Deployment over Heterogeneous Clouds with TOSCA and CAMP

Jose Carrasco, Javier Cubo, Ernesto Pimentel and Francisco Durán
Dept. Lenguajes y Ciencias de la Computación, Universidad de Málaga, Málaga, Spain

Keywords: Cloud Applications, Cross-cloud, Cross-deployment, Standards, TOSCA, CAMP.

Abstract: Cloud Computing providers offer diverse services and capabilities, which can be used by end-users to compose heterogeneous contexts of multiple cloud platforms to deploy their applications, in accordance with the best offered capabilities. However, this is an ideal scenario, since cloud platforms are being conducted in an isolated way by presenting interoperability and portability restrictions. Each provider defines its own API, non-functional requirements, QoS, add-ons, etc., and developers are often locked-in a concrete cloud environment, hampering the integration of heterogeneous provider services to achieve cross-deployment. This work presents an approach to deploy cross-cloud applications by using standardisation efforts of design, management and deployment of cloud applications. Specifically, using mechanisms specified by the TOSCA and CAMP standards, we propose a methodology to describe the topology and distribution of modules of a cloud application and to deploy the inter-connected modules over heterogeneous clouds. We present our prototype TOMAT, which supports the automatic distribution of cloud applications over multiple providers.

1 INTRODUCTION

Cloud Computing (Armbrust et al., 2010) is the result of the evolution of different paradigms and technologies, such as Virtualisation, Client-Server Model, Peer-to-Peer, or Grid-Computing. It has become a key component in the new Internet, by establishing a model for business services that allows a suitable and on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned (Mell and Grance, 2011). In this model, providers expose theirs resources as services through a cloud infrastructure. Hence, many providers are offering cloud-oriented services, from big companies such as Amazon, Google, or Microsoft, to small players. Then, developers may use these cloud resources to host and deploy the infrastructure and application modules in specific providers, taking advantage of the elasticity and scalability of the clouds. In fact, in an ideal scenario, users could select the services from several providers whose properties and capabilities best fit the requirements of every module of the application, by enabling a flexible and multiple deployment, as addressed in, e.g., (Brogi et al., 2014).

Cross-cloud application deployment is a complex task, since cloud platforms are being conducted in an isolated way, presenting many interoperability and portability restrictions, and offering similar resources differently. Providers define their own APIs to expose services, non-functional requirements, QoS, add-ons, etc. As a result, cloud developers are often locked-in specific services from some concrete cloud environment, complicating the integration of heterogeneous provider services to achieve a cross-deployment of cloud applications (Petcu, 2011), and making almost impossible to consider the possibility of migrating components between different platforms.

Organizations like IEEE and OASIS have proposed different approaches to mitigate these issues through the homogenization and normalization of the description and management of cloud applications. We find of particular interest two OASIS standards, namely TOSCA (Topology and Orchestration Specification for Cloud Applications) (OASIS, 2012b) and CAMP (Cloud Application Management for Platforms) (OASIS, 2012a). These standards define methodologies to describe and wrap the structure of cloud applications (components and relationships), and how they must be orchestrated (using plans) in a portable way to increase a vendor-neutral ecosystem. However, although they describe the mechanisms that must be implemented by the clouds to support standard-based application deployment and management, they do not have official implementations. Partial support for them is available through, e.g., OpenTOSCA (Binz et al., 2013) and Alien4Cloud[1]

[1] Alien4Cloud: http://alien4cloud.github.io/.

for TOSCA, and Apache Brooklyn[2] for CAMP.

TOSCA is a good option for representing the topology and orchestration of applications. However, the management of a possible TOSCA-compliant deployment could be a complex task. Since TOSCA does not enforce specific interfaces for node templates, which represent the provider's resources, the topology and orchestration plan may need to be modified when some module of the application is decided to be deployed on a different target provider.

Even if considering what the standard calls *declarative plans*, as we will see in Section 2.1, the TOSCA specification of the topology requires using node templates for specific providers, what prevents the specification from being agnostic. Moreover, although by using appropriate node templates we may specify and deploy cross-cloud TOSCA configurations, we need to take into account how providers services are managed by TOSCA, and how components are deployed and their interactions handled. OpenTOSCA requires each operation in the node types modeling provider resources to include the necessary mechanisms. Alien4Cloud uses Cloudify[3] as cloud service orchestrator for the providers interaction, what allows using a generic node type to model cloud services. However, the implementation artefacts corresponding to the operations of the node templates must be adapted to work with such an orchestrator. Although with Alien4Cloud the definition of node types is much simpler, the set of target providers is limited to those supported by the orchestrator.

We are interested in application models in which we do not need to lock to a specific vendor, and want to be able to postpone our deployment decision as much as possible, even opening the door to potential migration of individual components at runtime. Although cross-deployment was not one of the goals of the existing CAMP-compliant solutions, they follow a unified API which wraps the interface of cloud providers (e.g., Brooklyn uses Apache jClouds[4]). Nevertheless, CAMP lacks of a topology specification, which is crucial to maintain the application model at run-time, key if monitoring and reconfiguration actions are to be performed over the application distribution.

Although both standards present weaknesses, we believe they nicely complement each other, and can be used together in real scenarios. We propose the combined use of TOSCA and CAMP specifications and their respective approaches by using a transformation-based platform-agnostic model. Our solution automatizes the translation between both standards by using a transformation model, providing structural and design advantages for the topology description using (agnostic) TOSCA, and benefits of the run-time management using CAMP. Specifically, we combine the advantages of TOSCA's powerful modeling language to describe the structure of an application as a typed topology graph, in a portable and vendor-agnostic way, and simplifying the management and reusability of services thanks to its topology templates and node types, with CAMP's facilities for managing heterogeneous providers's features in a homogeneous way thanks to its approach for the unification in the interfaces of cloud platforms. Preliminary results related to our proposal were presented in (Carrasco et al., 2015).

The rest of this paper is structured as follows. Section 2 presents a brief introduction to TOSCA and CAMP. Section 3 presents the proposal, using examples to illustrate the approach. We wrap up in Section 4 by presenting some conclusions and some ideas for future development.

[2]Brooklyn: https://brooklyn.apache.org/.

[3]Cloudify: http://getcloudify.org/.

[4]jClouds: http://jclouds.apache.org/.

2 SOME BACKGROUND

In this section, we briefly present the TOSCA and CAMP standards, mainly focusing on their strengths and limitations.

2.1 The TOSCA Standard

The TOSCA standard (OASIS, 2012b) provides a model for the description of cloud applications, the corresponding services, and their relationships using a service topology, as well as the description of procedures to manage services using orchestration processes by using plans.

As depicted in Figure 1, in TOSCA, services are specified through *service templates*, by using node and relationship templates, which are concrete components of the application of corresponding node and relationship types. A *Node Type* expresses the capabilities, requirements, properties and interfaces of the services. *Relationship Types* are used to specify relationships between the elements of the system by means of a set of relations, which allow determining the connections and dependencies between components, taking into account their properties and interfaces. Type definitions include operations to describe their behavior, e.g., a server may define a start operation to initialize the execution, and another one to allow the deployment of applications inside itself.

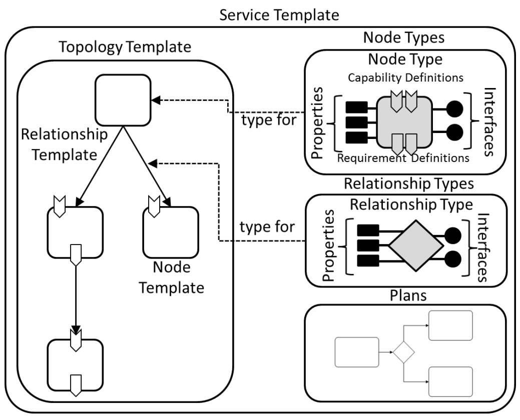


Figure 1: TOSCA Service Template elements.

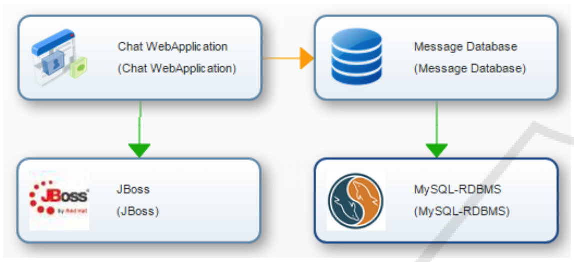


Figure 2: Chat App TOSCA topology (using Winery).

```
<NodeTemplate id="JBossMainWebServer"
  name="JBoss␣Main␣Web␣Server"
  type="JBossWebServer">
 <Properties>
  <ns2:JBossWebServerProperties>
   <httpPort>80</httpPort>
   </ns2:JBossWebServerProperties>
 </Properties>
 <Capabilities>
  <Capability
   id="JBossMainWebServer_webapps"
   name="webapps"
   type="JBossWebAppContainerCapability"/>
 </Capabilities>
</NodeTemplate>
```

Listing 1: JBoss node template.

Then, templates specify the concrete features of components, e.g., their properties and restrictions.

Figure 2 shows the TOSCA topology, using Winery (Kopp et al., 2013), for a very simple Chat Application composed by two modules, namely a webpage and a database. We use this example throughout our paper to illustrate the different aspects of our approach. In the example, we suppose we will deploy the webpage on Amazon AWS and the database on SoftLayer. The code in Listing 1 shows the XML TOSCA description of the node template for the webpage corresponding to a JBoss component.

In TOSCA templates, we can also specify orchestration *plans*. The plan specifies the operation calls of components to instantiate the structure of the topology. In the TOSCA standard, the values for the properties of node and relationship templates in a topology template can come from input passed in by users, by automated operations, or by specified default values.

A TOSCA-based description is packaged in a container file, called a CSAR (Cloud Service Archive). CSARs have a fix structure to allow portability between providers. To establish a homogeneous environment of clouds for deployment of CSARs, TOSCA defines a set of best practices to adapt the standard to services offered by cloud providers

2.2 The CAMP Standard

The CAMP standard (OASIS, 2012a) focuses on the deployment, management and monitoring of cloud applications, regardless of the used platform. Its main purpose is to allow the interoperability of interfaces of cloud platforms defining a unified API to be implemented by each platform supporting the standard. In such a way, developers and independent platforms could create services and mechanisms to interact with other platforms by adapting the CAMP specification and using their interfaces.

With its API, CAMP describes elements to model applications and the services and platforms used for deployment, providing the basis to mitigate the portability/interoperability problems. CAMP specifies models for platform, resources, services, sensors and operations. The *platform* represents the layer at the top of the platform under execution, and references the resources that represent the functionality provided by the platform, the applications running on this platform, and the metadata that describe the resources supported by the platform and the extensions by the provider. *Resources* can be implemented by means of the API provided by the cloud platform with different concepts (assemblies or components). *Services* are used to interact with the platform, representing the creation and management of resources (e.g., deploy a Web application). Following the goal of facilitating the management and monitoring of deployed systems, CAMP defines *sensors*, which through a RESTful interface give access to different metrics on the status of the application components. *Operations* represent actions that can be taken on resources. By combining sensors and operations, CAMP proposes the use of *policies* for self-management: operations depending on the information collected by sensors.

As TOSCA, CAMP defines a packaging artifact for application models and elements for the deployment called PDP (Platform Deployment Package).

```
1  services:
2  - serviceType: org.apache.brooklyn.entity.webapp.jboss.
       JBoss7Server
3    name: JBoss Main Web Server
4    location: aws-ec2:us-west-2
5    id: JBossMainWebServer
6    brooklyn.config:
7      port.http: 80+
8      port.https:3344+
```

Listing 2: Brooklyn description of a JBoss Server.

An application PDP contains its deployment plan in a YAML file, platform elements used during the life-cycle, and credential files. This description is done over CAMP's API to ensure portability.

Although there is no official CAMP implementation, Apache Brooklyn[2] supports almost all features and concepts in CAMP. Brooklyn is a framework for modeling, managing and monitoring applications, which offers a unified API, and allows the modeling of application modules and its infrastructure for deployment and monitoring. Brooklyn's API is built over jClouds, an open source cross-cloud toolkit to create platform-independent applications, which ensures the interoperability and portability with more than 30 different providers.

Brooklyn provides support for deploying applications, once the deployment location of each application module has been specified, providing support for establishing and maintaining the relationships between the different modules. This facility may be used for performing heterogeneous deployment, using services of multiple providers. We have indeed used this facility to partially solving the portability and interoperability problems between services, and to facilitate the complicated deployment in TOSCA. Listing 2 shows the Brooklyn YAML excerpt describing a JBoss server for the webpage (corresponding to the TOSCA template in Listing 1) using a common API. As specified in Line 4, the component is requested to be deployed on aws-ec2:us-west-2, Amazon's datacenter at Oregon, US.

3 CROSS-DEPLOYMENT OF CLOUD APPLICATIONS

In this section, we present our proposal for improving interoperability and reducing portability problems of heterogeneous clouds.

3.1 Overview of our Approach

The main goals and advantages of our approach are:

Topology Description. We consider applications composed of several modules or components, with relationships and dependencies between them. We propose using TOSCA for the specification of the application's topology and its distribution. This allows us maintaining the application structure independently of the cloud services used along the life-cycle of the application.

Portability and Interoperability. Dealing with a multiple deployment and heterogeneous distribution, portability and interoperability problems occur due to restrictions imposed by used services. Developers may need to adapt their systems (applications, services, components) to the interfaces of the specific cloud providers used for deployment. By following CAMP's philosophy, we mitigate this dependency through a common API wrapping and unifying cloud service features and requirements offered by a large number of diverse providers. Heterogeneity will be directly dealt with by managing cloud services in a homogeneous way.

Scalability and Elasticity. Our approach considers the scalability and elasticity benefits of multiple providers by using the most appropriate cloud to deploy each application module, and giving access to the facilities provided by each of them.

Figure 3 presents the overview of our approach to cross-deployment through the combined use of the TOSCA and CAMP standards. The proposed methodology consists of two main phases. In the first phase, *Topology Description*, the topology of an application is exhaustively defined using TOSCA (1). In the second phase, *Translation and Deployment*, the application modules are distributed over selected cloud providers using the CAMP-based deployment strategy implemented in Brooklyn. To solve the incompatibility problems between the TOSCA-based XML topology and the CAMP-based YAML deployment plan, we have modeled and developed a translation process based on an agnostic graph (2). As explained in the coming sections, the graph is built on an abstraction of the application topology, providing a representation independent of the specific used deployer, Brooklyn in this case. The use of these graphs also simplifies the translation process from TOSCA to CAMP, which finally generates the CAMP-compliant deployment plan expected by Brooklyn (3). Brooklyn is then in charge of deploying the given application, using jClouds to enable provider-independent deployment, and cross-deployment if required.

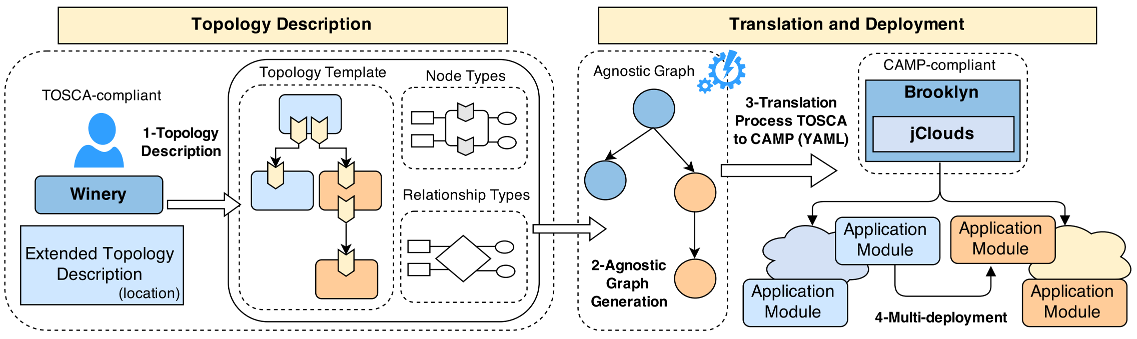


Figure 3: Overview of the methodology for cross-deployment.

3.2 TOSCA-compliant Topology

We use TOSCA to obtain a full description of the application by modeling its modules and dependencies. TOSCA descriptions are composed of components, which present types, properties, requirements, capabilities, and relationships between them.

TOSCA allows the specification of a detailed application topology using XML files. Although it has the advantage of providing the flexibility to design any component that an application may require, the modification, removal or addition of components in a topology may become a cumbersome task. We propose the use of graphical tools such as Winery (Kopp et al., 2013), from the OpenTOSCA environment, for these tasks. Winery provides support for the graphical generation of TOSCA-based application topologies and their maintenance.

For describing a topology, TOSCA does not provide any mechanism to specify the target provider to deploy application modules. Instead, NodeTypes define an orchestration plan and the implementation artifacts to specify the deployment operations. These artifacts define the specific distribution of the components implementing the logic to perform the deployment.

We described on Section 1 as the providers to be used to deploy our components on can be defined and replaced on some TOSCA-compliant approaches. However, the definition and maintenance of an orchestration plan is a complex and error-prone task. TOSCA imperative plans have to define each necessary step to deploy and configure the application, taking into account all the properties and requirements of the providers. TOSCA declarative plans need to add new implementation artifacts to modify the the operations of node templates. Thus, if we need to change the cloud providers on which modules are to be deployed, the plan's operations should be accordingly modified, since the modification of the providers is performed by substituting the artifacts that implement the deployment operations.

Our proposal does not need to run these artifacts though, and therefore, we do not need the required robust logic to infer the distribution information from them. But we instead need a mechanism to clearly specify the final provider each of the node templates is to be deployed on.

TOSCA allows the specification of properties to specify the way in which applications are going to behave, e.g., to achieve specific service level objectives. Indeed, the possibility of adding further properties to node templates is one of the main arguments of TOSCA's flexibility. We take advantage of this possibility and specify the location to which node templates are to be deployed as properties of node templates. By defining these properties, we get all the information we need to generate the corresponding CAMP specifications. Note that this flexibility, however, involves an even higher complexity in the TOSCA orchestration plan, since the plan must specify the management of all the properties according to the system configuration.

Listing 3 shows how the TOSCA XML specification of the JBoss node template in Listing 1 has been extended to specify the location to be deployed at (Line 7). With this specification, it will be deployed on Amazon AWS, using Oregon's clusters. A description as the one shown in Listing 2 can now be automatically generated.

3.3 TOSCA2CAMP Transformation

CAMP provides support for the deployment over heterogeneous clouds, solving interface incompatibilities through a unified API. By performing a transformation from TOSCA to CAMP elements, we take advantage of this capability to deploy TOSCA specifications over multiple cloud providers.

```
1   <NodeTemplate id="JBossMainWebServer"
2     name="JBoss␣Main␣Web␣Server"
3     type="JBossWebServer">
4    <Properties>
5     <ns2:JBossWebServerProperties>
6      <httpPort>80</httpPort>
7      <location>aws-ec2:us-west-2</location>
8     </ns2:JBossWebServerProperties>
9    </Properties>
10   <Capabilities>
11    <Capability
12     id="JBossMainWebServer_webapps"
13     name="webapps"
14     type="JBossWebAppContainerCapability" />
15   </Capabilities>
16  </NodeTemplate>
```

Listing 3: Node Template enriched with *Location*.

```
1   - service: agnostic.service.type.server.JBossServer
2     name: JBoss Main Web Server
3     location: aws-ec2:us-west-2
4     id: JBossMainWebServer
5     properties:
6       - id: http
7         type: port
8         value: 80
9       - id: https
10        type: port
11        value: 3344
```

Listing 4: Agnostic description of a JBoss server.

3.3.1 Agnostic Graph Generation

In the translation process, we use an intermediate agnostic graph to abstract system information to get a technology-independent intermediate representation of applications. This representation is independent both from TOSCA and CAMP description types, what would allow us to reuse the transformation to such an agnostic graph or the one from it, e.g., to target CAMP providers other than Brooklyn.

The topology of an application is abstracted as depicted in step (2) in Figure 3. The information in each component is included as a node of the graph, without reference to TOSCA elements, and using the edges to specify the relationships between them.

Listing 4 shows a textual representation of the agnostic graph as obtained from the JBoss node template in Listing 3.

3.3.2 Types and Properties

Service types of agnostic components are represented by technology-independent agnostic identifiers. E.g., agnostic.service.type.server.JBossServer is used to type a JBoss server in the definition in Listing 4. Similar mappings for each concrete TOSCA type into its corresponding agnostic type provides the necessary transformation rules.

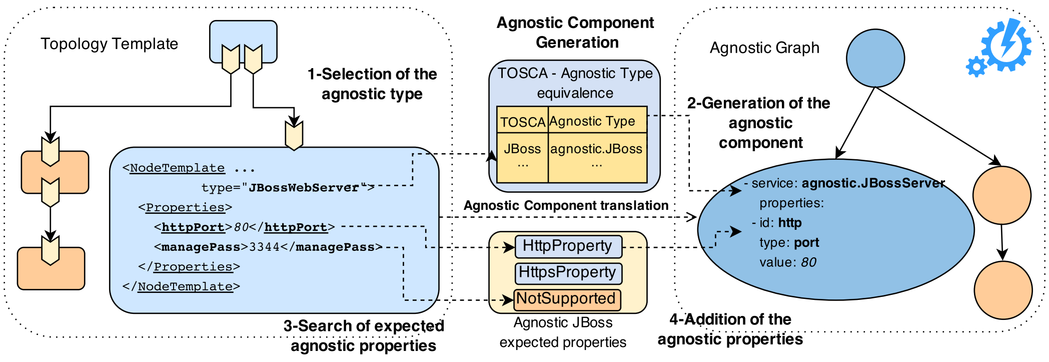
One important aspect in the correlation between TOSCA and agnostic components is the management of properties (see Section 3.2). By using agnostic elements we simplify the generation of CAMP elements to be used in the deployment, unbinding the description of TOSCA elements to the heterogeneous APIs used by CAMP implementations. Since each service or resource in the Brooklyn API defines a specific set of properties, for the configuration of such components, it is required to establish a relationship between the TOSCA properties defining the topology, which are not subject to constraints, and CAMP properties, which are predefined. E.g., a JBossWebServer in the Brooklyn API allows defining the port http, so it is necessary to find, in the corresponding TOSCA element definition, the property which specifies that value. Note that this property could be defined in different ways in the node template.

CAMP API allows a more detailed configuration based on the properties of theirs elements. E.g., the port http of a JBoss server in TOSCA can be defined by means of a plain property (a string), as a TOSCA property <httpPort>80</httpPort>. However, the equivalent CAMP property could add more information to define a port range, with, e.g., port.http: 80+, where the symbol '+' indicates that the http port is a value equal or greater than 80. See the agnostic representation of the properties http and https in Listing 4 (Lines 6-11) as obtained from the property port in the TOSCA description of the JBoss component in Listing 3 (Line 6).

This mechanism not only establishes the equivalence between properties, but also infers and maintains the necessary knowledge about them. Only in this way we can adapt them to the configuration required by the services described in CAMP APIs. We therefore need to fix the properties permitted in node templates, which will be interpreted and added to the agnostic elements using types.

During the CAMP deployment plan generation, the process uses the types of these properties to correctly configure the components described in the corresponding CAMP APIs. For example, since Brooklyn knows that http is a property of the port type, to specify this property, we need to add the symbol '+' to determine the port range.

Figure 4 refines Figure 3 to summarize the generation of a JBoss agnostic component. Steps (1) and (2) represent the translation of the TOSCA JBoss component into the agnostic one. Step (3) checks whether the properties of the node template are permitted in the properties list expected by the agnostic element. If


Topology Template
1-Selection of the agnostic type
Agnostic Component Generation
TOSCA - Agnostic Type equivalence
TOSCA | Agnostic Type
JBoss ... | agnostic.JBoss ...
Agnostic Component translation
HttpProperty
HttpsProperty
NotSupported
Agnostic JBoss expected properties
3-Search of expected agnostic properties
Agnostic Graph
2-Generation of the agnostic component
4-Addition of the agnostic properties

```
<NodeTemplate ...
         type="JBossWebServer">
  <Properties>
    <httpPort>80</httpPort>
    <managePass>3344</managePass>
  </Properties>
</NodeTemplate>
```

```
- service: agnostic.JBossServer
  properties:
  - id: http
    type: port
    value: 80
```

Figure 4: From TOSCA JBoss to JBoss agnostic component.

so, step (4) includes those agnostic properties into the agnostic component; otherwise, they are discarded.

3.3.3 Plan Generation and Deployment

A CAMP YAML plan is generated from the agnostic graph, which can then be used for deployment. A plan is generated by using another transformation, this time between the agnostic elements and the CAMP-compliant components specifying the component distribution to be deployed. We use Brooklyn to deploy CAMP specifications.

As for the previous transformation, we establish transformation patterns, which are used to analyze the graph and to generate the orchestration plan. For instance, for JBoss servers we map agnostic.service.type.server.JBossServer into org.apache.brooklyn.entity.webapp.jboss.JBoss7Server, and for PostgreSQL we map agnostic.service.type.sql.PostgreSQL into org.apache.brooklyn.entity.database.postgresql.PostgreSqlNode.

The management of properties is accomplished by applying mechanisms similar to the ones described for the transformation between TOSCA and the agnostic types. In this case, a service defined in a concrete CAMP API, like Brooklyn, specifies the properties supported. For example, although the modeling of the agnostic JBoss component supports properties http and https, in the corresponding CAMP API definition only the protocol http is supported, but not https. Therefore, https will not be included in the artifact in the generated plan.

Once the CAMP-compliant orchestration plan is generated, the distribution will depend on the concrete platform implementing the standard, Brooklyn in our case.

Listing 5 shows the CAMP-compliant Brooklyn blueprint for our simple example containing entities corresponding to a JBoss Server and a MySQL, deployed on Amazon (Oregon's Cluster) and SoftLayer (Seattle's Cluster), respectively.

```
services:
- serviceType: org.apache.brooklyn.entity.webapp.jboss.JBoss7Server
  name: JBoss Main Web Server
  location: aws-ec2:us-west-2
  id: JBossMainWebServer
  brooklyn.config:
    port.http: 80+
    port.https:3344+
    ...
- serviceType: org.apache.brooklyn.entity.database.mysql.MySqlNode
  id: db
  name: DB HelloWorld Visitors
  location: softlayer-seattle
  ...
```

Listing 5: CAMP-compliant Brooklyn blueprint.

3.4 Implementation

Our tool TOMAT (TOSCA sMArt Translator) implements the transformation process presented in the previous sections, providing support for the cross-deployment of applications over heterogeneous cross-clouds. TOMAT allows the transformation of TOSCA topologies into (Brooklyn) CAMP specifications. As pointed out in Section 3.2, for the deployment to be carried out, TOMAT requires the given TOSCA topology to include the providers' location. The tool is available at https://github.com/kiuby88/tomat. Some external libraries are used to process the transformation from TOSCA to (Brooklyn) CAMP. Specifically, to interpret the extended TOSCA topology description, we use a module from the OpenTOSCA container which models the TOSCA topology in Java by means of a customised JAXB. For the generation and visualisation of the agnostic graph, we use libraries JGraphT[5] and JGraph,[6] respectively.

[5]JGraphT: http://jgrapht.org/.
[6]JGraph: http://www.jgraph.com/.

4 CONCLUDING REMARKS

We have presented a flexible methodology to deploy cross-cloud applications over heterogeneous clouds through orchestration of services from multiple and diverse providers. Our approach is based on transformation processes between two major OASIS standards on cloud interoperability, namely TOSCA and CAMP, taking advantage of the benefits of each of them. TOSCA is used to specify the application topology, and CAMP to deploy the application modules over clouds.

We have proposed, implemented and validated a translation process from TOSCA-compliant topologies to CAMP-compliant plans, with an intermediate platform-agnostic model that ensures the independency between technologies while facilitates the adaptation to new ones, being only required the definition of translation rules from the agnostic model to the new technology. We consider this standard-based flexibility provides a value-added to previous initiatives to mitigate some vendor lock-in issues.

We would like to insist on two ideas. The unification of the interfaces is implicit in the CAMP definition, as well as the portability and interoperability among services of heterogeneous cloud providers. Thanks to this, despite the differences in the APIs of the final cloud providers, end-users see a unified API which allows them the development of portable and platform-agnostic applications, whose deployment can be distributed over multiple and heterogeneous clouds. Note also that mechanisms for scalability and elasticity supported by cloud providers is still available. For example, the above mapping maps Cluster to its corresponding CAMP-compliant (Brooklyn) element.

Much work remains ahead. Our tool only supports the translation and deployment of Java applications (JBoss, Jetty, Tomcat, ...) and MySQL databases. We are currently working on including other technologies (like PHP and Node) and on increasing the supported database connections. TOSCA has been recently working on the use of YAML-based topology descriptions (OASIS, 2014). We plan to extend our solution to also support the translation process between TOSCA YAML and CAMP YAML. Indeed, it would be interesting to consider the inclusion of this transformation into a system like Brooklyn, so that it can take TOSCA YAML specifications and directly do the deployment. We are currently involved in a project working on the integration of Alien4Cloud and Brooklyn with such goal. Our current implementation has Brooklyn CAMP as the target technology for the transformation. In the future, we will explore the potential of our intermediate agnostic graph to add support for other CAMP/TOSCA specifications, such as Ubuntu Juju[7] and Alien4Cloud.

ACKNOWLEDGEMENTS

This work has been partially supported by EU project FP7-610531 SeaClouds; Spanish MINECO/FEDER projects TIN2014-52034-R and TIN2015-67083-R; Andalusian Government project P11-TIC-7659; and Universidad de Málaga, Campus de Excelencia Internacional Andalucía Tech.

REFERENCES

Armbrust, M. et al. (2010). A view of cloud computing. *Comm. of ACM*, 53(4):50–58.

Binz, T. et al. (2013). OpenTOSCA–a runtime for TOSCA-based cloud applications. In *Service-Oriented Computing*, pp. 692–695. Springer.

Brogi, A. et al. (2014). SeaClouds: a European project on seamless management of multi-cloud applications. *ACM SIGSOFT SEN*, 39(1):1–4.

Carrasco et al. (2015). Towards a flexible deployment of multi-cloud applications based on TOSCA and CAMP. In *ESOCC Workshops*, vol. 508 of *CCIS*, pp. 278–286. Springer.

Kopp, O. et al. (2012). BPMN4TOSCA: A domain-specific language to model management plans for composite applications. In *Business Process Model and Notation*, pp. 38–52. Springer.

Kopp, O. et al. (2013). Winery–a modeling tool for TOSCA-based cloud applications. In *Service-Oriented Computing*, pp. 700–704. Springer.

Mell, P. and Grance, T. (2011). The NIST definition of cloud computing. Technical report. Available at http://dx.doi.org/10.6028/NIST.SP.800-145.

OASIS (2012a). CAMP 1.1. Available at http://docs.oasis-open.org/camp.

OASIS (2012b). TOSCA 1.0. Available at http://docs.oasis-open.org/tosca.

OASIS (2013). Interoperability Demo of OASIS TOSCA. https://www.oasis-open.org/events/cloud/2013/toscademo.

OASIS (2014). TOSCA YAML. Available at http://docs.oasis-open.org/tosca.

Paraiso, F. et al. (2012). A federated multi-cloud PaaS infrastructure. In *IEEE CLOUD*, pp. 392–399. IEEE.

Petcu, D. (2011). Portability and interoperability between clouds: challenges and case study. In *Towards a Service-Based Internet*, pp. 62–74. Springer.

[7]Ubuntu Juju: https://github.com/juju/juju-tosca.