standard.start

# create and start the Node container
node:Standard.create
node:Standard.start

# create and start the Maven container
maven:Standard.create
maven:Standard.start

# create, configure, push_default and start the API software
api:Standard.create
api:Standard.configure
api:api_interface.push_default
api:Standard.start

# create, configure and start the GUI software
gui:Standard.create
gui:Standard.configure
gui:Standard.start
```

Figure 8. A plan to start all the components of the Thinking application.

`component:interface.operation`), `INPUTS` is an optional sequence of input parameters to be passed to the TOSCA application[‖].

It is also possible to directly execute one or more operations on the topology (instead of providing a plan):

```
$ tosker exec FILE OPERATIONS.. [INPUTS]..
```

where `FILE` and `INPUTS` are the same as before, while `OPERATIONS` is a list of operations in the format `component:interface.operation`

Example 4. Consider again the *Thinking* application in Sect. 3. Suppose that we wish to run all the components of the application by exploiting the plan in Fig. 8.
We can instruct TOSKER to do so, by executing:

```
$ tosker exec thinking.csar --plan=thinking.up.plan
```

TOSKER executes each operation in the plan (as shown in Fig. 9), and an instance of the Thinking application gets up and running. After the deployment, by executing the command `ls` we can visualise the state of the components of the application, as shown in Fig. 10.

Suppose now that we wish to delete the *GUI* of our running application. We can do it by executing the command:

```
$ tosker exec thinking.csar gui:Standard.stop \
                            gui:Standard.delete
```

[‖]Details on how to process inputs for TOSCA applications can be found in [32].

DOI: 10.1002/spe.2625

```
~ tosker exec thinking.csar --plan thinking.up.plan
(update memory: ok)
✔ Check deployment plan... Done
✔ Create network... Done
✔ Execute op "Standard.create" on "dbvolume"... Done
✔ Execute op "Standard.create" on "mongodb"... Done
✔ Execute op "Standard.start" on "mongodb"... Done
✔ Execute op "Standard.create" on "node"... Done
✔ Execute op "Standard.start" on "node"... Done
✔ Execute op "Standard.create" on "maven"... Done
✔ Execute op "Standard.start" on "maven"... Done
✔ Execute op "Standard.create" on "api"... Done
✔ Execute op "Standard.configure" on "api"... Done
✔ Execute op "Standard.start" on "api"... Done
✔ Execute op "Standard.create" on "gui"... Done
✔ Execute op "Standard.configure" on "gui"... Done
✔ Execute op "Standard.start" on "gui"... Done
```

Figure 9. The output displayed by TOSKER while executing the plan `thinking.up.plan` for deploying an instance of *Thinking*.

```
~ tosker ls
(update memory: ok)
Application    Component    Type        State    Full name
-------------  -----------  ---------   -------  -----------------
thinking       dbvolume     Volume      created  thinking.dbvolume
thinking       mongodb      Container   running  thinking.mongodb
thinking       node         Container   running  thinking.node
thinking       maven        Container   running  thinking.maven
thinking       api          Software    running  thinking.api
thinking       gui          Software    running  thinking.gui
```

Figure 10. The output of the TOSKER `ls` command after a successful execution of the plan `thinking.up.plan` for deploying an instance of *Thinking*.

The `ls` command then shows that the component `GUI` is deleted. □

To test the current prototype of TOSKER, we specified the open-source application *Thinking* in TOSCA, as well as three other existing applications, viz., (i) a Wordpress instance running on a PHP web server and connecting to a MySQL back-end, (ii) a NodeJS-based REST API connecting to a MongoDB back-end, and (iii) an application with three interacting servers written in NodeJS. All applications were effectively deployed by the current prototype of TOSKER, and they constituted the basis for developing a battery of unit tests**.

6.2. *How* TOSKER *works*

The prototype of TOSKER is composed by a main package (viz., `tosker`) containing a set of Python modules and packages implementing the various components forming the architecture of TOSKER (Fig. 11). The package `tosker` contains a set of modules and the subpackages `graph` and `managers`. The Python module `docker_interface.py` implements the architecture module Docker Interface, while `orchestrator.py`, `protocol_help.py`, `storage.py`, `tosca_parser.py`, and `ui.py` implement Orchestrator, Plan Checker, State Storage, TOSCA Parser and UI, respectively. The `graph` sub-package contains the modules that implement the internal representation of an application specification, viz., `artifacts.py`,

**The TOSCA application specifications and the battery of unit tests that we implemented are publicly available on GitHub at `https://github.com/di-unipi-socc/TosKer/tree/master/data/examples` and `https://github.com/di-unipi-socc/TosKer/tree/master/tests`, respectively.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

`nodes.py`, `protocol.py`, `relationships.py`, and `template.py`. Instead, the sub-package `managers` contains the modules `container_manager.py`, `volume_manager.py`, `software_manager.py` which respectively implements the three managers in the architecture of TOSKER (Fig. 7).

```
tosker
├── graph ..................................................... graph module
│   ├── artifacts.py ..................................... TOSCA artifacts
│   ├── nodes.py ..................................... TOSKER nodes
│   ├── protocols.py ..................................... management protocol
│   ├── relationships.py ..................................... TOSCA relationships
│   └── template.py ..................................... TOSCA topology template
├── managers ..................................... managers module
│   ├── container_manager.py ..................................... container manager
│   ├── software_manager.py ..................................... software manager
│   └── volume_manager.py ..................................... volume manager
├── docker_interface.py. ..................................... docker interface
├── orchestrator.py ..................................... orchestrator
├── protocol_helper.py ..................................... plan checker
├── storage.py ..................................... state storage
├── tosca_parser.py ..................................... TOSCA parser
└── ui.py ..................................... user interface
```

Figure 11. The code organisation in the current prototype of TOSKER.

The orchestration algorithm implemented in TOSKER supports the management protocols. This permits implementing ad-hoc management mechanisms for each component by defining a proper protocol. We indeed developed a default management protocol for each component (viz., Docker containers, Docker volumes and software components) to model their default behaviour, and we permit customising the behaviour of software components by means of policies. Default/customised protocol are then automatically composed by TOSKER, to derive the management behaviour of an application. We report below some details concerning the implementation of management protocols in TOSKER.

Management protocols are implemented in TOSKER as directed graphs (`graph/protocol.py`), whose nodes represent the states of a component, and whose edges represent the transitions among such states. TOSKER associates an instance of (a graph representing) a management protocol to each component of a topology, plus a pointer to the actual state of such component.

Management protocols are first parsed from the TOSCA specification, and then processed to manage the fact that the orchestration of software components strictly depends on the availability of their hosting containers. More precisely, two conditions must be enforced:

(i) A container must be *running* to permit executing a management operation of a software component hosted on such container, and

(ii) a container can be *deleted* only if all software components hosted on it are also *deleted* (otherwise, deleting a container would also force deleting of the software components hosted on it).

TOSKER automatically ensures both conditions by transparently extending the (default and customised) management protocols of Docker containers and software components as follows. To ensure (i), each transition of the protocols of software components is automatically enriched by constraining its firing to the satisfaction of the requirement *host*. As containers are satisfying such requirement only when *running*, this forces them to be in such state to allow executing management operations on the software components they host.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

To ensure (ii), TOSKER artificially adds a requirement *alive* to all software components, and a capability alive to all Docker containers and software components. The requirement *alive* of a software component is then bound to the capability *alive* of the node hosting such component. The purpose of requirements/capabilities *alive* is to allow to hosting nodes to witness whether they are still "alive" to the nodes they are hosting[††]. This is ensured by automatically extending the management protocols of Docker containers and software components as follows. The capability *alive* is offered by both Docker containers and software components in each state but the initial one (to witness to the node that they are "alive"). The requirement *alive* is assumed by each software component in each state but the initial one (to witness that they assume their container to be "alive" throughout their management lifecycle). In this way, a container can be *deleted* only if all the software component hosted on it are in their initial state (viz., *deleted*).

6.3. Concluding remarks

During the design and development of TOSKER we incurred in a set of problems that could be interesting also to other researchers and practitioners willing to exploit Docker container as lightweight virtual hosts for software components.

Docker containers are isolated processes whose lifetime is strictly coupled with that of the main command they actually execute. We can hence distinguish between containers whose lifetime is limited (because they run a main command whose execution time is limited), and containers that continue to run until a stop command is explicitly issued (because their main command is non-stopping throughout time - e.g., because it is running a service). This distinction was very important in our case, as we were using containers as virtual hosts, and we had to ensure that such hosts were continuing to run as long as the software components hosted on them were requiring it. In TOSKER, we solved this issue by allowing developers to indicate the type of artifacts they are using to implement a Docker container, viz., *tosker.artifacts.Image* and *tosker.artifacts.Dockerfile* for artifacts implementing containers whose lifetime is limited, and *tosker.artifacts.Image.Service* and *tosker.artifacts.Dockerfile.Service* for artifacts implementing containers that run persistently. This allows TOSKER to deal with containers whose lifetime is limited, by instructing them to execute a non-stopping command (viz., `sh -c "while true;do sleep 1;done"`), hence ensuring that they will run untile a command to stop them is explicitly issued. Instead, TOSKER does not change the main command of containers that persistently run, as this could impede them to offer the service they are running.

Another interesting issue was how to manage multiple software components hosted on the same container. Docker containers are static entities. They are instances of given images, which are read-only templates, and which are not thought to be changed at runtime (e.g., by installing, starting or uninstalling the software components run by corresponding containers). We decided to approach this issue by exploiting what is naturally supported by Docker, namely volumes and the command `docker exec`. TOSKER automatically mounts a dedicated volume on each container used as virtual host, and it places all artifacts implementing the management operations of the software components hosted by such container. This allows TOSKER to execute (the artifact implementing) a management operation of a software component by simply issuing a command `docker exec` on the container hosting the component (when such container is running). The latter is automatically executed by TOSKER whenever it is required to run of a management operation of an application component. TOSKER also automatically redirects the output of the execution of an operation to a log file contained in the automatically mounted volume, and it exploits such log to permit checking what happens when executing management operations on the components forming an application (through the command `tosker log`).

[††]The addition of requirement and capabilites *alive* was first proposed in [9] to address a similar problem.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

7. TOSKER AT WORK: THE *SOCK SHOP* CASE STUDY

Sock Shop [39] is an open-source web application that simulates the user-facing part of an e-commerce website that sells socks. It is a multi-component application intended to aid the demonstration and testing of solutions and tools supporting such a kind of applications. We hence exploited it to run a case study on our approach, which we present hereafter.

The *Sock Shop* application is composed by 14 components, the main ones being the following:

- A *Frontend* displaying a graphical user interfaces for e-shopping socks.
- A set of pairs of services and databases for storing and managing the catalogue of available socks (viz., *Catalogue* and *CatalogueDB*), the users of the application (viz., *Users* and *UsersDB*), the users' shopping carts (viz., *Carts* and *CartsDB*), and the users' orders (viz., *Orders* and *OrdersDB*).
- Two services (called *Payment* and *Shipping*) for simulating the payment and shipping of orders.

The *Sock Shop* application is then completed by three other components, namely *Edge Router*, *RabbitMQ* and *Queue Master*. The *Edge Router* redirects user requests to the *Frontend*. The *RabbitMQ* is a message queue that is filled of shipping requests by the *Shipping* service. The shipping requests are then consumed by the *Queue Master*, to simulate the actual shipping of orders.

We first analysed the implementation of *Sock Shop* in Docker compose, to understand the actual implementation of its components and how they communicate with each other. From the Docker compose file, we were only able to partition the Docker containers packaging the components of *Sock Shop* in two main classes, viz., the application-specific containers, each packaging a component implementing some business logic of the *Sock Shop* application (e.g., *Frontend*, *Catalogue*), and the general purpose containers, implementing a support infrastructure for *Sock Shop* (e.g., the databases, the edge router).

We then analysed the Dockerfile of the application-specific containers and we identified the list of instructions used to build and run the software components they are packaging. We extracted the sequence of commands allowing to install a software component within a container from those executed within such container during the building process (viz., from the arguments of the `RUN` commands of the corresponding Dockerfile). Similarly, we extracted the sequence of commands allowing to configure and/or run a software component from those those executed by the start-up command of its container (viz., from the arguments of the `CMD` command in the corresponding Dockerfile). Instead, we built the scripts to stop/delete a software component by manually determining the sequence of commands needed to revert the effects of those used to start/create it. (e.g., `pkill "/app/catalogue"` reverts the effects of `/app/catalogue -port=80`).

Additionally, none of the relationships occurring among the components of *Sock Shop* (viz., the components with which a component communicates) were described in the Docker compose file. We had to discover them at runtime, by running an instance of the *Sock Shop* application and by inspecting the communications occurring among the containers of such instance.

We then represented the *Sock Shop* application in TOSCA as displayed in Fig. 12. We modelled all databases and infrastructure components as nodes of type *tosker.nodes.Container*, and we exploited the Docker containers already configured by Weaveworks to actually implement them. We instead specified the services *Frontend*, *Catalogue*, *Users*, *Carts*, *Orders*, *Payment* and *Shipping* as nodes of type *tosker.nodes.Software*, each of which is hosted on a node of type *tosker.nodes.Container* providing the runtime environment it needs (e.g., as *Frontend* requires NodeJS, it is hosted on a container implemented with the Docker image `node:6`).

It is worth noting that the TOSCA-based representation of *Sock Shop* in Fig. 12 is not the only one possible. For instance, TOSKER can support also the specification where components are not decoupled from their Docker containers (in the same way as Docker compose does), or a specification where more components have been extracted and decoupled from their hosting

DOI: 10.1002/spe.2625

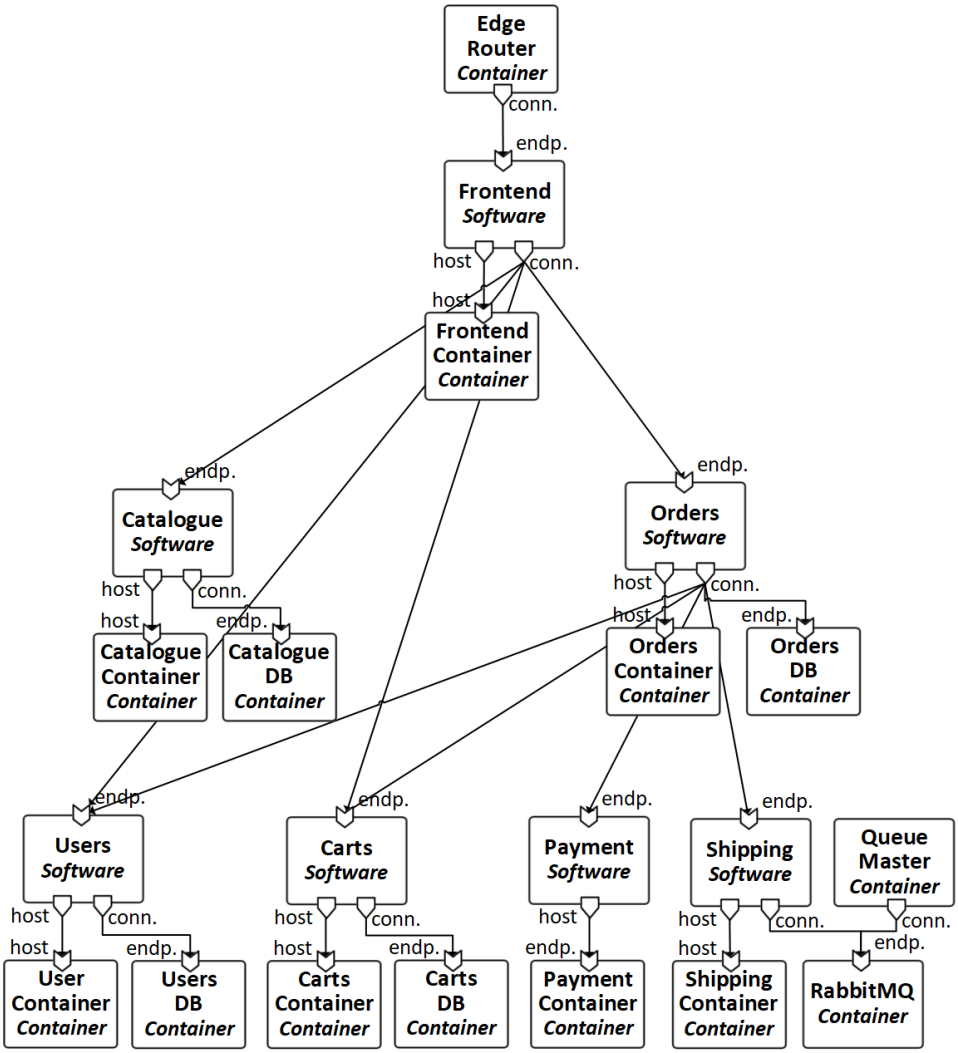

Figure 12. Our representation of *Sock Shop* in TOSCA, with the types defined in Sect. 4.

containers (e.g., the hosting stack of the *Frontend* may include three components, viz., the *Frontend*, which is hosted on a NodeJS runtime, which is in turn hosted on a container `ubuntu`).

Decoupling software components from the Docker containers hosting them allows TOSKER to independently manage them. Both software components and Docker containers can have their own requirements and capabilities, which will be interconnected to influence the order in which they have to be orchestrated. Additionally, decoupling of a software component from its hosting container permits easily changing the actual implementation of the Docker container without modifying the description nor the implementation of the software component [12]. Finally, by decoupling software components from their hosting containers, it is easier to customise the management lifecycle of such software components (e.g., by attaching them an ad-hoc management protocol).

It is also worth noting that in the TOSCA-based specification of *Sock Shop* all interconnections between its components are explicitly represented (as typed relationships). This is very useful not only for the orchestration per se, but also to implement statically/dynamically analysing the communications among components both at design-time and at run-time.

We implemented the operations to install, configure, start, stop and uninstall the above mentioned services each with a different shell script. For each service, the shell script `install.sh` installs the service in a dedicated folder of its host, by cloning the GitHub repository containing its

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

sources within such folder and by compiling (if needed) such sources. The script `configure.sh` configures the endpoints to be offered by the service. The scripts `start.sh` and `stop.sh` start and kill the process corresponding to the service, respectively. The script `uninstall.sh` deletes the folder containing the service installation.

We built a CSAR archive file (containing the TOSCA specification and the management scripts) and two management plans (viz., `sockshop.up.plan` to get the application running and `sockshop.down.plan` to tear it down)[‡‡].

We passed the obtained specification and a the `sockshop.up.plan` as input to TOSKER, by executing the following command:

```
$ tosker exec sockshop.csar --plan sockshop.up.plan
```

This effectively resulted in obtaining a running instance of the *Sock Shop* application. We then verified that the obtained instance of *Sock Shop* was working by successfully executed a set of tests orders (as suggested in [39]). We finally executed the plan `sockshop.down.plan`, which successfully resulted in undeploying the instance of *Sock Shop*.

8. RELATED WORK

In the following, we position TOSKER with respect to other currently available solutions for orchestrating the management of multi-component applications with Docker and/or TOSCA.

8.1. Docker-based orchestration

Docker natively supports multi-container Docker applications with Docker compose [20]. Docker compose permits specifying the (images of) containers forming an application, the virtual network to be set between such containers, and the volumes to be mounted. Based on that, Docker compose is capable of deploying the specified application. However, Docker compose treats containers as black-boxes, meaning that there is no information on which components are hosted by a container, and that it is not possible to orchestrate the management of application components separately from that of their containers. We showed a concrete example of this in Sect. 7, where we illustrated how Docker compose files do not allow to explicitly indicate the components forming an application (but only the black-box containers that package them), hence not allowing to represent the relationships occurring among such components. TOSKER instead permits modelling and orchestrating the software components independently from the container that host them, and to explicitly indicate the different types of relationships occurring among such components. This not only makes the interactions occurring among the components of an application easier to understand, but also brings various advantages (at the price of requiring developers to specify more information), e.g., simplifying the update of the container used to host a component, as we illustrated in [12].

Other approaches worth mentioning are Docker swarm [22], Kubernetes [38], and Mesos [3]. Docker swarm permits creating a cluster of replicas of a Docker container, and seamlessly managing it on a cluster of hosts. Kubernetes and Mesos instead permit automating the deployment, scaling, and management of containerised applications over clusters of hosts. Docker swarm, Kubernetes and Mesos differ from TOSKER as they focus on how to schedule and manage containers on clusters of hosts, rather than on how to orchestrate the management of the components and containers forming multi-component applications.

Similar considerations apply to ContainerCloudSim [35], which provides support for modelling and simulating containerised computing environments. ContainerCloudSim is based on CloudSim [15], and it focuses on evaluating resource management techniques, such as container scheduling, placement and consolidation of containers in a data center, by abstracting from the application components actually running in such containers. Our solution instead focuses on

[‡‡]The TOSCA specification, the scripts, the CSAR file and the plans are all avaible at `https://github.com/di-unipi-socc/TosKer/tree/master/data/examples/sockshop-app`.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

allowing to independently orchestrate the components forming an application and the containers used to host them.

8.2. TOSCA-based orchestration

A first attempt to synergically combine TOSCA and Docker has been proposed in [28], which presents an approach for using TOSCA for specifying the internals of a container. [28] then requires the user to manually build containers, which can then be orchestrated (as black-boxes) on Mesos. TOSKER instead permits orchestrating software components directly and independently from their hosting containers, as it automatically places a software component within its container and since it executes management operations directly on a component.

Other approaches worth mentioning are OpenTOSCA [6] SeaClouds [10], Brooklyn [2], Alien-4Cloud [24], Cloudify [26], and ARIA TOSCA [4]. OpenTOSCA [6] is an open-source engine for deploying and managing TOSCA applications. It is designed to work with a former, XML-based version of TOSCA [30] and it supports the orchestration of Docker containers as black-boxes (viz., it does not permit orchestrating application components independently of the containers hosting them). TOSKER differs from OpenTOSCA since it works with the newer, YAML-based version of TOSCA [32], and since it permits orchestrating the software components of an application separately from the Docker containers used to host them.

SeaClouds [10] is a middleware solution for deploying and managing multi-component applications on heterogenous IaaS/PaaS clouds. SeaClouds fully supports TOSCA, but it lacks a support for Docker containers. The latter makes SeaClouds not suitable to orchestrate the management of multi-component applications including Docker containers.

Brooklyn [2] instead natively supports Docker containers. Thanks to its extension called "Brooklyn-TOSCA" [16], Brooklyn enables the orchestration of the management of the software components and Docker containers forming cloud application. However, Brooklyn treats Docker containers as black-boxes, and this does not permit orchestrating the management of the components of an application independently of that of the containers used to host them.

Alien4Cloud [24], Cloudify [26], and ARIA TOSCA [4] also permit orchestrating the management of the software components and Docker containers forming cloud applications. They however all differ from TOSKER because they treat Docker containers as black-boxes (hence not permitting to orchestrate the management of application components separately from that of the containers hosting them).

Brooklyn [2] and Cloudify [26] also differ from TOSKER as they require to specify applications in non-standard blueprint languages (inspired to, but not fully compliant with, the OASIS standards CAMP [31] and TOSCA [31], respectively). For instance, a relationship is specified in TOSCA by connecting a requirement of one component to a capability of another, and requirements/capabilities can be used to express interconnection constraints (which then permit validating TOSCA application topologies [11]). Cloudify blueprints instead do not include any notion of requirements or capabilities, as relationships just connect a source node to a target node.

Additionally, all the above mentioned approaches differ from TOSKER because they do not provide a way to customise the management behaviour of the components forming a TOSCA application (e.g., even if some management operation can be added to a TOSCA node type, it is not possible to declaratively indicate when/how to invoke it throughout the management lifecycle of instances of such type). TOSKER instead permits customising the management behaviour of the components of a TOSCA application, by supporting the definition of management protocols that indicate how to orchestrate the components' management operations.

8.3. Summary

To the best of our knowledge, ours is the first solution that permits specifying and orchestrating multi-component applications with TOSCA and Docker both (i) by allowing to customise the management behaviour of the software components forming an application, and (ii) by managing such components independently of the containers hosting them.

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

9. CONCLUSIONS

The OASIS standard TOSCA [32] and the Docker ecosystem [19] are two emerging solutions trying to address, from different perspectives, the problem of automating the management of multi-component applications over heterogeneous cloud infrastructures. While TOSCA follows a model-driven approach, Docker follows a "snapshot-based" approach. Of course, both approaches have their own pros and cons, and the objective of this paper was to propose a solution trying to synergically combine the pros of both approaches.

We indeed illustrated a way to employ TOSCA and Docker together to orchestrate the management of multi-component applications. More precisely, we proposed a TOSCA-based representation for multi-component applications, which permits (i) distinguishing the Docker containers and software components in a multi-component application, (ii) indicating the relationships occurring among them, and (iii) customising their management behaviour through management protocols [7]. We also presented TOSKER, an orchestration engine for automatically deploying and managing multi-component applications based on TOSCA and Docker. We finally showed how our approach can be used to fruitfully orchestrate the management of multi-component applications by means of a concrete case study (based on a third-party application).

The current prototype of TOSKER can already be exploited (as is) by researchers and practitioners to orchestrate multi-component applications. Such prototype, as well as the proposed TOSCA-based type system, can also be extended to support the orchestration of other types of components. The latter can be done by defining the TOSCA types for representing such components, and by plugging into the modular architecture of TOSKER the corresponding type managers.

The current prototype of TOSKER can also be exploited as the foundations for the development of other research solutions or tools, as we did in [12], for instance. [12] presents TOSKERISER, a tool that is capable of automatically determining the Docker containers used to host the components forming an application. With TOSKERISER, developers can only specify the software components forming an application, each together with the software distributions it needs to run. Obtained (incomplete) specifications can then be passed to TOSKERISER, which interacts with DOCKERANALYSER [13] to search for Docker containers, and which automatically completes them with Docker containers offering the software support needed by the application components. Completed specification can then be effectively orchestrated by TOSKER.

It is worth nothing that the current prototype of TOSKER is intended to be a demonstrator of a research result, namely to prototype a solution that tries to synergically combine the pros of both TOSCA and Docker. For this reason, TOSKER it is not yet ready for production environments, as it will need further engineering of the tool, as well as further validation and test of its functionalities. Such engineering, validation and test activities are left for future work.

It is also worth noting that the current prototype of TOSKER interacts with the Docker engine installed on a host to actually enact the deployment and orchestration of multi-component applications. This means that all the containers running the components of an application reside on the same host. As part of our future work, we plan to allow TOSKER to deploy and orchestrate the multi-component application over clusters of hosts. Such an extension can be implemented by allowing multiple interacting instances of TOSKER to run on different hosts, in a "master-slave" manner (similarly to Docker Swarm [22], for instance). Master instances of TOSKER coordinate and schedule the deployment of the components of an application over a set of hosts, where slave instances are running. The latters receive commands from the master instances, and they enact the concrete management operations by interacting with the infrastructure where they are running. The above described distributed version of TOSKER can concretely be obtained and deployed on top of existing cluster management systems (e.g., Kubernetes [38] or Mesos [3]), or on more advanced container-based orchestration solutions (e.g., Fargate [1]).

Furthermore, the current prototype of TOSKER permits orchestrating the management of multi-component applications through already developed management plans. The integration of TOSKER with an existing solution for automatically determining management plans reaching given management goals (e.g., [8], [9] or [18]) is also in the scope of our future work. This would indeed

Submitted draft. Prepared using speauth.cls
DOI: 10.1002/spe.2625

allow providing TOSKER only with the desired configuration of a given application, and it would be TOSKER that automatically determines the plan reaching such configuration.

ACKNOWLEDGMENTS

The authors would like to thank Claus Pahl for all helpful and stimulating discussions on how to enhance the current support for orchestrating multi-component applications with TOSCA and Docker.

REFERENCES

1. Amazon. Fargate. `https://aws.amazon.com/fargate/`. last accessed: May 28th, 2018.
2. Apache Software Foundation. Brooklyn. `https://brooklyn.apache.org`. last accessed: May 28th, 2018.
3. Apache Software Foundation. Mesos. `https://mesos.apache.org/`. last accessed: May 28th, 2018.
4. Apache Software Foundation Incubator. ARIA TOSCA. `http://ariatosca.incubator.apache.org/`. last accessed: May 28th, 2018.
5. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4):50–58, 2010.
6. Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA – a runtime for TOSCA-based cloud applications. In Samik Basu, Cesare Pautasso, Liang Zhang, and Xiang Fu, editors, *Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings*, pages 692–695, Berlin, Heidelberg, 2013. Springer.
7. Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling and analysing cloud application management. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service-Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings*, pages 19–33. Springer, 2015.
8. Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware application management protocols. In Marco Aiello, Broch Einar Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, pages 219–234. Springer, 2016.
9. Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware management protocols for multi-component applications. *Journal of Systems and Software*, 139:189 – 210, 2018.
10. Antonio Brogi, José Carrasco, Javier Cubo, Francesco D'Andria, Ahmad Ibrahim, Ernesto Pimentel, and Jacopo Soldani. EU Project SeaClouds - adaptive management of service-based applications across multiple clouds. In *Proceedings of the 4th International Conference on Cloud Computing and Services Science (CLOSER 2014)*, pages 758–763, 2014.
11. Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani. Validating TOSCA application topologies. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 667–678. SciTePress, 2017.
12. Antonio Brogi, Davide Neri, Luca Rinaldi, and Jacopo Soldani. From (incomplete) tosca specs to running apps, with docker. In *Software Engineering and Formal Methods - SEFM 2017 Collocated Workshops, Trento, Italy, September 4-5, 2017, Revised Selected Papers*, Lecture Notes in Computer Science. Springer, 2017. *[To appear]*.
13. Antonio Brogi, Davide Neri, and Jacopo Soldani. A microservice-based architecture for (customisable) analyses of docker images. *Software: Practice and Experience*, 2018. *[In press]*.
14. Antonio Brogi, Luca Rinaldi, and Jacopo Soldani. TosKer: Orchestrating applications with TOSCA and Docker. In Zoltán A. Mann and Volker Stolz, editors, *Advances in Service-Oriented and Cloud Computing. ESOCC 2017*, volume 824 of *Communications in Computer and Information Science*. Springer, 2017.
15. Rodrigo N. Calheiros, Ranjan Rajiv, Beloglazov Anton, Csar A. F. De Rose, and Rajkumar Buyya. CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2010.
16. José Carrasco, Javier Cubo, Francisco Durán, and Ernesto Pimentel. Bidimensional cross-cloud management with TOSCA and brooklyn. In *9th IEEE International Conference on Cloud Computing, CLOUD 2016, San Francisco, CA, USA, June 27 - July 2, 2016*, pages 951–955, 2016.
17. Muhammad Aufeef Chauhan, Muhammad Ali Babar, and Boualem Benatallah. Architecting cloud-enabled systems: a systematic survey of challenges and solutions. *Software: Practice and Experience*, 47(4):599–644, 2017.
18. Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Information and Computation*, 239(0):100 – 121, 2014.
19. Docker Inc. Docker. `https://www.docker.com/`. last accessed: May 28th, 2018.
20. Docker Inc. Docker compose. `https://docs.docker.com/compose/`. last accessed: May 28th, 2018.
21. Docker Inc. Docker hub. `https://hub.docker.com/`. last accessed: May 28th, 2018.
22. Docker Inc. Docker swarm. `https://docs.docker.com/engine/swarm/`. last accessed: May 28th, 2018.
23. Christian Endres, Uwe Breitenbücher, Michael Falkenthal, Oliver Kopp, Frank Leymann, and Johannes Wettinger. Declarative vs. Imperative: Two Modeling Patterns for the Automated Deployment of Applications. In *Proceedings*

DOI: 10.1002/spe.2625

of the 9th International Conference on Pervasive Patterns and Applications, pages 22–27. Xpert Publishing Services (XPS), 2017.
24. FastConnect, Bull, and Atos. Alien4cloud. `https://alien4cloud.github.io/`. last accessed: May 28th, 2018.
25. Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.
26. GigaSpaces Technologies. Cloudify. `http://cloudify.co/`. last accessed: May 28th, 2018.
27. Pooyan Jamshidi, Claus Pahl, and Nabor C. Mendonca. Pattern-based multi-cloud architecture migration. *Software: Practice and Experience*, 47(9):1159–1184, 2016.
28. Stefan Kehrer and Wolfgang Blochinger. TOSCA-based container orchestration on mesos. *Computer Science - Research and Development*, 2017.
29. Karl Matthias and Sean P. Kane. *Docker: Up and Running*. O'Reilly Media, 2015.
30. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA), Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`, 2013.
31. OASIS. Cloud Application Management for Platforms (CAMP), Version 1.1. `http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.pdf`, 2016.
32. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Simple Profile in YAML, Version 1.0. `http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf`, 2016.
33. Claus Pahl. Containerization and the paas cloud. *IEEE Cloud Computing*, 2(3):24–31, 2015.
34. Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*. `https://doi.org/10.1109/TCC.2017.2702586`, *[In press]*.
35. Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N. Calheiros, and Rajkumar Buyya. ContainerCloudSim: An environment for modeling and simulation of containers in cloud data centers. *Software: Practice and Experience*, 47(4):505–521, 2016.
36. Randall Smith. *Docker Orchestration*. Packt Publishing, 2017.
37. Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. *SIGOPS Oper. Syst. Rev.*, 41(3):275–287, 2007.
38. The Kubernetes Authors. Kubernetes. `https://kubernetes.io/`. last accessed: May 28th, 2018.
39. Weaveworks Inc. and Container Solutions Inc. Sock Shop. `https://microservices-demo.github.io/`. last accessed: May 28th, 2018.

DOI: 10.1002/spe.2625