TOSCA for Microservice Deployment in Distributed Control Systems: Experiences and Lessons Learned

Heiko Koziolek*, Rhaban Hark*, Nafise Eskandani*, Phuoc Sang Nguyen† and Pablo Rodriguez*
*ABB Corporate Research
Ladenburg, Germany
Email: firstname.lastname@de.abb.com
†Frankfurt University of Applied Science
Frankfurt, Germany

***Abstract*—The OASIS TOSCA language provides means for specifying the deployment of microservices to cloud-platforms in a vendor-neutral way. Designed in a independent of any application domain, it needs to be tailored to the distributed control systems (DCS), which for example manage the automation in chemical refineries, renewables production, and mining applications. There is still a lack of experience reports applying OASIS TOSCA in real-world settings, therefore the benefits and drawbacks of using this technology are still not well understood. In this context, we designed a simple DCS consisting of several microservices modelled in TOSCA and implemented an according TOSCA orchestrator. We executed a case study deploying the microservices to an on-premise and a cloud-based Kubernetes environment. While TOSCA provides a sophisticated object-oriented language, we found a few specification gaps, challenges when creating portable service templates, and challenges for synchronizing TOSCA orchestrators with DCS engineering tools as well as container orchestrators. The adoption of TOSCA in the process automation domain thus requires more work on the specification and tools and remains a mid-term goal.**

***Index Terms*—software architecture, microservice, deployment, OASIS TOSCA, distributed control systems, case study, modeling, Azure, StarlingX**

I. Introduction

OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) provides a domain-specific language to specify applications and hardware resources independent of a particular cloud platform [1]. Application developers can create TOSCA templates according to this language and process them with TOSCA orchestrators, which parse the templates and translate the statements into deployment actions for infrastructure-as-code tools targeting concrete cloud platforms [2]. Due to its cloud-agnostic design, TOSCA templates shall support multi-cloud deployment for applications as well as portability of applications to new target infrastructures [3]. TOSCA orchestration shall span the entire life-cycle of applications from initial deployment over runtime adaptations to decommissioning [1].

While TOSCA was originally designed for consumer-facing cloud applications, the Open Process Automation Forum[1] of the Open Group is now considering creating a TOSCA profile for the orchestration of distributed control systems [4] (DCS) on premises and/or in the cloud[2]. Such systems control complex industrial processes, such as power production, chemical refinery, or mining. Using TOSCA shall enable a standard-based, vendor-neutral, and interoperable orchestration approach that complies with the long life-cycles of DCS (i.e. often more than 15 years). While the TOSCA specification has been evolved for almost 10 years and many research projects have been carried out around TOSCA [5], TOSCA-based orchestration is still not popular in practice [6]. TOSCA's maturity and suitability for the process automation domain as well as its limitations thus remain underexplored.

In a systematic literature review Bellendorf et al. [5] identified around 20 articles that applied TOSCA on concrete systems. Li et al. [7] used an early XML version of TOSCA to deploy the software components for an HVAC system. Cardoso et al. [8] deployed the SugarCRM platform, an open-source system for customer relationship management.Kostoska et al. [9] deployed an University Management System with TOSCA XML 1.0, commenting that the notation can become challenging for more complex systems. TOSCA for DevOps was shown by Wettinger et al. [10], while Sampaio [11] demonstrated how to use TOSCA in the context of performance testing. Shvetcova et al. [12] summarized benefits and drawbacks when combining TOSCA with Ansible. Luzar et al. [2] tested and compared several current TOSCA orchestrators. None of the approaches dealt with systems from the process automation domain.

The contribution of this paper are experiences and lessons learned when starting to adapt TOSCA for DCS. We executed a case study, creating a minimal TOSCA profile for DCS, specifying a topology template for a simple DCS, implementing a TOSCA orchestrator, and using the template to deploy components to different target environments. Specifically we created resource inventories for the open source Kubernetes

[1]https://www.opengroup.org/forum/open-process-automation-forum

[2]https://www.youtube.com/watch?v=xgmH95Jg0c0

platform StarlingX[3] and a Microsoft Azure Kubernetes Service[4] environment and fed them into our TOSCA orchestrator to learn about the specification's, the state of tool development, and open conceptual issues.

Our main findings include identifying a few specification gaps and experiencing challenges when attempting to create portable service templates. Furthermore, in the process automation domain of DCS already many tools are involved in configuration and monitoring, which would need to be synchronized with a TOSCA orchestrator at runtime. While providing an object-oriented, domain specific language, in practice the TOSCA notation is competing with less sophisticated text templates for specifying cloud workloads in Kubernetes. These are portable across cloud platforms. With the current maturity of TOSCA and its ecosystem and the complexity of DCS systems, an adoption of TOSCA in the process automation domain remains a mid-term goal.

This experience report is structured as follow: Section 2 provides quick overview of the TOSCA landscape, before Section 3 introduce a simplistic TOSCA profile for DCS and explaining the constrains in the process automation domain. We introduce the design rationale for our TOSCA orchestrator in Section 4, before using it in a case study for deploying containerized DCS components to Azure and StarlingX in Section 5. Lessons learned are summarized in Section 6, providing pointers for addressing the open gaps.

II. TOSCA LANDSCAPE

The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) technical committee started its work for interoperable cloud template descriptions in 2012. The TOSCA technical committee is composed of more than 40 organizations and 150 individual representatives, with Cisco, HP, Huawei, IBM, Red Hat, and SAP among them. Version 1.0 of the TOSCA specification was published in 2014 and provided XML schemas for the language definition. The TOSCA Simple Profile in YAML was introduced with version 1.1 in 2018, and the latest version is 1.3 from 2019. Work on version 2.0 with the non-normative TOSCA simple profile stripped out from the main grammar of the language is ongoing, but the specification is still not final as of 2022. Additional work is ongoing for a TOSCA NFV profile and a Function-as-a-Service (FaaS) profile.

TOSCA service templates consist of topology templates, where application developers instantiate pre-defined node and relationship types into node templates and relationship definitions (Fig. 1). A node can for example be a software component or computer host and provides a number of capabilities, while requiring capabilities from other nodes. A node can also have properties (e.g., number of CPU cores, configuration parameters of software components) and expose interfaces (e.g., for creating, updating, or deleting nodes). The operation implementations for the interfaces are not part of the TOSCA specification but can for example be Bash[5] scripts or Ansible[6] playbooks, which interface with cloud platforms. Also new types of relationships can be defined which, like nodes, have properties and interfaces.

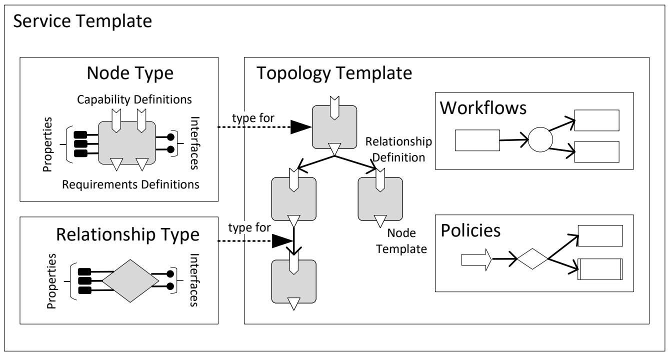


Fig. 1. OASIS TOSCA: Structural Elements

The TOSCA specification also supports workflows to instruct an orchestrator to execute tasks in a specific order. For example, there could be workflows to instruct an orchestrator to perform a backup of a database. Policies can further instruct orchestrators to make deployment decisions within the policy bounds. For example, a scaling policy could instruct an orchestrator to always keep a certain number of service instances deployed. Service templates and type definitions are packaged into a Cloud Service Archive (CSAR), which can also contain artifacts, such as installers, software container images or VM images.

Developers can use different kinds of tools in the context of TOSCA service templates[7]: TOSCA modeling tools allow creating TOSCA service templates with a graphical user interface, where for example nodes can be connected using relationships and configurations can be entered before generating a CSAR file. Examples are Cloudify, Alien4Cloud or Eclipse Winery [13]. TOSCA processors or parsers can parse TOSCA service templates. Examples are TOSCA-Parser, JTOSCA, Puccini, or xOpera. TOSCA orchestrators process the information extracted from CSAR files, make deployment decisions e.g., based on policy definitions and workflows, and then instruct artifact processors, such as Python scripts or Infrastructure-as-Code tools to carry out the actual deployment. Examples are OpenTOSCA [14], xOpera, ToSKer, Ubicity, or Yorc.

Bellendorf and Mann [5] provided a survey of TOSCA approaches and case studies reported in academic literature. They surveyed 124 papers from 2012-2019 and categorized them into tools, language extensions, methods for processing TOSCA models, and TOSCA case studies. TOSCA usages have been reported in DevOps, IoT, NFV, and testing. They also found that security topics, such as privacy and data protection in context with TOSCA hardly have received attention

[3]https://www.starlingx.io/
[4]https://azure.microsoft.com/en-us/products/kubernetes-service/
[5]https://www.gnu.org/software/bash/
[6]https://www.ansible.com/
[7]https://github.com/philippemerle/tosca-implementation-landscape

in academic communities, furthermore automated deployment optimization remains underexplored.

In the software architecture community, researchers proposed a number of approaches involving TOSCA. Artac et al. [15] introduced DICER, a model-driven approach for the deployment of data-intensive architectures. They devised a UML profile and according model transformation to TOSCA and Chef, which they tested on Hadoop and WikiStats. Lipton et al. [1] provided an overview of TOSCA and available tooling, pointing out that TOSCA may provide more unification and stability in a area of rapidly shifting technologies, since it is based on a technology-neutral internationally agreed standard. DiFrancesco et al. [16] mentioned TOSCA as part of a systematic mapping study on architecting microservices. They viewed TOSCA as the most promising candidate for an industrial standard describing the architecture or microservice-based system, but also found that none of their primary standards on designing microservice architectures had actually used TOSCA.

DesLauriers et al. [17] presented the MICADO engine to generate infrastructure-as-code scripts from TOSCA models, which was created in the EU Horizon 2020 Project COLA. MiCADO provides container orchestration via Kubernetes and supports worker node provisioning using Terraform and Occopus. The workloads can be deployed to OpenStack, Microsoft Azure, AWS, and Google cloud. They successfully applied MiCADO to deploy the microservices of the DIGITbrain system for applying machine learning in the manufacturing industry. In 2022, Alonso et al. [18] proposed a DevSecOps approach, where TOSCA service templates are generated after model-checking abstractions of an execution environment.

As part of the Open Process Automation Forum (OPAF) of the Open Group, the companies ExxonMobil and CPLANE.ai showed a demonstration of installing DCS software via TOSCA templates and subsequently orchestrating a system composed of several automation controllers and servers simulating a chemical process[8]. OPAF is now working on adapting TOSCA for the process automation domain.

III. Distributed Control System Model in TOSCA

To derive TOSCA models for a DCS, we used as "strawman" template a Simplified Distributed Control System (sDCS), which is depicted in Fig. 2 in UML. The system consists of three Distributed Control Nodes (DCN1-3, i.e., DCS controllers) and an Advanced Computing Platform (ACP, i.e., one server). In contrast to PLCs for a machine or robot, DCS controllers typically control a large number of devices in process automation. The system controls the filling level in a distillation column, depicted in Fig. 2 as a schematic Piping and Instrumentation Diagram (P&ID) on the left-hand side. Each host runs a Function Block Execution engine, e.g., an IEC 61131-3 runtime [19]. These engines cyclically execute function blocks deployed to them, calculating output actuator signals based on current sensor data. For example, DCN3 executes a PID (proportional/integral/derivative controller) to compute new output signals for a flow valve, based on the inputs from a level controller and flow transmitter.

The hosts and applications in the sDCS example have been chosen so that they cover different typical situations. DCNs are for example embedded devices or Industry PCs that can optionally include direct field device connections. DCN1 does not execute any application logic by itself, but just relays the signals from level transmitter LT101-1 into the network for other hosts to process. DCN2 executes the filtering logic for two analog input signals, with one of them actually processing the signal for LT101-1 coming from DCN1. DCN3 resembles a smart actuator that computes the application logic for the flow valve controller and also filters the output signal, which is relayed to the flow valve FV101-1, which is directly connected to this host. ACP1 is a more powerful server node and hosts a PID function block, executing higher-level control logic for the level controller. It also hosts services for supervision, i.e. a Human Machine Interface service, an Alarm Management service and a Historian service, as well as infrastructure services, such as an OPC UA Aggregation Server, and an OPC UA Global Discovery Server, and Engineering Tools [20]. In practical situations there are many more DCNs and ACPs, the nodes execute many more software services, and the network topologies are more sophisticated. Furthermore, each host may require operating system configuration as well as setup of virtualization software from an orchestration system.

Based on the sDCS example, we derived a simple TOSCA node model to explore the concepts for an DCS orchestration system. Fig. 3 shows a UML visualization of the model, where each UML class resembles a TOSCA node, properties and methods are omitted for brevity. The nodes `SoftwareComponent` and `Compute` have been imported from the TOSCA 1.3 simple YAML profile and thus directly connect to TOSCA. As one important DCS software component type, we introduce `FunctionBlockEngine`, which hosts `FunctionBlockApplication`. There may be different types of such engines [21] and different vendor-specific variants, which could inherit from this node type. Instances of function block compositions are typically bundled into *applications*, which are a unit of execution and are defined in engineering tools. We assume that the function block instances visible in Fig. 2 are already bundled into such applications. Further software component types are `SoftwareServices` (which subsume different supervision and engineering components for now) and IoEngines. The latter have the TOSCA capability of being a `CommunicatedSignalProvider`, which means they can publish the latest values of specific signals into the network.

For simplicity we modelled two types or TOSCA `Compute` node types, namely `DCN` and `ACP`. Both of these types have the capability to be `DistributedControlPlatforms`, which means that these hosts support specific network profiles, security facets, and system management profiles necessary in a DCS. Other hardware and software capabilities are not detailed

[8]https://cplaneai.com/wp-content/uploads/2020/09/Orchestration-of-an-Open-Industrial-Control-System-1.pdf

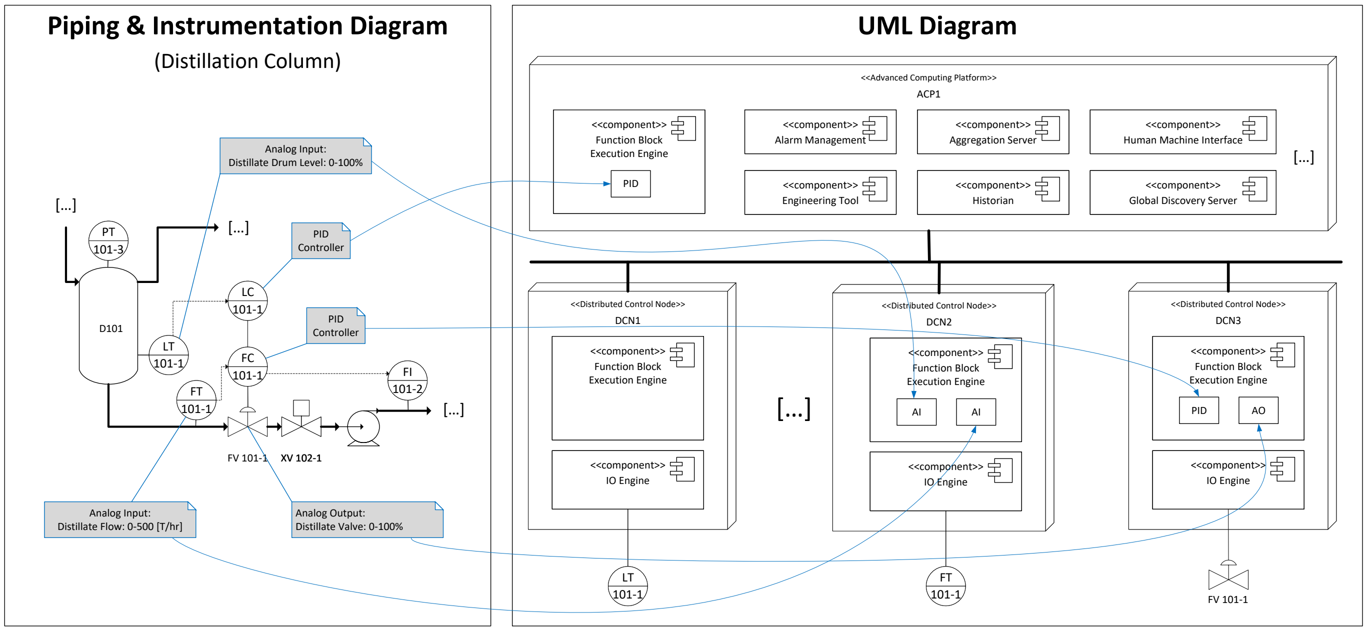

Piping & Instrumentation Diagram
(Distillation Column)
Analog Input: Distillate Drum Level: 0-100%
PID Controller
PT 101-3
LC 101-1
LT 101-1
D101
FC 101-1
FI 101-2
FT 101-1
FV 101-1
XV 102-1
Analog Input: Distillate Flow: 0-500 [T/hr]
Analog Output: Distillate Valve: 0-100%
UML Diagram
<<Advanced Computing Platform>> ACP1
<<component>> Function Block Execution Engine
PID
<<component>> Alarm Management
<<component>> Aggregation Server
<<component>> Human Machine Interface
<<component>> Engineering Tool
<<component>> Historian
<<component>> Global Discovery Server
<<Distributed Control Node>> DCN1
<<Distributed Control Node>> DCN2
<<Distributed Control Node>> DCN3
<<component>> IO Engine
AI
AO
[...]

Fig. 2. Simplified Distributed Control System: P&ID for a Distillation Column with annotated automation requirements (e.g., sensors), tracing to a UML representation of the DCS components

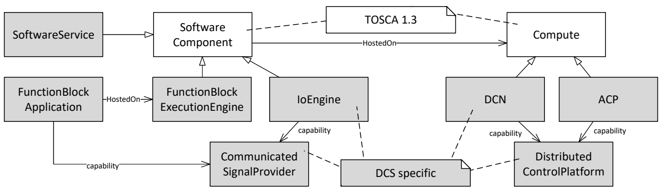

SoftwareService
Software Component
TOSCA 1.3
Compute
HostedOn
FunctionBlock Application
FunctionBlock ExecutionEngine
IoEngine
DCN
ACP
capability
Communicated SignalProvider
DCS specific
Distributed ControlPlatform

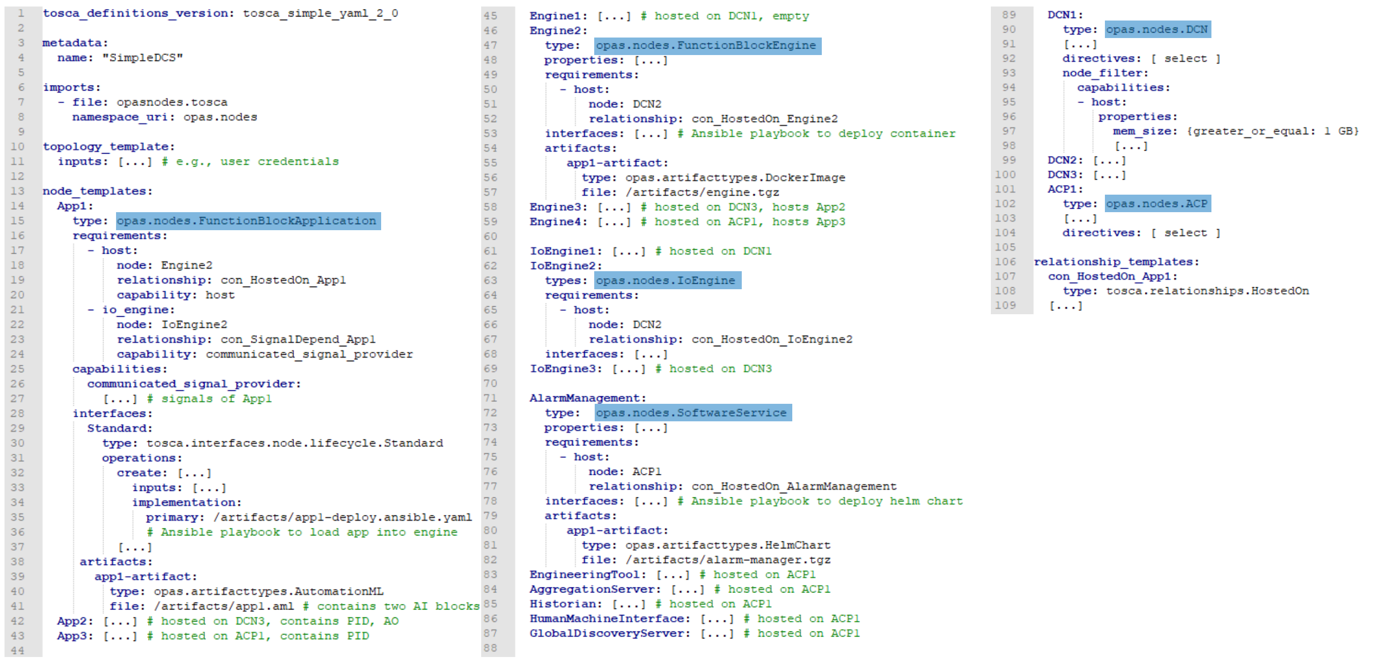
Fig. 3. Simple TOSCA Model for Distributed Control Systems

for now, but it is conceivable to model for example real-time capabilities or safety mechanism into such node types. Network devices and topologies are not yet included in the model for simplicity, as the application deployment currently assumes making decision only based on host capabilities, but not on network capabilities. Existing TOSCA network models for basic networking or network function virtualization could be integrated here as a starting point, while specific DCS models can be extended in the future.

Adding more detail to the model should be driven by future deployment requirements, however unecessary complexity must be avoided. In future DCS, an orchestration system shall take as many deployment decisions as possible and relieve system administrators and plant operators from manually mapping control functions to complex virtualized IT infrastructures consisting of dozens of hosts. Thus, it shall for example be possible to choose appropriate deployment hosts for function block applications during initial commissioning and to dynamically adjust the deployment during operation without affecting the underlying production [22]. Therefore, the deployment relationships visible in Fig. 2 should only be abstractly defined over certain Quality-of-Service requirements, but leave an orchestration system sufficient degrees of freedom to choose an appropriate deployment. This can be supported by TOSCA's select and substitute directives.

To explain the implementation of the TOSCA model shown in Fig. 3, we show a YAML representation of a TOSCA node template according to the model and the sDCS in Fig. 4. It illustrates the additional concepts needed to make the model operative but does not provide a fully complete view. The newly defined DCS profile from Fig. 3 needs to be imported (line 7), so that the newly defined types can be used. The topology template can for example capture global parameters, such as the user credentials needed to access the different hosts (line 11). The `App1` defined in line 14 is a `FunctionBlockApplication` hosted on DCN2 and executes two AI function blocks (also see Fig. 2). Besides the host requirement, it also requires the presence of an `IoEngine2` having the capability of being a `CommunicatedSignalProvider` for the signals required by the Analog Input function blocks. Notice that this requirement does not necessarily imply a deployment of function block engine and IO engine on the same computing host.

To deploy `App1`, an orchestrator needs to refer to the implementation of a deploy script, here an Ansible playbook to upload the application into a `FunctionBlockExecutionEngine`, e.g., via the OPC UA protocol. The application `App1` itself is a AutomationML file (line 41) containing a PLCopen specification of the control logic of the two analog input function blocks in IEC 61131-3 Structured Text. `Engine2` defined in line 46, is a deployable `SoftwareComponent` of the type `FunctionBlockEngine`. It requires a DCN host node



```yaml
tosca_definitions_version: tosca_simple_yaml_2_0

metadata:
  name: "SimpleDCS"

imports:
  - file: opasnodes.tosca
    namespace_uri: opas.nodes

topology_template:
  inputs: [...] # e.g., user credentials

node_templates:
  App1:
    type: opas.nodes.FunctionBlockApplication
    requirements:
      - host:
          node: Engine2
          relationship: con_HostedOn_App1
          capability: host
      - io_engine:
          node: IoEngine2
          relationship: con_SignalDepend_App1
          capability: communicated_signal_provider
    capabilities:
      communicated_signal_provider:
        [...] # signals of App1
    interfaces:
      Standard:
        type: tosca.interfaces.node.lifecycle.Standard
        operations:
          create: [...]
            inputs: [...]
            implementation:
              primary: /artifacts/app1-deploy.ansible.yaml
              # Ansible playbook to load app into engine
          [...]
     artifacts:
       app1-artifact:
         type: opas.artifacttypes.AutomationML
         file: /artifacts/app1.aml # contains two AI blocks
  App2: [...] # hosted on DCN3, contains PID, AO
  App3: [...] # hosted on ACP1, contains PID

  Engine1: [...] # hosted on DCN1, empty
  Engine2:
    type:  opas.nodes.FunctionBlockEngine
    properties: [...]
    requirements:
      - host:
          node: DCN2
          relationship: con_HostedOn_Engine2
    interfaces: [...] # Ansible playbook to deploy container
    artifacts:
       app1-artifact:
         type: opas.artifacttypes.DockerImage
         file: /artifacts/engine.tgz
  Engine3: [...] # hosted on DCN3, hosts App2
  Engine4: [...] # hosted on ACP1, hosts App3

  IoEngine1: [...] # hosted on DCN1
  IoEngine2:
    types: opas.nodes.IoEngine
    requirements:
      - host:
          node: DCN2
          relationship: con_HostedOn_IoEngine2
    interfaces: [...]
  IoEngine3: [...] # hosted on DCN3

  AlarmManagement:
    type:  opas.nodes.SoftwareService
    properties: [...]
    requirements:
      - host:
          node: ACP1
          relationship: con_HostedOn_AlarmManagement
    interfaces: [...] # Ansible playbook to deploy helm chart
    artifacts:
       app1-artifact:
         type: opas.artifacttypes.HelmChart
         file: /artifacts/alarm-manager.tgz
  EngineeringTool: [...] # hosted on ACP1
  AggregationServer: [...] # hosted on ACP1
  Historian: [...] # hosted on ACP1
  HumanMachineInterface: [...] # hosted on ACP1
  GlobalDiscoveryServer: [...] # hosted on ACP1

  DCN1:
    type: opas.nodes.DCN
    [...]
    directives: [ select ]
    node_filter:
      capabilities:
      - host:
          properties:
            mem_size: {greater_or_equal: 1 GB}
            [...]
  DCN2: [...]
  DCN3: [...]
  ACP1:
    type: opas.nodes.ACP
    [...]
    directives: [ select ]

relationship_templates:
  con_HostedOn_App1:
    type: tosca.relationships.HostedOn
  [...]
```

Fig. 4. Simplified Distributed Control System in TOSCA YAML based on defined types

and is, in this example, provided as a Docker image to be executed by a container runtime. Notice that this implies that the DCN host node has such a runtime installed, which could be part of the orchestration, but is not detailed here for simplicity.

`IoEngine2` is of type `IoEngine` and so far only treated the same as a function block execution engine. `AlarmManagement` is a software service to be hosted on an ACP. This service is provided as a Helm chart (Kubernetes package manager), and needs to be deployed into a Kubernetes orchestration framework hosted on one or more ACPs. The `DCNs` are abstractly defined using TOSCA *select* directives and node filters to allow the orchestrator to for example match memory requirements to the host nodes in an inventory. This allows an orchestrator to make independent deployment decisions and optimize the deployment considering other factors, such as current workload, maintenance phases, or hardware replacements.

An orchestrator must thus support the included artifact types and implementation types, so that the entire node template can be processed. It can manage the nodes in scope of an inventory that needs to be dynamically adjusted to the hosts in the network and the status of the deployed components. For example, in case of the nodes being part of a Kubernetes cluster, an orchestrator can retrieve runtime information for nodes and services via the Kubernetes API. To deploy software, the orchestrator or its integrated artifacts need to have sufficient authorization to install and modify software on each of the managed host nodes. This requires careful consideration of security properties.

IV. TOSCA Orchestrator

We searched for an appropriate TOSCA Orchestrator, considering number of different requirements: it shall be able to process TOSCA 2.0 YAML templates including select and substitution directives to be able to select cloud resources or substitute vendor-specific DCS components. It shall provide a dynamic inventory that can be populated with information from Redfish servers discovered via SSDP. It shall support Ansible, shell script, and other artifacts, but not be tied to a specific deployment environment.

We checked the "official" list of known TOSCA implementations[9] as well as the TOSCA implementation landscape[10] maintained by Philippe Merle, which provide more than a dozen of TOSCA orchestrators, available either commercially or as open source. Although the TOSCA specification defines conformance criteria for orchestrators, such as the ability to process CSAR-files and support the entire TOSCA grammar, it is unknown which orchestrators have been subject to compliance testing. There is no list of fully compliant orchestrators available.

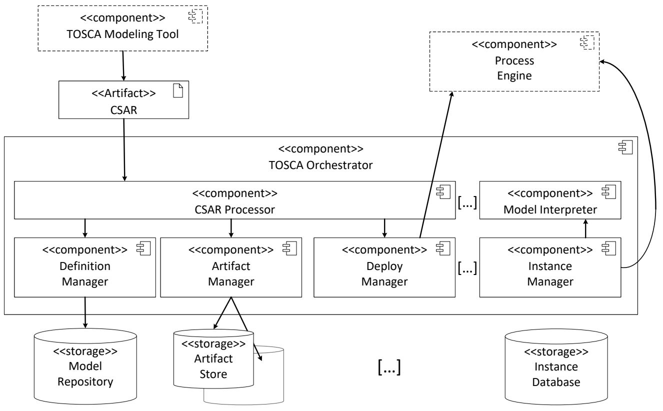
We checked a few existing orchestrators to learn about their suitability for our context. Cloudify uses its own Cloudify Domain Specific Language, which is based on TOSCA, but cannot process all kinds of TOSCA templates. The OpenTOSCA runtime was bound to using XML-based CSARs as input. xOpera claims compliance to TOSCA YAML v1.3, but we could not find examples using select or substitute directives. xOpera is also restricted to using Ansible artifacts.

[9]https://github.com/oasis-open/tosca-community-contributions/wiki/Known-TOSCA-Implementations

[10]https://github.com/philippemerle/tosca-implementation-landscape

TORCH lacked appropriate documentation. The documentation of Unfurl lists substitution mappings as a ”not yet implemented” feature. Yorc supports only TOSCA Simple Profile in YAML 1.2. Ubicity seems to support TOSCA YAML 2.0 and select/substitute directives but is only available commercially. Puccini supports parsing TOSCA YAML 2.0 but is not a full orchestrator. Its companion application Turandot supports select and substitute directives, but is restricted to using Kubernetes resources. None of the orchestrators seems to support some kind of inventory *discovery* as desired for process automation applications, since this is not part of the TOSCA specification and considered implementation-specific.

As no existing orchestrator seemed to fit our requirements, we started to create a custom implementation. The TOSCA Primer Document[11] targets software developers and provides a minimal reference architecture as a template for implementing TOSCA orchestrators (Fig. 5).



<<component>> TOSCA Modeling Tool
<<Artifact>> CSAR
<<component>> Process Engine
<<component>> TOSCA Orchestrator
<<component>> CSAR Processor
[...]
<<component>> Model Interpreter
<<component>> Definition Manager
<<component>> Artifact Manager
<<component>> Deploy Manager
[...]
<<component>> Instance Manager
<<storage>> Model Repository
<<storage>> Artifact Store
[...]
<<storage>> Instance Database

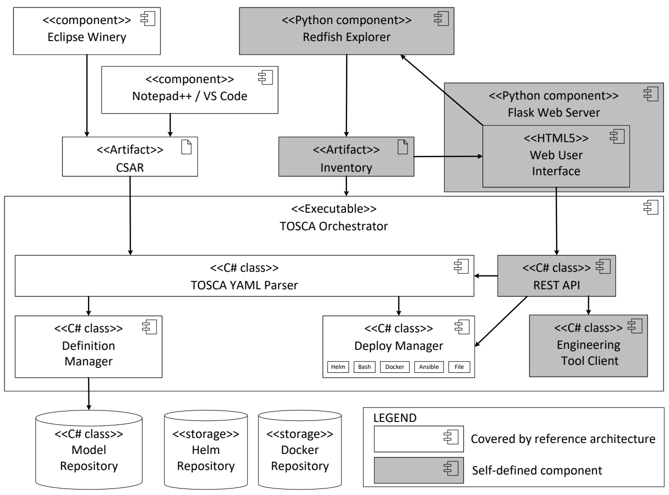
Fig. 5. TOSCA Orchestrator Reference Architecture

A `TOSCA Modeling Tool` provides a user interface to create `CSAR`-files. The tool can be a simple text editor or a graphical modeling tool, such as Eclipse Winery, Cloudfiy, or Alien4cloud. A CSAR archive serves as input for the actual `TOSCA Orchestrator` and is first processed by the `CSAR Processor`. This includes extracting the included files and parsing their YAML or XML contents. The `Definition Manager` stores the retrieved information in a `Model Repository`.

The `Artifact Manager` extracts artifacts, such as VM or Docker images, application installers, or executables out of the CSAR archive and stores them in one or multiple `Artifact Stores`. Once this is done, the `Deploy Manager` carries out the deployment of the nodes and artifacts in the target environment using the included TOSCA implementations. This can for example include triggering a cloud provisioning tool to create virtual machines, installing virtualization software like a container runtime, and installing software services on the target nodes.

An optional `Model Interpreter` may be used to transform a declarative TOSCA template into an imperative one before handing it over to the `Deploy Manager`. The latter can optionally make use of a `Process Engine` to process workflow definitions in the TOSCA template. The `Instance Manager` creates an instance of the cloud application, which it manages through the `Instance Database`.

With the reference architecture as template, we created our own TOSCA orchestrator implementation in .NET/C# as depicted in Fig. 6. As TOSCA modeling tool, we used Eclipse Winery for graphical modeling as well as Notepad++ and Visual Studio Code for textual refinement. We also specified an `Inventory` file for the orchestrator to work with. It contains a list of host nodes and can be connected to a separate file containing user credentials to connect to these nodes.



<<component>> Eclipse Winery
<<Python component>> Redfish Explorer
<<component>> Notepad++ / VS Code
<<Python component>> Flask Web Server
<<HTML5>> Web User Interface
<<Artifact>> CSAR
<<Artifact>> Inventory
<<Executable>> TOSCA Orchestrator
<<C# class>> TOSCA YAML Parser
<<C# class>> REST API
<<C# class>> Definition Manager
<<C# class>> Deploy Manager
Helm
Bash
Docker
Ansible
File
<<C# class>> Engineering Tool Client
<<C# class>> Model Repository
<<storage>> Helm Repository
<<storage>> Docker Repository
LEGEND
Covered by reference architecture
Self-defined component

Fig. 6. DCS TOSCA Orchestrator Architecture (Prototype)

The `Redfish Explorer` can generate the `Inventory` file. This is for example useful in situations where a DCS is already installed on premises or in a cloud hosting environment and shall co-host the application components specified in the CSAR. DMTF Redfish[12] is a system management standard used in process automation.

Redfish servers provide a RESTful interface and system management models covering for example CPU architecture, memory, storage, endpoints etc. An orchestrator can potentially use the information retrieved from a Redfish server to inform autonomous deployment decisions. For example, the Redfish model could contain information on the IoDevices directly connected to a DCN, and thus the orchestrator could decide to deploy specific control applications to the DCN to lower communication latency between controller and IO. Similar to UPnP servers, Redfish servers support discovery via the SSDP protocol.

In our implementation, a user can initiate the Redfish discovery process in a given subnet via a `Web User Interface` hosted in a `Flask Web Server`. The WebUI also displays the results of the TOSCA template parsing pro-

[11]http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.html

[12]https://www.dmtf.org/standards/redfish

cess and provides users the ability to modify initial deployment decisions manually.

Inside the TOSCA Orchestrator, the `TOSCA YAML Parser` processes CSAR archives and parses the included TOSCA YAML 2.0 files, supporting constructs, such as TOSCA 'select' and 'substitute' directives. The orchestrator finds concrete instances for abstract node definitions marked with directives. Select directives are preferably resolved using concrete nodes from the host inventory. It resolves substitute directives by searching for any additionally provided topologies in the `Model Repository` that are marked as being a valid substitute for this specific abstract type. Within both, it applies any filters defined specification.

The orchestrator also supports operation calls, e.g., to create, configure, or delete nodes. These are automatically called depending on the lifecycle stage and according to the specification. The orchestrator forwards any expected inputs to the artifacts of the operations and consumes their outputs to store them as attributes in nodes.

The parser does not yet support TOSCA workflows, groups and policies, since these were not yet used in the guiding sDCS scenario but are expected to become important in the future to allow the orchestrator to make more independent decisions. The `Definition Manager` creates a TOSCA model from the information gathered by the parser, which can then be traversed to carry out deployment actions.

Our orchestrator does not feature an `Artifact Handler` to store artifacts supplied in CSARs into repositories. It can pass Helm charts included in a CSAR to deploy Kubernetes workloads to Ansible playbooks but does not directly interact with repositories. For simplification, we assume that the deployment artifacts referenced in the CSARs are already stored in pre-configured artifact repositories, e.g., Docker repositories for Docker images.

The user can trigger the deployment process via the WebUI that delegates to the `Deploy Manager`. This component can deal with Helm, Bash, Docker, Ansible, and File artifacts, which is sufficient to process the referenced artifacts in the TOSCA sDCS template in Fig. 4. User credentials for the artifact repositories and host nodes can be supplied by configuration files or interactively queried from the user executing the orchestration. The TOSCA specification does not provide a standard way of handling user credentials for host nodes and considers it implementation-specific.

The orchestrator also includes an `Engineering Tool Client` that can query existing DCS Engineering Tools to distribute their configurations onto the previously installed application components. In the case of a DCS, these are for example control applications containing function block assemblies for the use case at hand to be fed into Function Block Execution Engines. Another example are operator process graphics for HMI components.

V. Case Study

We executed a first case study to determine the feasibility and challenges when using the Simple TOSCA Model for DCS and our custom implemented TOSCA orchestrator to deploy the sample sDCS to two different target environments. We chose as first target environment an Azure AKS Cluster in a public cloud to emulate the use case of performing simulation-based DCS testing in the cloud without physical IO connections. In this case the signals coming from the IO devices need to be simulated. As second target environment, we chose a local on-premise K8s cluster based on the open source StarlingX K8s distribution. Each environment included one K8s master node and four worker nodes to reflect the four nodes (3x DCN + 1x ACP) required by the sDCS application.

Notice that our simplified sDCS application relies on statically defined assignments of applications and function block engines to particular DCNs. The DCN templates contain a TOSCA select directive (Fig. 4, line 92), to allow the orchestrator to select a concrete node from a given inventory. Other than this selection, the orchestrator cannot yet make any additional deployment decisions for the sDCS application, since it lacks policy definitions and for example does not include a dynamic selection of DCNs for given function block execution engines. Fig. 7 shows the selection schematically.

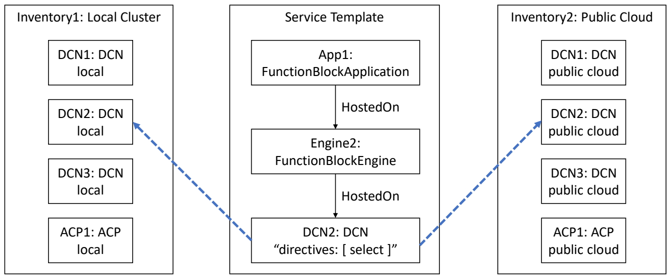
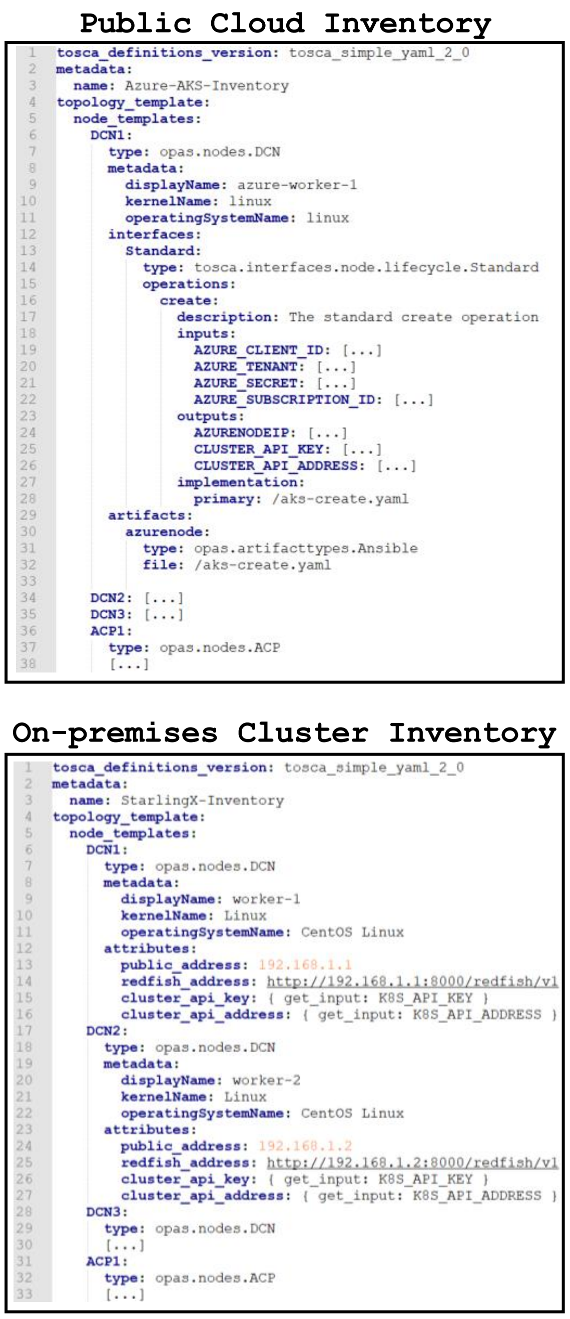


Fig. 7. Choosing a target environment using the TOSCA 'select' directive and different inventories.

We assume for our case study that the human orchestration user makes the decision for a target environment manually, by providing the orchestrator a specific inventory file via the user interface before parsing and processing the TOSCA template. The TOSCA 'select' directive triggers a standard mechanism as specified in the TOSCA 2.0 specification but is however not supported by all orchestrators. Implementing the mechanism in a way that the main TOSCA template remains environment independent required several subjective decisions, since the TOSCA specification itself provides no guidance in how to obtain the inventories.

In case of the public cloud deployment, we assumed that a Kubernetes cluster needed to be created from scratch to emulate the usage scenario of ad-hoc software testing in a cluster before the actual automation equipment was installed on premises. For our case study, we chose the Microsoft Azure Kubernetes Service and used an Ansible playbook together with the Azure.Azcollection module for Ansible to bootstrap the cluster. Fig. 8 shows the public cloud inventory for Azure-AKS, which embeds the Ansible playbook 'aks-create.yaml'. To be able to execute the playbook, an Azure client ID, tenant ID, secret, and subscription ID are required. The playbook


Public Cloud Inventory

```
tosca_definitions_version: tosca_simple_yaml_2_0
metadata:
  name: Azure-AKS-Inventory
topology_template:
  node_templates:
    DCN1:
      type: opas.nodes.DCN
      metadata:
        displayName: azure-worker-1
        kernelName: linux
        operatingSystemName: linux
      interfaces:
        Standard:
          type: tosca.interfaces.node.lifecycle.Standard
          operations:
            create:
              description: The standard create operation
              inputs:
                AZURE_CLIENT_ID: [...]
                AZURE_TENANT: [...]
                AZURE_SECRET: [...]
                AZURE_SUBSCRIPTION_ID: [...]
              outputs:
                AZURENODEIP: [...]
                CLUSTER_API_KEY: [...]
                CLUSTER_API_ADDRESS: [...]
              implementation:
                primary: /aks-create.yaml
      artifacts:
        azurenode:
          type: opas.artifacttypes.Ansible
          file: /aks-create.yaml

    DCN2: [...]
    DCN3: [...]
    ACP1:
      type: opas.nodes.ACP
      [...]
```

On-premises Cluster Inventory

```
tosca_definitions_version: tosca_simple_yaml_2_0
metadata:
  name: StarlingX-Inventory
topology_template:
  node_templates:
    DCN1:
      type: opas.nodes.DCN
      metadata:
        displayName: worker-1
        kernelName: Linux
        operatingSystemName: CentOS Linux
      attributes:
        public_address: 192.168.1.1
        redfish_address: http://192.168.1.1:8000/redfish/v1
        cluster_api_key: { get_input: K8S_API_KEY }
        cluster_api_address: { get_input: K8S_API_ADDRESS }
    DCN2:
      type: opas.nodes.DCN
      metadata:
        displayName: worker-2
        kernelName: Linux
        operatingSystemName: CentOS Linux
      attributes:
        public_address: 192.168.1.2
        redfish_address: http://192.168.1.2:8000/redfish/v1
        cluster_api_key: { get_input: K8S_API_KEY }
        cluster_api_address: { get_input: K8S_API_ADDRESS }
    DCN3:
      type: opas.nodes.DCN
      [...]
    ACP1:
      type: opas.nodes.ACP
      [...]
```

Fig. 8. Public cloud inventory vs. on-premises cluster inventory to be fed into a TOSCA orchestrator (excerpts)

(not shown in detail here for brevity) first checks if a K8s Master node is already deployed and instantiates a new Master node otherwise. Afterwards, it adds a new worker node and returns its IP address, so that the orchestrator can later pass it to other deployment tools. As TOSCA processes each DCN individually, this procedure needs to be repeated for each DCN.

In case of the on-premise cluster deployment, we assumed that a Kubernetes cluster already existed to emulate the usage scenario that DCS applications shall be deployed into an already bootstrapped cluster after initial field commissioning. For our case study, we chose a local StarlingX Kubernetes cluster with in-band Redfish servers on each node as the target environment. While the nodes could be also retrieved via the K8s API, we use Redfish here, since it is not tied to Kubernetes, and we envision scenarios where K8s is not available. Thus, the inventory was created by the `Redfish Explorer`, which used the SSDP protocol to determine the available nodes and their capabilities. The information is statically stored into the inventory file, which thus already contains specific IP addresses when invoking the orchestrator. There are no additional Ansible playbooks needed in this case, however the orchestrator needs to access the K8s API and API key to be able to deploy workloads into the cluster. Additional information from the Redfish servers (e.g., the host's CPU and memory, as well as connected IO devices) could be passed to the orchestrator here to be able to make more sophisticated deployment decisions in the future.

Given the inventories and additional security keys, the TOSCA orchestrator could process the entire TOSCA template for the sDCS application (Fig. 4) and deploy the specified applications to the respective nodes. This involved triggering additional Ansible playbooks to start Docker containers, install Helm charts, and upload applications in AutomationML syntax into the function block engines. Table I shows the overall measured deployment times for the two target environments. The public cloud deployment takes much longer, since the entire K8s cluster is bootstrapped before deploying the applications, whereas for the on-premise deployment the K8s cluster was already set up.

	Public cloud Deployment Time	On-premises Deployment Time
Test Run 1	632 sec	68 sec
Test Run 2	635 sec	75 sec
Test Run 3	641 sec	70 sec
Test Run 4	627 sec	76 sec
Test Run 5	594 sec	72 sec

TABLE I
DEPLOYMENT TIMES FOR THE sDCS APPLICATION

The main TOSCA template required no changes for the two different target environments, as we were able to move all platform dependencies and technology specifics into the inventories and Ansible playbooks. Thus, the template should be reusable for other deployment targets as well. The Ansible playbooks are hardly reusable in other contexts independent from the orchstrator, since they are specially structured for the TOSCA processing (e.g., each K8s worker node is set up individually). Our TOSCA orchestrator can notify a DCS engineering tool of changes to the deployment via its `Engineering Tool Client`. This is however not covered by the TOSCA specification and thus a proprietary extension. Other TOSCA orchestrators would leave DCS engineering tools unsynchronized, which can introduce conflicts when altering the DCS configuration (e.g., assigning an application to another node) via the engineering tools.

VI. Lessons Learned

We summarize several learnings, experiences, and observations from our TOSCA modeling, orchestrator implementation and case study in the following. We report on these experiences to aid in accelerating TOSCA's specification and tooling maturity to become more widespread in practice.

Gaps in TOSCA specification: While implementing the TOSCA orchestrator, we found that the TOSCA specification lacks guidance on how certain features are supposed to be implemented, which may hurt compatibility between orchestrators. For example, TOSCA allows to specify input and output parameters for operations, but concrete artifact processors may provide different options of handling such parameters. Bash scripts can for example receive parameters by passing them on invocation, by reading them from a file, or by retrieving them from previously set environment variables. The TOSCA specification mentions the preferred use of environment variables for parameter passing, but this is not normative and may be problematic if passwords are involved. For other kinds of artifacts, there is no preferred method of passing parameters in the specification.

Notification handling is similarly underspecified as parameter handling. Furthermore, the TOSCA grammar does not allow `node filters` to use functions to pass node attributes from additional input files. Resource `inventories` are only briefly mentioned in the specification, but their usage remains unclear and the implementation is left to the orchestrator developer, which may make TOSCA templates non-portable. We decided to implement the inventories for our sDCS application as TOSCA topology templates, but other types of implementations are conceivable. In general, it is left open how TOSCA `select` and `substitution` directives shall be resolved if multiple matches are detected. Thus, different orchestration implementations could end up with different results given the same inputs.

Artifact handlers tied to orchestrator implementations: The consequence of omitting standardized parameter passing is that artifact processors are essentially tied to a particular TOSCA orchestrator implementation as both have to have a compatible mechanism. Because the artifact handlers are embedded into TOSCA templates, these templates need to be edited to work with other TOSCA orchestrators.

But with TOSCA templates tied to a specific orchestrator implementation, the benefits of using a standard notation are reduced, and the template user is tied to a specific tool vendor-This is especially crucial in the industrial automation domain, where systems may need to be maintained for decades and possibly have longer life-cycles than specific IT tools. OASIS UML models had a similiar tool exchange problems initially prompting the development of XMI [23].

TOSCA orchestrators not fully standard-compliant: While the specification includes conformance criteria for TOSCA processors and orchestrators, we did not find a complete set of reusable test cases for orchestrator developers to perform conformance checks. When screening the tool landscape for TOSCA orchestrators, we noticed that many of them for example do not yet support substitution mappings, which were only (re-)introduced in the TOSCA specification in 2019, after previous incomplete attempts. Many TOSCA examples found in open source repositories only use simple TOSCA concepts, but do not utilize all provided possibilities. This again may make TOSCA templates unprocessable if they use certain constructs that a TOSCA orchestrator does not support. This issue may be resolved once the specification stabilizes, and the TOSCA orchestrator implementations evolve and are TOSCA-compliance tested.

Challenge of creating service templates for different target environments: While in theory TOSCA decouples the topology model from technology and implementation specifics, in current practice most TOSCA orchestrators seem to be tied to specific artifacts (e.g., only Python scripts) and technologies (e.g., only Kubernetes resources) and cannot be broadly used in other situations. The most popular commercial orchestrator Cloudify even defined its own domain-specific language inspired by TOSCA, which may render the service templates unprocessable by TOSCA parsers.

Furthermore, creating service templates applicable for different cloud platforms (e.g., Microsoft Azure, Amazon AWS) requires dealing with environment-specific properties. In our simple example, we were able to contain the environment-specific properties in the used inventories, since we aimed at rather generic container and K8s application workloads. More sophisticated real-world templates that allow to switch the deployment from one cloud provider to another provider and do not only include containers and helm charts however likely would likely make use of such properties. Best practice recommendation from the TOSCA specification is to create abstract TOSCA nodes and to use substitution mappings to replace these with concrete ones specific for a selected target environment. Unfortunately, substitution mappings are not yet supported by most orchestrators.

Challenge of synchronizing with DCS engineering tools: In the domain of distributed control systems, the deployment of function block applications to DCNs is classically done through vendor-specific, proprietary engineering tools [4]. For example, a control logic editor enables an automation engineer to instantiate function blocks, such as a PID block, from a library and assemble them to implement a control function [21]. Such a tool is also used to assign the resulting function block application to configured DCN nodes. These tools then provide an upload function to the DCNs to deploy the applications.

This functionality obviously overlaps with the intention for a TOSCA orchestrator, which classically also assigns applications to nodes. To avoid conflicting specifications and inconsistencies, a TOSCA orchestrator could work with a TOSCA template *generated* from information retrieved from the aforementioned engineering tools. Other forms of generation have been enabled by the recently evolving standards for smart P&IDs [24]. However, the orchestrator may make optimized deployment decisions based on dynamic inventory discovery or other policies, which then changes the configuration originally specified in the control logic engineering tool. If an automation engineer then wants to use the engineering tools after an orchestrator has made independent decisions, they may have an outdated configuration.

Removing the control application assignment to nodes from the engineering tools and exclusively having it in an TOSCA orchestrator may be possible. However, in this case the automation engineer may perceive a lack of control over the system and reject an independently acting orchestration tool. For example, in case of errors, troubleshooting may be complicated if the engineering tools lack deployment information.

Challenge of synchronizing with container orchestrators: Besides the engineering tools, the TOSCA orchestrator may also need to synchronize with a container orchestration framework, such as Kubernetes. The latter may alter the deployment independent from the TOSCA orchestrator, e.g. draining nodes for maintenance or dynamically adjusting workloads for more balanced resource usage. Users may prefer to use popular K8s tools instead of a TOSCA orchestrator to alter the deployment. Such changes would need to be also reflected in the TOSCA orchestrator, so that future dynamic deployment decisions inside the orchestrator remain informed.

Continuously synchronizing multiple systems (i.e., engineering tools, TOSCA orchestrator, container orchestration) is inherently complex and error-prone. There may be conflicting changes coming from different sources. Each of the mentioned systems has different kinds of users (e.g., automation engineers, IT administrators, cloud providers) with different expertise and different perspectives on the system. Thus, it is unlikely that such systems can be merged into a single one, instead, a careful and continuous synchronization is needed.

TOSCA is also designed to make concrete deployment decisions, for example selecting appropriate nodes for deployment from an inventory. It is difficult to delegate concrete deployment decisions to a container orchestration system, as TOSCA does not foresee deploying a service to a group of host nodes and leaving the concrete node selection to a separate system. The TOSCA specification is still rather host node oriented, being designed when IaaS was more popular and container orchestration was not yet available.

Overshadowed by other deployment automation technologies: The OASIS TOSCA technical committee started its work in 2012 and has evolved the standard already for over 10 years. Although TOSCA includes a flexible object-oriented model independent from specific artifact technologies or cloud vendors, it has not gained wide-spread adoption in the IT industry [5], [6]. A literature review [5] collected more than a dozen of case studies applying TOSCA, but most of them were carried out in academic contexts. Wurster et al. [6] found in 2020 through a Google search that other deployment automation technologies, such as Puppet, Chef, Ansible, Kubernetes, and Terraform were by far more popular than Cloudify, which is the most popular TOSCA-based orchestrator. These technologies have slightly different assumptions and functions than TOSCA but are used in practice for similar purposes [6].

For example, Ansible is a popular infrastructure-as-code tool and configuration manager and is provided with several cloud-specific modules. For TOSCA the implementation of artifact processors is rather orchestrator-specific and there is no commercially relevant repository or marketplace with ready-made solutions. Similar to TOSCA, Ansible can be used to specify and bootstrap an IT infrastructure, furthermore it can interface with Kubernetes for application deployment and container handling. Since Kubernetes text templates are cloud-agnostic, and Kubernetes has gained vast popularity, it could even be argued that Kubernetes is the de-facto standard for portable cloud application deployment that TOSCA intended to be. TOSCA is more general though and not tied to container technologies. Furthermore, the same as Kubernetes templates, Ansible playbooks do not feature object-oriented models as in TOSCA, but also provide a declarative specification for configuration management.

Of course, all these deployment automation technologies could be used in combination, as in our case study. However, maintaining TOSCA models, Ansible playbooks, and Kubernetes templates is complex and potentially error-prone.

From the longevity perspective of distributed control systems, it could be argued that Ansible and Kubernetes are transient and vendor-specific technologies that may be hard to support over a 30-year life-span of an industrial plant, whereas TOSCA is a more long-living industry-agreed standard. However, both Ansible and Kubernetes use no proprietary binary notations to express configurations and application deployments, but simple textual notations. Therefore, it is comparably easy to write converters to migrate such specifications to future deployment automation tools, once the original ones become obsolete.

VII. Conclusions

We learned that TOSCA is a sophisticated specification that may need more maturation and fully compliant tool support to become more widely used. It was feasible to map our abstract sDCS scenario to TOSCA templates, given a simple TOSCA profile for DCS. We implemented a prototypical TOSCA orchestrator, which prompted several design decisions due to specification gaps. Specifically in the DCS application context, synchronizing a TOSCA orchestrator with other configuration and orchestration software is challenging, though not infeasible.

As part of future work, more DCS-specific concepts need to be encoded in a domain-specific TOSCA profile, for example properties of IO and network devices or deployment policies. It needs to be assessed how far the efforts for specifying TOSCA files can be reduced by partially generating them from other typical engineering artifacts. The roles of future DCS engineering systems and orchestrators need to be clarified. To be able to compose TOSCA models from different vendors as envisioned by the Open Process Automation Forum, it would be important to have open repositories curated by the Open Group. To facilitate the technology transfer into practice user-friendly modeling tools and orchestrator UIs are necessary.

References

[1] P. Lipton, D. Palma, M. Rutkowski, and D. A. Tamburri, "Tosca solves big problems in the cloud and beyond!" *IEEE cloud computing*, 2018.

[2] A. Luzar, S. Stanovnik, and M. Cankar, "Examination and comparison of tosca orchestration tools," in *European Conference on Software Architecture*. Springer, 2020, pp. 247–259.

[3] T. Binz, G. Breiter, F. Leyman, and T. Spatzier, "Portable cloud services using tosca," *IEEE Internet Computing*, vol. 16, no. 3, pp. 80–85, 2012.

[4] M. Hollender, *Collaborative process automation systems*. ISA, 2010.

[5] J. Bellendorf and Z. Á. Mann, "Specification of cloud topologies and orchestration using tosca: a survey," *Computing*, vol. 102, no. 8, pp. 1793–1815, 2020.

[6] M. Wurster, U. Breitenbücher, M. Falkenthal, C. Krieger, F. Leymann, K. Saatkamp, and J. Soldani, "The essential deployment metamodel: a systematic review of deployment automation technologies," *SICS Software-Intensive Cyber-Physical Systems*, vol. 35, no. 1, pp. 63–75, 2020.

[7] F. Li, M. Vögler, M. Claeßens, and S. Dustdar, "Towards automated iot application deployment by a cloud-based approach," in *2013 IEEE 6th international conference on service-oriented computing and applications*. IEEE, 2013, pp. 61–68.

[8] J. Cardoso, T. Binz, U. Breitenbücher, O. Kopp, and F. Leymann, "Cloud computing automation: Integrating usdl and tosca," in *International Conference on Advanced Information Systems Engineering*. Springer, 2013, pp. 1–16.

[9] M. Kostoska, I. Chorbev, and M. Gusev, "Creating portable tosca archive for iknow university management system," in *2014 Federated Conference on Computer Science and Information Systems*. IEEE, 2014, pp. 761–768.

[10] J. Wettinger, U. Breitenbücher, O. Kopp, and F. Leymann, "Streamlining devops automation for cloud applications using tosca as standardized metamodel," *Future Generation Computer Systems*, vol. 56, pp. 317–332, 2016.

[11] A. Sampaio, T. Rolim, N. C. Mendonça, and M. Cunha, "An approach for evaluating cloud application topologies based on tosca," in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. IEEE, 2016, pp. 407–414.

[12] V. Shvetcova, O. Borisenko, and M. Polischuk, "Using ansible as part of tosca orchestrator," in *2020 Ivannikov Ispras Open Conference (ISPRAS)*. IEEE, 2020, pp. 109–114.

[13] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, "Winery–a modeling tool for tosca-based cloud applications," in *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 700–704.

[14] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, and S. Wagner, "Opentosca–a runtime for tosca-based cloud applications," in *International Conference on Service-Oriented Computing*. Springer, 2013, pp. 692–695.

[15] M. Artac, T. Borovšak, E. Di Nitto, M. Guerriero, D. Perez-Palacin, and D. A. Tamburri, "Infrastructure-as-code for data-intensive architectures: A model-driven development approach," in *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 2018, pp. 156–15 609.

[16] P. Di Francesco, P. Lago, and I. Malavolta, "Architecting with microservices: A systematic mapping study," *Journal of Systems and Software*, vol. 150, pp. 77–97, 2019.

[17] J. DesLauriers, J. Kovacs, and T. Kiss, "Abstractions of abstractions: Metadata to infrastructure-as-code," in *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2022, pp. 230–232.

[18] J. Alonso, R. Piliszek, and M. Cankar, "Embracing iac through the devsecops philosophy: Concepts, challenges, and a reference framework," *IEEE Software*, 2022.

[19] M. Tiegelkamp and K.-H. John, *IEC 61131-3: Programming industrial automation systems*. Springer, 2010.

[20] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*. Springer Science & Business Media, 2009.

[21] H. Koziolek, A. Burger, M. Platenius-Mohr, and R. Jetley, "A classification framework for automated control code generation in industrial automation," *J. Syst. Softw.*, vol. 166, p. 110575, 2020. [Online]. Available: https://doi.org/10.1016/j.jss.2020.110575

[22] H. Koziolek, A. Burger, P. Abdulla, J. Rückert, S. Sonar, and P. Rodriguez, "Dynamic updates of virtual plcs deployed as kubernetes microservices," in *European Conference on Software Architecture*. Springer, 2021, pp. 3–19.

[23] B. Lundell, B. Lings, A. Persson, and A. Mattsson, "Uml model interchange in heterogeneous tool environments: An analysis of adoptions of xmi 2," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2006, pp. 619–630.

[24] H. Koziolek, A. Burger, M. Platenius-Mohr, J. Rückert, H. Abukwaik, and R. Jetley, "Rule-based code generation in industrial automation: four large-scale case studies applying the cayenne method," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, 2020, pp. 152–161.