Modelling the Behaviour of Management Operations in Cloud-based Applications*

Antonio Brogi, Andrea Canciani, Jacopo Soldani, and PengWei Wang

Department of Computer Science, University of Pisa, Italy

Abstract. How to flexibly manage complex applications over heterogeneous clouds is one of the emerging problems in the cloud era. The OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) aims at solving this problem by providing a language to describe and manage complex cloud applications in a portable, vendor-agnostic way. TOSCA permits to define an application as an orchestration of nodes, whose types can specify states, requirements, capabilities and management operations — but not how they interact each another.

In this paper we first propose how to extend TOSCA to specify the behaviour of management operations and their relations with states, requirements, and capabilities. We then illustrate how such behaviour can be naturally modelled, in a compositional way, by means of open Petri nets. The proposed modelling permits to automate different analyses, such as determining whether a deployment plan is valid, which are its effects, or which plans allow to reach certain system configurations.

1 Introduction

Available cloud technologies permit to run on-demand distributed software systems at a fraction of the cost which was necessary just a few years ago. On the other hand, how to flexibly deploy and manage such applications over heterogeneous clouds is one of the emerging problems in the cloud era.

In this perspective, OASIS recently released the *Topology and Orchestration Specification for Cloud Applications* (TOSCA [19,20]), a standard to support the automation of the deployment and management of complex cloud-based applications. TOSCA provides a modelling language to specify, in a portable and vendor-agnostic way, a cloud application and its deployment and management. An application can be specified in TOSCA by instantiating component types, by connecting a component's requirements to the capabilities of other components, and by orchestrating components' operations into plans defining the deployment and management of the whole application.

Unfortunately, the current specification of TOSCA [19] does not permit to describe the behaviour of the management operations of an application. Namely, it is not possible to describe the order in which the management operations of

* This work has been partly supported by the EU-FP7-ICT-610531 project SeaClouds.

a component must be invoked, nor how those operations depend on the requirements and affect the capabilities of that component. As a consequence, the verification of whether a plan to deploy an application is valid must be performed manually, with a time-consuming and error-prone process.

In this paper, we first propose a way to extend TOSCA to specify the behaviour of management operations and their relations with states, requirements, and capabilities. We define how to specify the management protocol of a TOSCA component by means of finite state machines whose states and transitions are associated with conditions on (some of) the component's requirements and capabilities. Intuitively speaking, those conditions define the consistency of component's states and constrain the executability of component's operations to the satisfaction of requirements.

We then illustrate how the management protocols of TOSCA components can be naturally modelled, in a compositional way, by means of open Petri nets [2,14]. This allows us to obtain the management protocol of an arbitrarily complex cloud application by combining the management protocols of its components. The proposed modelling permits to automate different analyses, such as determining whether a deployment plan is valid, which are its effects, or which plans allow to reach certain system configurations.

The rest of the paper is organized as follows. Sect. 2 introduces the needed background (TOSCA and open Petri nets), while Sect. 3 illustrates a scenario motivating the need for an explicit, machine-readable representation of management protocols. Sect. 4 describes how TOSCA can be extended to specify the behaviour of management operations, how such behaviour can be naturally and compositionally modelled by means of open Petri nets, and how the proposed modelling permits to automate different types of analysis. Related work is discussed in Sect. 5, while some concluding remarks are drawn in Sect. 6.

2 Background

2.1 TOSCA

TOSCA [19] is an emerging standard whose main goals are to enable (i) the specification of portable cloud applications and (ii) the automation of their deployment and management. In this perspective, TOSCA provides an XML-based modelling language which allows to specify the structure of a cloud application as a typed topology graph, and deployment/management tasks as plans. More precisely, each cloud application is represented as a `ServiceTemplate` (Fig. 1), which consists of a `TopologyTemplate` and (optionally) of management `Plans`.

The `TopologyTemplate` is a typed directed graph that describes the topological structure of the composite cloud application. Its nodes (`NodeTemplate`s) model the application components, while its edges (`RelationshipTemplate`s) model the relations between those application components. `NodeTemplate`s and `RelationshipTemplate`s are typed by means of `NodeType`s and `Relationship-Type`s, respectively. A `NodeType` defines (i) the observable properties of an application component C, (ii) the possible states of its instances, (iii) the requirements

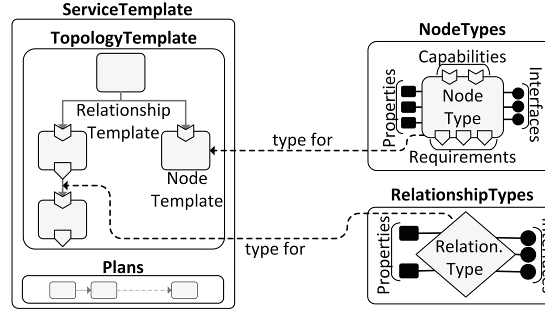


Fig. 1. TOSCA `ServiceTemplate`.

needed by C, (iv) the capabilities offered by C to satisfy other components' requirements, and (v) the management operations of C. `RelationshipTypes` describe the properties of relationships occurring among components. Syntactically, properties are described by `PropertiesDefinitions`, states by `InstanceStates`, requirements by `RequirementDefinitions` (of certain `RequirementTypes`), capabilities by `CapabilityDefinitions` (of certain `CapabilityTypes`), and operations by `Interfaces` and `Operations`.

On the other hand, `Plans` enable the description of application deployment and/or management aspects. Each `Plan` is a workflow that orchestrates the operations offered by the application components (i.e., `NodeTemplates`) to address (part of) the management of the whole cloud application[1].

2.2 (Open) Petri nets

Before providing a formal definition of open Petri nets (Def. 2), we recall the definition of Petri nets just to introduce the employed notation. We instead omit to recall other very basic notions about Petri nets (e.g., marking of a net, firing of transitions, etc.) as they are well-know and easy to find in literature [18].

Definition 1. *A* Petri net *is a tuple* $\mathcal{P} = \langle P, T, \bullet\cdot, \cdot\bullet \rangle$ *where* P *is a set of places,* T *is a set of transitions (with* $P \cap T = \varnothing$*), and* $\bullet\cdot, \cdot\bullet : T \to 2^P$ *are functions assigning to each transition its input and output places.*

According to [2], an open Petri net is an ordinary Petri net with a distinguished set of (open) places that are intended to represent the interface of the net towards the external environment, meaning that the environment can put or remove tokens from those places. In this paper, we will employ a subset of open Petri nets, where transitions consume at most one token from each place, and where the environment can both add/remove tokens to/from all open places.

Definition 2. *An* open Petri net *is a pair* $\mathcal{Z} = \langle \mathcal{P}, I \rangle$*, where* $\mathcal{P} = \langle P, T, \bullet\cdot, \cdot\bullet \rangle$ *is an ordinary Petri net, and* $I \subseteq P$ *is the set of* open *places. The places in* $P \setminus I$ *will be referred to as* internal *places.*

[1] A more detailed and self-contained introduction to TOSCA can be found in [7].

3 Motivating scenario

Consider a developer who wants to deploy and manage the web services *SendSMS* and *Forex* on a TOSCA-compliant cloud platform. She first describes her services in TOSCA, and then selects the third-party components (i.e. `NodeTypes`) needed to run them. For instance, she indicates that her services will run on a *Tomcat* server installed on an *Ubuntu* operating system, which in turn runs on an *AmazonEC2* virtual machine. Fig. 2 illustrates the resulting `TopologyTemplate`,

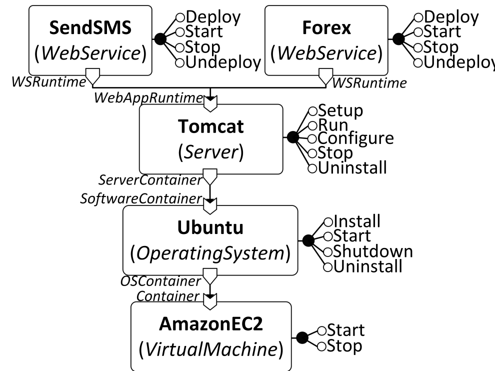
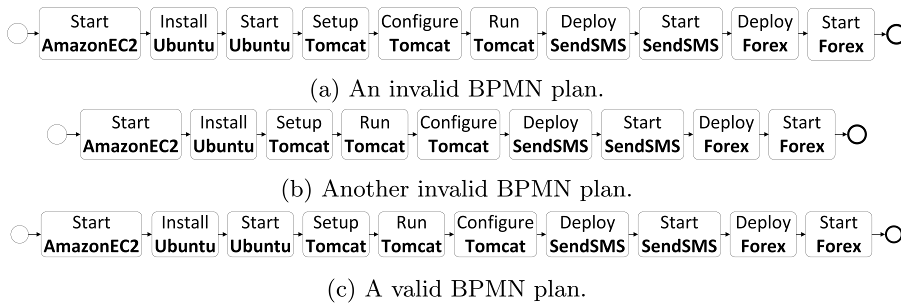


Fig. 2. Motivating scenario.

according to the Winery graphical notation [15]. For the sake of simplicity, and without loss of generality, in the following we focus only on the lifecycle interface [7] of each `NodeType` instantiated in the topology (i.e., the interface containing the operations to install, configure, start, stop, and uninstall a component).

Suppose that the developer wants to describe the automation of the deployment of the *SendSMS* and *Forex* services by writing a TOSCA `Plan`. Since TOSCA does not include any representation of the management protocols of (third-party) `NodeTypes`, developers may produce invalid `Plans`. For instance, while Fig. 3 illustrates three seemingly valid `Plans`, only the third is a valid



(a) An invalid BPMN plan.

(b) Another invalid BPMN plan.

(c) A valid BPMN plan.

Fig. 3. Deployment `Plans`.

plan. The other `Plans` cannot be considered valid since (a) *Tomcat*'s `Configure` operation cannot be executed before *Tomcat* is running, and (b) *Tomcat* cannot be installed when the *Ubuntu* operating system is not running.

While the validity of `Plans` can be manually verified, this is a time-consuming and error-prone process. In order to enable the automated verification of the validity of `Plans`, TOSCA should be extended so as to permit specifying the behaviour of and the relations among `NodeTypes`' management operations.

4 Modelling management protocols

While a TOSCA `NodeType` can be described by means of its states, requirements, capabilities, and management operations, there is currently no way to specify how management operations affect states, how operations or states depend on requirements, or which capabilities are concretely provided in a certain state.

The objective of the next section is precisely to propose a way to extend TOSCA to specify the behaviour of management operations and their relations with states, requirements, and capabilities.

4.1 Cloud-based application management protocols in TOSCA

Let N be a TOSCA `NodeType`, and let us denote its states, requirements, capabilities, and management operations with S_N, R_N, C_N, and O_N, respectively.

We want to permit describing whether and how the management operations of N depend on other operations of the same node as well as on operations of the other nodes providing the capabilities that satisfy the requirements of N.

- The first type of dependencies can be easily described by specifying the relationship between states and management operations of N. More precisely, the order with which the operations of N can be executed can be described by means of a transition relation T, that specifies whether an operation o can be executed in a state s, and which state is reached by executing o in s.
- The second type of dependencies can be described by associating transitions and states with (possibly empty) sets of requirements to indicate that the corresponding capabilities are assumed to be provided. More precisely, the requirements associated with a transition t specify which are the capabilities that must be offered by other nodes to allow the execution of t. The requirements associated with a state of a `NodeType` N specify which are the capabilities that must (continue to) be offered by other nodes in order for N to (continue to) work properly.

To complete the description, each state s of a `NodeType` N also specifies the capabilities provided by N in s.

Definition 3. *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a* `NodeType`*, where* S_N*,* R_N*,* C_N*, and* O_N *are the sets of its states, requirements, capabilities, and management operations.* $\mathcal{M}_N = \langle \overline{s}, R, C, T \rangle$ *is the* management protocol *of* N*, where*

- $\overline{s} \in S_N$ *is the initial state,*
- R *is a function indicating, for each state* $s \in S_N$*, which conditions on requirements must hold (i.e.,* $R(s) \subseteq R_N$*, with* $R(\overline{s}) = \varnothing)$[2]*,*
- C *is a function indicating which capabilities of* N *are concretely offered in a state* $s \in S_N$ *(i.e.,* $C(s) \subseteq C_N$*, with* $C(\overline{s}) = \varnothing$*), and*
- $T \subseteq S_N \times 2^{R_N} \times O_N \times S_N$ *is a set of quadruples modelling the transition relation (i.e.,* $\langle s, H, o, s' \rangle \in T$ *means that in state* s*, and if condition* H *holds,* o *is executable and leads to state* s'*).*

Syntactically, to describe $\mathcal{M}_N$ we slightly extend the syntax for describing a TOSCA `NodeType`. Namely, we enrich the description of an `InstanceState` by introducing the nested elements `ReliesOn` and `Offers`. `ReliesOn` defines R (of Def. 3) by enabling the association between states and assumed requirements, while `Offers` defines C by indicating the capabilities offered in a state. Furthermore, we introduce the element `ManagementProtocol`, which allows to specify the `InitialState` $\overline{s}$ of the protocol, as well as the `Transitions` defining the transition relation T.

The management protocols of the `NodeType`s in the motivating scenario of Sect. 3 are shown in Fig. 4, where $\mathcal{M}_{WS}$ is the management protocol for `Web-Services`, $\mathcal{M}_S$ for `Server`, $\mathcal{M}_{OS}$ for `OperatingSystem`, and $\mathcal{M}_{VM}$ for `Virtual-Machine`. Consider for instance the management protocol $\mathcal{M}_S$ of `NodeType`

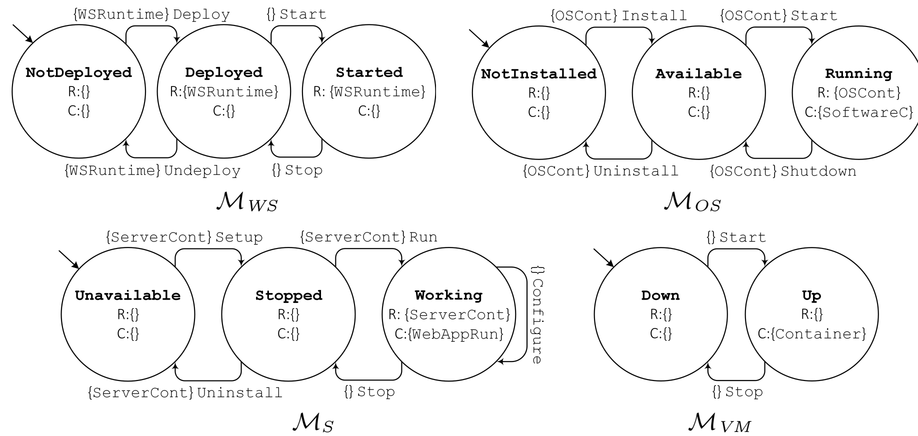


Fig. 4. Management protocols of the `NodeType`s in our motivating scenario.

`Server` defining the *Tomcat* server. Its states S_N are `Unavailable` (initial state), `Stopped`, and `Working`, the only requirement in R_N is `ServerContainer`, the only capability in C_N is `WebAppRuntime`, and its management operations are `Setup`, `Uninstall`, `Run`, `Stop`, and `Configure`. States `Unavailable` and `Stopped` are not associated with any requirement or capability. State `Working` instead

[2] Without loss of generality, we assume that the initial state of a management protocol has no requirements and does not provide any capability.

specifies that the capability corresponding to the `ServerContainer` requirement must be provided (by some other node) in order for `Server` to (continue to) work properly. State `Working` also specifies that `Server` provides the `WebAppRuntime` capability when in such state. Finally, all transitions (but those involving operations `Stop` and `Configure`) constrain their firability by requiring the capability that satisfies `ServerContainer` to be offered (by some other node).

Note that Def. 3 permits to define operations that have non-deterministic effects when applied in a state (e.g., a state can have two outgoing transitions corresponding to the same operation and leading to different states). This form of non-determinism is not acceptable in the management of a TOSCA application [7]. We will thus focus on *deterministic* management protocols, i.e. protocols ensuring deterministic effects when performing an operation in a state.

Definition 4. *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a* `NodeType`*. The management protocol* $\mathcal{M}_N = \langle \overline{s}, R, C, T \rangle$ *is* deterministic *if and only if*

$$\forall \langle s_1, H_1, o_1, s_1' \rangle, \langle s_2, H_2, o_2, s_2' \rangle \in T \colon s_1 = s_2 \wedge o_1 = o_2 \Rightarrow s_1' = s_2'$$

4.2 Modelling cloud-based application management protocols in (open) Petri nets

A (deterministic) management protocol $\mathcal{M}_N$ of a `NodeType` N can be easily encoded by an open Petri net. Each state of $\mathcal{M}_N$ is mapped into an internal place of the Petri net, and each capability and requirement of N is mapped into an open place of the same net. Furthermore, each transition $\langle s, H, o, s' \rangle$ of $\mathcal{M}_N$ is mapped into a Petri net transition t with the following inputs and outputs:

(i) The input places of t are the places denoting s, the requirements that are needed but not already available in s (i.e., $(R(s') \cup H) - R(s)$), and the capabilities that are provided in s but not in s' (i.e., $C(s) - C(s')$).
(ii) The output places of t are the places denoting s', the requirements that were needed but are no more assumed to hold in s' (i.e., $(R(s) \cup H) - R(s')$), and the capabilities that are provided in s' but not in s (i.e., $C(s') - C(s)$).

The initial marking of the obtained net prescribes that the only place initially containing a token is that corresponding to the initial state $\overline{s}$ of $\mathcal{M}_N$.

Definition 5. *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a* `NodeType`*, with* $\mathcal{M}_N = \langle \overline{s}, R, C, T \rangle$*. The management protocol* $\mathcal{M}_N$ *is encoded into an open net* $\mathcal{Z}_N = \langle \mathcal{P}_N, I_N \rangle$*, with* $\mathcal{P}_N = \langle P_N, T_N, \bullet\cdot, \cdot\bullet \rangle$ *and* $I_N \subseteq P_N$*, as follows.*

- *The set* P_N *of places contains a separate place for each state in* S_N*, for each requirement in* R_N*, and for each capability in* C_N*.*
- *The set* $I_N \subset P_N$ *of open places contains the places denoting the requirements in* R_N *and the capabilities in* C_N*.*
- *The set* T_N *contains a net transition* t *for each transition* $\langle s, H, o, s' \rangle \in T$*.*
  (i) The set $\bullet t$ *of input places contains the place* s*, the places denoting the requirements in* $(R(s') \cup H) - R(s)$*, and those denoting the capabilities in* $C(s) - C(s')$*.*

(ii) The set $t\bullet$ of output places contains the place s', the places denoting the requirements in $(R(s) \cup H) - R(s')$, and those denoting the capabilities in $C(s') - C(s)$.

The initial marking of $\mathcal{Z}_N$ consists of only one token in place $\overline{s}$.

The above definition ensures that the Petri net encoding of a management protocol satisfies the following properties:

- There is a one-to-one correspondence between the marking of the internal places of the Petri net and the states of a management protocol. Namely, there is exactly one token in the internal place denoting the current state, and no tokens in the other internal places.
- Each operation can be performed if and only if all the necessary requirements are available in the source state, and no capability required by any connected component is disabled in the target state.

Consider for instance the management protocol $\mathcal{M}_S$ (Fig. 4), whose corresponding Petri net is shown in Fig. 5. Each state in $\mathcal{M}_S$ is translated into an internal place (represented as a circle), while the `ServerContainer` requirement and the `WebAppRuntime` capability are translated into open places (represented as diamonds). Additionally, protocol transitions are translated into net transitions. For example, the transition $\langle$`Stopped`, {`ServerContainer`}, `Run`, `Working`$\rangle$ is translated into a Petri net transition, whose inputs places are `Stopped` and `ServerContainer`, and whose outputs places are `Working` and `WebAppRuntime`.

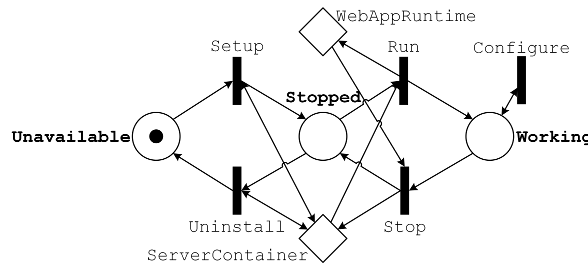


Fig. 5. Example of Petri net translation.

4.3 Analysis of cloud-based application management protocols

We now show how the Petri net modelling the management protocol of a TOSCA `TopologyTemplate` (specifying a whole cloud-based application) can be obtained, in a compositional way, from the Petri nets modelling the management protocols of the `NodeTypes` in such `TopologyTemplate`.

We first need to model (by open Petri nets working as a *capability controllers*) the `RelationshipTemplates` that define in a `TopologyTemplate` the

association between the requirements of a `NodeTypes` and the capabilities of other `NodeTypes`. To do that, we first define an utility *binding* function that returns the set of requirements with which a capability is associated.

Definition 6. *Let S be a* `ServiceTemplate`*, and let c be a capability offered by a* `NodeType` *in S. We define $b(c, S) = \{r_1, \ldots, r_n\}$, where $r_1, \ldots, r_n$ are the requirements connected to c in S by means of* `RelationshipTemplate`*s.*

We now exploit function b to define *capability controllers*. On the one hand, the controller must ensure that once a capability c is available, the nodes exposing the connected requirements $r_1, \ldots, r_n$ are able to simultaneously exploit it. This is obtained by adding a transition $c_\uparrow$ able to propagate the token from place c to places $r_1, \ldots, r_n$ (i.e., the input place of $c_\uparrow$ is c, and its output places are $r_1, \ldots, r_n$). On the other hand, the controller has also to ensure that the capability is not removed while at least another node is actively assuming its availability (with a condition on a connected requirement). Thus, we introduce a transition $c_\downarrow$ whose input places are $r_1, \ldots, r_n$ and whose output place is c.

Definition 7. *Let S be a* `ServiceTemplate`*, and let c be a capability offered by a* `NodeType` *instantiated in S. Let $r_1, \ldots, r_n$ be the requirements exposed by the nodes in S such that $b(c, S) = \{r_1, \ldots, r_n\}$. The* controller *of c is an open Petri net $\mathcal{Z}_c = \langle \mathcal{P}_c, I_c \rangle$, with $\mathcal{P}_c = \langle P_c, T_c, \bullet\cdot, \cdot\bullet \rangle$, defined as follows.*

- *The set P_c of places contains a separate place for the capability c and for each requirement $r_1, \ldots, r_n$.*
- *The set I_c coincides with P_c.*
- *The set T_c contains only two Petri net transitions $c_\uparrow$ and $c_\downarrow$.*
  - *The input and output places of $c_\uparrow$ are the place c and the places $r_1, \ldots, r_n$, respectively (i.e., $\bullet c_\uparrow = \{c\}$ and $c_\uparrow\bullet = \{r_1, \ldots, r_n\}$).*
  - *The input and output places of $c_\downarrow$ are the places $r_1, \ldots, r_n$ and the place c, respectively (i.e., $\bullet c_\uparrow = \{r_1, \ldots, r_n\}$ and $c_\uparrow\bullet = \{c\}$).*

The initial marking of $\mathcal{Z}_c$ is empty (i.e., no place contains a token).

An example of controller (for a capability c connected to two requirements r_1 and r_2) is illustrated in Fig. 6.

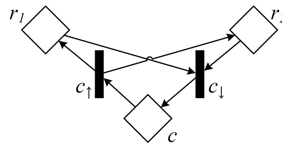


Fig. 6. Example of *capability controller.*

We can now compose the nets modelling the management protocols of the `NodeTypes` instantiated in a `ServiceTemplate`'s topology by interconnecting them with the above introduced controllers. The composition is quite simple: We just collapse the open places corresponding to the same requirements/capabilities.

Definition 8. *Let S be a* `ServiceTemplate`*. We encode S with an open Petri net $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, where $\mathcal{P}_S = \langle P_S, T_S, \bullet\cdot, \cdot\bullet \rangle$, as follows.*

- *For each node N in the topology of S, we encode its management protocol with an open Petri net $\mathcal{Z}_N$ obtained as shown in Def. 5.*
- *For each capability c exposed by a* `NodeTemplate` *in S, we create an open Petri net $\mathcal{Z}_c$ (acting as its controller) as shown in Def. 7.*
- *We then compose the above mentioned nets by taking their disjoint union and merging the places denoting the same requirement r or capability c.*

The initial marking of $\mathcal{Z}_S$ is the union of the markings of the collapsed nets.

For example, Fig. 7 shows the net obtained for the motivating scenario in Sect. 3.

The Petri net encoding of the management of a `ServiceTemplate` S, permits us defining what is a *valid plan* according to such management. Essentially, thanks to the encoding of capability controllers and to the way we compose these controllers with management protocol encodings, the obtained net ensures that no requirement can be assumed to hold if the corresponding capability is not provided, and that no capability can be removed if at least one of the corresponding requirements is assumed to hold. This permits to consider a plan valid if and only if it corresponds to a firing sequence in the net encoding of S.

Definition 9. *Let S be a* `ServiceTemplate` *and let $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, with $\mathcal{P}_S = \langle P_S, T_S, \bullet\cdot, \cdot\bullet \rangle$, be the Petri net encoding of S. A sequential plan*[3] *$o_1 o_2 \dots o_m$ is* valid *if and only if there is a firing sequence $t_1 t_2 \dots t_n$ in $\mathcal{Z}_S$ from the initial marking such that $o_1 \cdot o_2 \cdot \dots \cdot o_m = \lambda(t_1) \cdot \lambda(t_2) \cdot \dots \cdot \lambda(t_n)$, where $\cdot$ indicates the concatenation operator*[4] *and:*

$$\lambda(t) = \begin{cases} \epsilon & \text{if } t \text{ denotes a } c_\uparrow \text{ or } c_\downarrow \text{transition} \\ o & \text{if } t \text{ denotes a management protocol transition } \langle s, H, o, s' \rangle \end{cases}$$

It is easy to see now that plan (c) of Fig. 3 is valid since, for instance,

> *AmazonEC2*:`Start` `Container`$_\uparrow$ *Ubuntu*:`Install` *Ubuntu*:`Start` `SoftwareContainer`$_\uparrow$ *Tomcat*:`Setup` *Tomcat*:`Run` *Tomcat*:`Configure` `WebAppRuntime`$_\uparrow$ *SendSMS*:`Deploy` *SendSMS*:`Start` *Forex*:`Deploy` *Forex*:`Start`

is a corresponding firing sequence for the Petri net in Fig. 7. Conversely, plans (a) and (b) in Fig. 3 are not valid as there are no corresponding firing sequences. Intuitively speaking, (a) is not valid since after firing, for instance,

> *AmazonEC2*:`Start` `Container`$_\uparrow$ *Ubuntu*:`Install` *Ubuntu*:`Start` `SoftwareContainer`$_\uparrow$ *Tomcat*:`Setup`

[3] In Def. 9 we consider sequential plans. A workflow plan is valid if and only if all its sequential traces are valid.

[4] The empty string ϵ is the neutral element of $\cdot$, hence controllers' net transitions are ignored (as $\lambda(t) = \epsilon$ when t denotes a $c_\uparrow$ or $c_\downarrow$ transition).

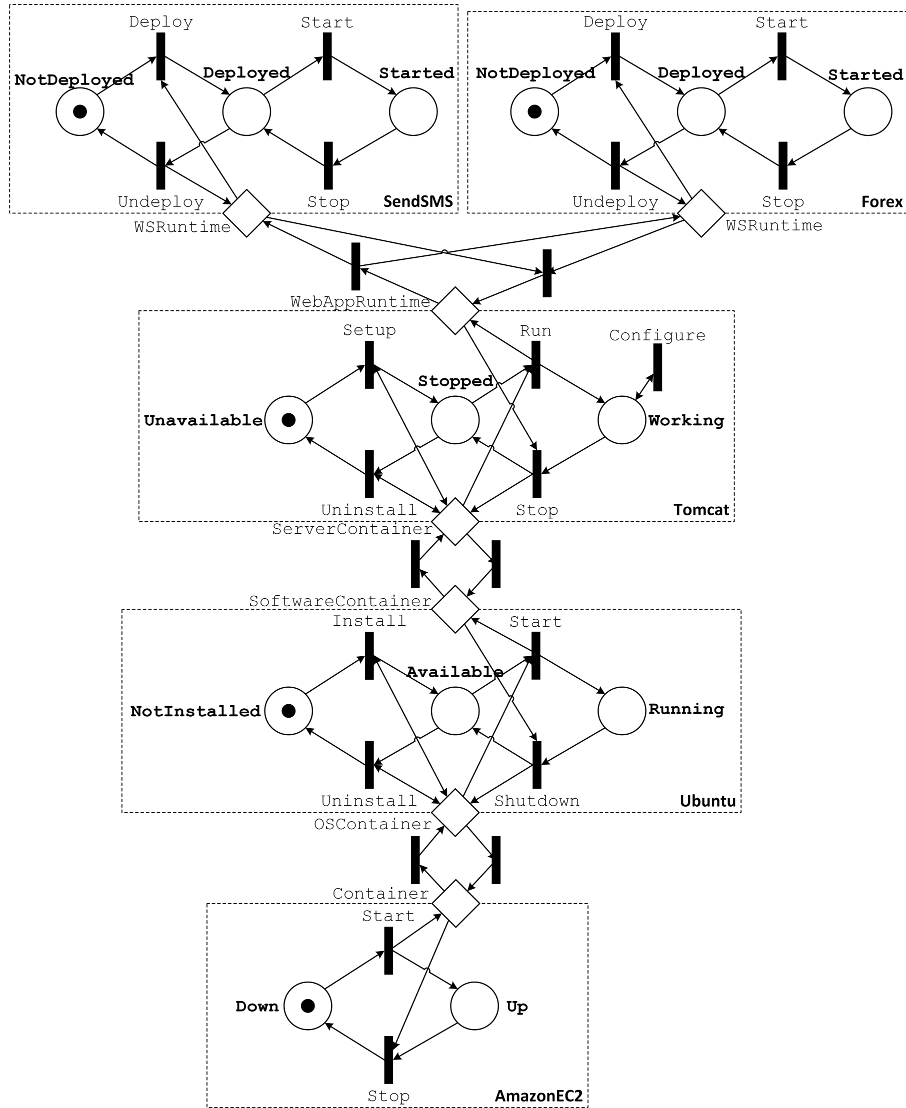


Fig. 7. Petri net encoding for the motivating scenario in Sect. 3.

transition *Tomcat*:`Configure` cannot be fired. It indeed requires a token in the `Working` place, but that place is empty and it is not possible to add tokens to it without firing *Tomcat*:`Run`. On the other hand, (b) is not valid since after firing

AmazonEC2:`Start` $\texttt{Container}_{\uparrow}$ *Ubuntu*:`Install`

transition *Tomcat*:`Setup` cannot fire. It requires a token in the place denoting the `ServerContainer` requirement, but that place is empty and it is not possible to add tokens to it without firing $\texttt{SoftwareContainer}_{\uparrow}$, which in turn cannot fire as it misses a token in the place denoting the *Ubuntu*'s `SoftwareContainer` capability (and no token can be added to such place without firing *Ubuntu*:`Start`).

It is important to observe that the correspondence between firing sequences and valid plans can be exploited for many other purposes besides checking plans' validity. The effects of a plan on the states of the components of a TOSCA `ServiceTemplate`, as well as on the requirements that are satisfied and the capabilities that are available, can be directly determined from the marking that is reached performing the corresponding firing sequence. Additionally, various classical notions in the Petri net context assume a specific meaning in the context of TOSCA applications. For example the problem of finding whether there is a plan which achieves a specific goal (e.g., bringing some components of an application to specific states or making some capabilities available) can be reduced in a straightforward way to the coverability problem [18] on the associated Petri net. Moreover, it is possible to consider as initial marking any other (reachable) marking so as to analyse maintenance plans (starting from non-initial states) besides deployment plans. Obviously, the very same properties and techniques also apply in this case. Finally, the Petri net is *reversible* [18] if and only if it is always possible to (softly) reset the application. This is a very convenient property, because it guarantees that it is always possible to generate a plan for any reachable goal from any application state.

5 Related work

Automating application management is a well-known problem in computer science. With the advent of cloud computing, it has become even more prominent because of the complexity of both applications and platforms [8]. This is witnessed by the proliferation of so-called *configuration management systems*, like Chef [9] or Puppet [21]. These systems provide a domain-specific language to model the desired configuration for a machine and employ a client-server model in which a server holds the model and the client ensures this configuration is met. However, the lack of a machine readable representation of management protocols of application components inhibits the possibility of automating verification on components' configurations and dependencies.

A large body of research has been devoted to model interacting systems by means of finite state machines, Petri nets, and other formal models [4,11]. Because of space limitations, we discuss next only the work more closely related to ours, tailored to to model the behaviour of cloud application management.

A first attempt to master the complexity of the cloud is given by the Aeolus component model [10]. The Aeolus model is specifically designed to describe several characteristics of cloud application components (e.g., dependencies, non-functional requirements, etc.), as well as the fact that component interfaces might vary depending on the internal component state. However, the model only allows to specify what is offered and required in a state. Our approach instead allows developers to clearly separate the requirements ensuring the consistency of a state from those constraining the applicability of a management operation. This allows developers to easily express transitions where requirements are affecting only the applicability of an operation and not the consistency of a state (e.g., the transition $\langle \texttt{Unavailable}, \{\texttt{ServerContainer}\}, \texttt{Setup}, \texttt{Stopped} \rangle$ of the management protocol $\mathcal{M}_S$ in Fig. 4). Such a kind of transitions cannot be easily modelled in Aelous. Furthermore, Aelous and other emerging solutions like Juju [13] and Engage [12], differ from our approach since they are geared towards the deployment of cloud applications, thus not including also their maintenance. Additionally, Aelous, Juju, and Engage are currently not integrated with any cloud interoperability standard, thus limiting their applicability to only some supported cloud platforms. Our approach, instead, intends to model the entire lifecycle of a cloud application component, and achieves cloud interoperability by relying on the TOSCA standard [19].

To this end, TOSCA offers a rich type system permitting to match, adapt and reuse existing solutions [7]. Since our proposal extends this type system, it can also be exploited to refine existing reuse techniques, like [5,6,22]. Currently, these techniques are matchmaking and adapting (fragments of) existing `ServiceTemplate`s to implement a desired `NodeType` by checking whether the features of the latter are all offered by the former. To overcome syntactic differences, ontologies may be employed to check whether two different names are denoting the same concept. However, these techniques are behaviour-unaware: There is no way to determine whether the behaviour of the identified (fragment of) `ServiceTemplate` is coherent with that of the desired `NodeType`. Since our approach permits describing the behaviour of management operations, it can be exploited to extend the aforementioned techniques to become behaviour-aware.

It is also worth highlighting that we could directly compose the finite state machines specifying management protocols, and model valid plans as the language accepted by the composite finite state machine. However, the size of the latter grows exponentially with the number of application components. This results in a high computational complexity, even if we exploit composition-oriented automata (e.g., *interface automata* [1]). On the other hand, with open Petri nets [2,14], we have a very simple composition approach, and the exponential growth only affects the amount of reachable markings (instead of the size of the net). A simpler composition approach is even more convenient since cloud applications can change over time. For instance, to add another web service to our motivating scenario, our approach just requires to add the open Petri net encoding its management protocol, and to connect the open places denoting its requirement with the corresponding $c_{\uparrow}$ and $c_{\downarrow}$ transitions. On the other hand, with an au-

tomata based approach, the composition would be much harder, as it requires to compute the Cartesian product of the automatons' states.

6 Conclusions

In this paper we have proposed an extension of TOSCA that permits to specify the behaviour of management operations of cloud-based applications, and their relations with states, requirements, and capabilities. We have then shown how the management protocols of TOSCA components can be naturally modelled, in a compositional way, by means of open Petri nets, and that such modelling permits to automate different analyses, such as determining whether a plan is valid, which are its effects, or which plans allow to reach certain system configurations.

Please note that, while some of those Petri-net analyses have an exponential time complexity in the worst case, they still constitute a significant improvement with respect to the state of the art, where the validity of deployment plans can be verified only manually, after delving through the documentation of application components. Please also note that our approach builds on top of, but is not limited to, TOSCA. It can be easily adapted to other stateful behaviour models of systems that describe states, requirements, capabilities, and operations.

We see different possible extensions of our work. We are currently working on a prototype implementation of our approach, which includes a graphical user interface to support the definition of valid TOSCA specifications that include management protocols. The graphical user interface will compile the management protocols of a TOSCA application into a PNML file [3], hence enabling to plug-in different PNML processing environments (e.g., LoLa, ProM, or WoPeD, just to mention some) to implement the analyses described in Sect. 4.3. We also intend to improve the efficiency of the analyses by reducing the complexity of the nets that is due to the $c_\uparrow$ and $c_\downarrow$ transitions introduced by the controllers. Indeed the net encoding of a cloud application can be simplified by "folding" the controllers' transitions (by modifying the transitions whose input/output places contain a place representing capability c so that they are replaced by the places representing the requirements connected to c by means of `RelationshipTemplates`). Another interesting direction for future work is to investigate the applicability of more sophisticated fault diagnosis analyses (like [16,17]) to identify the reasons why a plan may not be valid (besides just showing the points in which a plan may get stuck, as we currently do). Finally, we want to extend the matchmaking and adaptation techniques we previously proposed [5,6,22] by including the behaviour information coming from management protocols (as illustrated in Sect. 5).

References

1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of the 8th European Software Engineering Conference / 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 109–120. ESEC/FSE-9, ACM (2001)

2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Mathematical Structures in Computer Science 15(01), 1–35 (2005)
3. Billington, J., Christensen, S., Van Hee, K., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, technology, and tools. In: Proceedings of the 24th International Conference on Applications and Theory of Petri Nets. pp. 483–505. ICATPN'03, Springer (2003)
4. Bochmann, G.V., Sunshine, C.A.: A survey of formal methods. In: Computer Network Architectures and Protocols, pp. 561–578. Applications of Communications Theory, Springer (1982)
5. Brogi, A., Soldani, J.: Matching cloud services with TOSCA. In: Advances in Service-Oriented and Cloud Computing, CCIS, vol. 393, pp. 218–232. Springer (2013)
6. Brogi, A., Soldani, J.: Reusing cloud-based services with TOSCA. In: INFORMATIK 2014. LNI, vol. 232, pp. 235–246. Gesellschaft für Informatik (GI) (2014)
7. Brogi, A., Soldani, J., Wang, P.: TOSCA in a Nutshell: Promises and Perspectives. In: Villari, M., Zimmermann, W., Lau, K.K. (eds.) Service-Oriented and Cloud Computing. LNCS, vol. 8745, pp. 171–186. Springer (2014)
8. Buyya, R., Yeo, C.S., Venugopal, S., Broberg, J., Brandic, I.: Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. Future Generation Computer Systems 25(6), 599 – 616 (2009)
9. Chef: Opscode. https://www.opscode.com/chef
10. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: A component model for the cloud. Information and Computation 239(0), 100 – 121 (2014)
11. Diaz, M.: Modeling and analysis of communication and cooperation protocols using Petri net based models. Computer Networks 6(6), 419 – 441 (1982)
12. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: A deployment management system. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 263–274. PLDI '12, ACM (2012)
13. Juju: DevOps distilled. https://juju.ubuntu.com
14. Kindler, E.: A compositional partial order semantics for Petri net components. In: Proceedings of ICATPN '97. pp. 235–252. Springer-Verlag (1997)
15. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery – Modeling Tool for TOSCA-based Cloud Applications. In: Proceedings of the 11th International Conference on Service-Oriented Computing. Springer (2013)
16. Lohmann, N.: Why does my service have no partners? In: Bruni, R., Wolf, K. (eds.) Web Services and Formal Methods, LNCS, vol. 5387, pp. 191–206. Springer (2009)
17. Lohmann, N., Fahland, D.: Where did I go wrong? In: Business Process Management, LNCS, vol. 8659, pp. 283–300. Springer (2014)
18. Murata, T.: Petri nets: Properties, analysis and applications. Proceedings of the IEEE 77(4), 541–580 (1989)
19. OASIS: Topology and Orchestration Specification for Cloud Applications. http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf (2013)
20. OASIS: TOSCA Simple Profile in YAML. http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf (2014)
21. Puppet: Puppet labs. https://puppetlabs.com
22. Soldani, J., Binz, T., Breitenbücher, U., Leymann, F., Brogi, A.: TOSCA-MART: A method for adapting and reusing cloud applications. Tech. Rep., University of Pisa (March 2015), http://eprints.adm.unipi.it/2331/1/SBBLB15_-_TR.pdf