# A Petri Net-Based Approach to Model and Analyze the Management of Cloud Applications

Antonio Brogi[1], Andrea Canciani[1], Jacopo Soldani[1(✉)], and PengWei Wang[2]

[1] Department of Computer Science, University of Pisa, Pisa, Italy
soldani@di.unipi.it
[2] School of Computer Science and Technology,
Donghua University, Shanghai, China

**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 [24,25]), 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,

This work is an extended version of [8]. It has been partly supported by the EU-FP7-ICT-610531 project SeaClouds.


M. Koutny et al. (Eds.): ToPNoC XI, LNCS 9930, pp. 28–48, 2016.
DOI: 10.1007/978-3-662-53401-4_2

and by orchestrating components' operations into plans defining the deployment and management of the whole application.

Unfortunately, the current specification of TOSCA [24] 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 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 [2,18]. 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. Section 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. Section 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 [24] 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 (`NodeTemplates`)

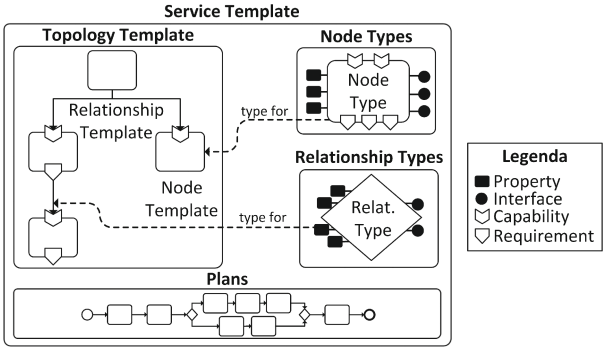



**Fig. 1.** TOSCA `ServiceTemplate`.

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 `RelationshipType`s, respectively. A `NodeType` defines (i) the observable properties of an application component $C$, (ii) the possible states of its instances, (iii) the requirements needed by $C$, (iv) the capabilities offered by $C$ to satisfy other components' requirements, and (v) the management operations of $C$. `RelationshipType`s describe the properties of relationships occurring among components.

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., `NodeTemplate`s) 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 (Definition 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 [23].

**Definition 1.** *A* Petri net *is a tuple* $\mathcal{P} = \langle P, T, {}^\bullet\cdot, \cdot^\bullet, M_0 \rangle$ *where* $P$ *is a set of places,* $T$ *is a set of transitions (with* $P \cap T = \varnothing$*),* ${}^\bullet\cdot, \cdot^\bullet : T \to 2^P$ *are functions assigning to each transition its input and output places, and* $M_0 : P \to \mathbb{N}$ *is the initial marking of* $\mathcal{P}$*.*

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

[1] A more detailed and self-contained introduction to TOSCA can be found in [10].

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* $\mathcal{P} = \langle P, T, \bullet\cdot, \cdot\bullet, M_0 \rangle$ *is an ordinary Petri net, and* $I \subseteq P$ *is the set of* open *places. The places in* $P \backslash I$ *will be referred to as* internal *places.*

## 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. Figure 2 illustrates the resulting `TopologyTemplate`, according to the Winery graphical notation [19]. For the sake of simplicity, and without loss of generality, in the following we focus only on the lifecycle interface [10] of each `NodeType` instantiated in the topology (i.e., the interface containing the operations to install, configure, start, stop, and uninstall a component).

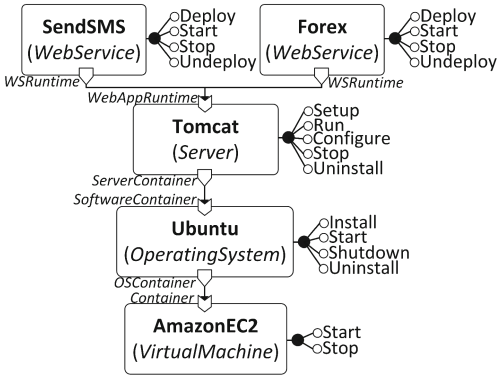
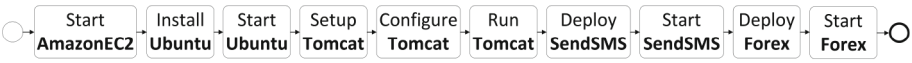



**Fig. 2.** Motivating scenario.

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 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




(a) An invalid BPMN plan.


(b) Another invalid BPMN plan.


(c) A valid BPMN plan.

**Fig. 3.** Deployment `Plan`s.

validity of `Plan`s, TOSCA should be extended so as to permit specifying the behaviour of and the relations among `NodeType`s' 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 behavior of management operations and their relations with states, requirements, and capabilities.

### 4.1 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 $\tau$, 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.* $\mathcal{M}_N = \langle \overline{s}, \rho, \gamma, \tau \rangle$ *is the* management protocol *of* $N$*, where*

- $\overline{s} \in S_N$ *is the initial state,*
- $\rho$ *is a function indicating, for each state* $s \in S_N$*, which conditions on requirements must hold (i.e.,* $\rho(s) \subseteq R_N$*, with* $\rho(\overline{s}) = \varnothing$*)*[2]*,*
- $\gamma$ *is a function indicating which capabilities of* $N$ *are concretely offered in a state* $s \in S_N$ *(i.e.,* $\gamma(s) \subseteq C_N$*, with* $\gamma(\overline{s}) = \varnothing$*), and*
- $\tau \subseteq S_N \times 2^{R_N} \times O_N \times S_N$ *is a set of quadruples modelling the transition relation (i.e.,* $\langle s, H, o, s' \rangle \in \tau$ *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[3] for describing a TOSCA `NodeType`. Namely, we enrich the description of an instance state by introducing the nested elements `ReliesOn` and `Offers`. `ReliesOn` defines $\rho$ (of Definition 3) by enabling the association between states and assumed requirements, while `Offers` defines $\gamma$ 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 $\tau$.

The management protocols of the `NodeTypes` in the motivating scenario of Sect. 3 are shown in Fig. 4, where $\mathcal{M}_{WS}$ is the management protocol for `WebServices`, $\mathcal{M}_S$ for `Server`, $\mathcal{M}_{OS}$ for `OperatingSystem`, and $\mathcal{M}_{VM}$ for `VirtualMachine`. Consider for instance the management protocol $\mathcal{M}_S$ of `NodeType 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 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).

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

[3] A more detailed syntax for extended `NodeTypes` can be found in [7].

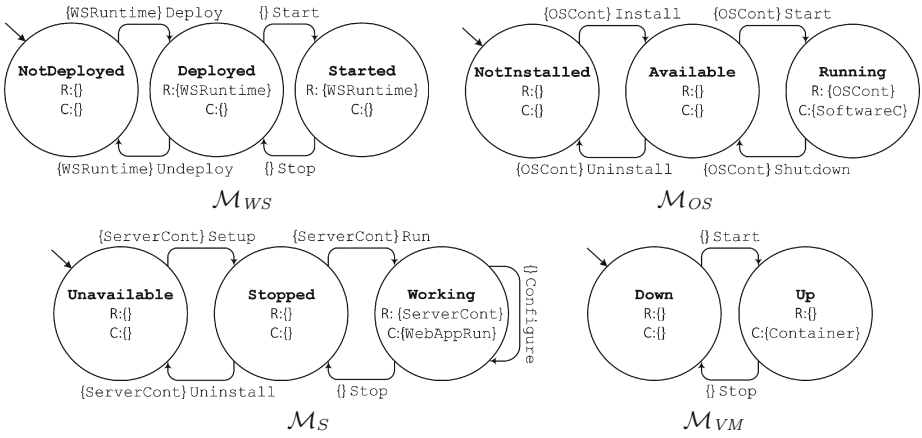



**Fig. 4.** Management protocols of the `NodeType`s in our motivating scenario.

Note that Definition 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 [10]. 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* $\mathcal{M}_N = \langle \overline{s}, \rho, \gamma, \tau \rangle$ *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 Encoding Management Protocols in 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., $(\rho(s') \cup H) - \rho(s)$), and the capabilities that are provided in $s$ but not in $s'$ (i.e., $\gamma(s) - \gamma(s')$).
(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., $(\rho(s) \cup H) - \rho(s')$), and the capabilities that are provided in $s'$ but not in $s$ (i.e., $\gamma(s') - \gamma(s)$).

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 $\mathcal{M}_N$.

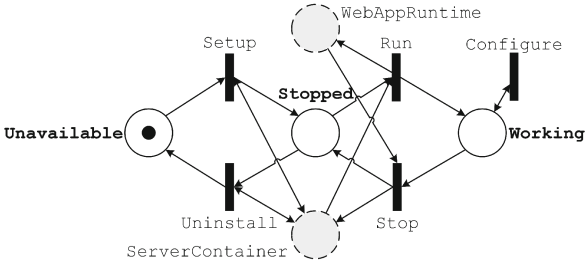



**Fig. 5.** Example of Petri net translation.

**Definition 5.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a* `NodeType`*, with* $\mathcal{M}_N = \langle \overline{s}, \rho, \gamma, \tau \rangle$*. The* management protocol $\mathcal{M}_N$ is encoded into an open Petri net $\mathcal{Z}_N = \langle \mathcal{P}_N, I_N \rangle$*, with* $\mathcal{P}_N = \langle P_N, T_N, \bullet\cdot, \cdot\bullet, M_0 \rangle$ *and* $I_N \subseteq P_N$*, as follows.*

- $P_N = S_N \cup R_N \cup C_N$*, i.e. 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* $C_N$*.*
- $I_N = R_N \cup C_N$*, i.e. the set* $I_N \subset P_N$ *of open places contains the places denoting the requirements in* $R_N$ *and the capabilities in* $C_N$*.*
- $T_N = \tau$ *(i.e., the set* $T_N$ *contains a net transition* $t$ *for each transition* $\langle s, H, o, s' \rangle \in \tau$*), and* $\forall t = \langle s, H, o, s' \rangle \in T_N$
  - *(i)* $\bullet t = \{s\} \cup ((\rho(s') \cup H) - \rho(s)) \cup (\gamma(s) - \gamma(s'))$*, i.e. the set* $\bullet t$ *of input places contains the place* $s$*, the places denoting the requirements in* $(\rho(s') \cup H) - \rho(s)$*, and those denoting the capabilities in* $\gamma(s) - \gamma(s')$*.*
  - *(ii)* $t\bullet = \{s'\} \cup ((\rho(s) \cup H) - \rho(s')) \cup (\gamma(s') - \gamma(s))$*, i.e. the set* $t\bullet$ *of output places contains the place* $s'$*, the places denoting the requirements in* $(\rho(s) \cup H) - \rho(s')$*, and those denoting the capabilities in* $\gamma(s') - \gamma(s)$*.*
- *The initial marking* $M_0$ *of* $\mathcal{Z}_N$ *is defined as follows:*

$$\forall p \in P_N . M_0(p) = \begin{cases} 1 & \text{if } p \text{ denotes } \overline{s} \\ 0 & \text{otherwise} \end{cases}$$

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 dashed circles). Additionally, protocol transitions are translated into net transitions. For example, the transition ⟨`Stopped`,{`ServerContainer`}, `Run`, `Working`⟩ is translated into a Petri net transition, whose inputs places are `Stopped` and `ServerContainer`, and whose outputs places are `Working` and `WebAppRuntime`.

### 4.3 Modelling the Management of a `ServiceTemplate`

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 `NodeType`s in such `TopologyTemplate`.

We first need to model (by open Petri nets working as a *capability controllers*) the `Relationship-Template`s that define in a `TopologyTemplate` the association between the requirements of a `NodeType`s and the capabilities of other `NodeType`s. 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 $\mathcal{P}_c = \langle P_c, T_c, \bullet\cdot, \cdot\bullet, M_0 \rangle$, defined as follows.*

- *The set $P_c$ of places contains a separate place for the capability $c$ and for each requirement $r_1, \ldots, r_n$. It also contains a place $r_c$ that witnesses the availability of the capability $c$.*
- *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 $r_1, \ldots, r_n$ and $r_c$, respectively (i.e., $\bullet c_\uparrow = \{c\}$ and $c_\uparrow\bullet = \{r_1, \ldots, r_n\} \cup \{r_c\}$).*
  - *The input and output places of $c_\downarrow$ are the places $r_1, \ldots, r_n$ and $r_c$, and the place $c$, respectively (i.e., $\bullet c_\uparrow = \{r_1, \ldots, r_n\} \cup \{r_c\}$ and $c_\uparrow\bullet = \{c\}$).*
- *The initial marking $M_0$ of $\mathcal{Z}_c$ is $\forall p \in P_c . M_0(p) = 0$.*

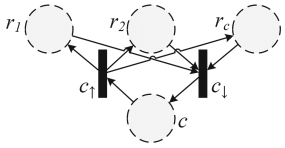



**Fig. 6.** Example of *capability controller*.

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

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 $\mathcal{P}_S = \langle P_S, T_S, \bullet\cdot, \cdot\bullet, M_0 \rangle$, 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 Definition 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 Definition 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 $M_0$ 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. For the sake of readability, in the figure we omit, for each capability $c$, the place $r_c$ of its controller.

A very convenient property of the obtained encoding is that it is *safe* (i.e., the number of tokens in each place does not exceed one, for any marking $M$ that is reachable from the initial marking $M_0$ [23]). To prove it, we need to further characterize the Petri net encoding we provided through Definitions 5, 7 and 8.

*Property 1.* Let $S$ be a `ServiceTemplate`, and let $\mathcal{Z}_S$ be its Petri net encoding.

$$\mathcal{Z}_S \textit{ is safe.}$$

*Proof.* The property follows from the properties (i), (ii), and (iii) shown in Lemma 1 (see Appendix). More precisely, (i) proves that the internal places denoting node states can contain at most one token, (ii) proves that each open place denoting a capability $c$ (as well as the corresponding place $r_c$) can contain at most one token, and (iii) proves that each open place denoting a requirement can contain at most one token. Therefore, all places in $\mathcal{Z}_S$ can contain at most one token (in any reachable marking), thus making the whole net safe [23]. □

### 4.4 Analyzing the Management of a `ServiceTemplate`

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 $\mathcal{P}_S = \langle P_S, T_S, \bullet\cdot, \cdot\bullet, M_0 \rangle$, be the Petri net encoding of $S$. A sequence $o_1 o_2 ... o_m$ of management operations is a* valid sequential plan *for $S$ if and only if there is a firing sequence $t_1 t_2 \dots t_n$ (with $t_i \in T_S$) from the initial marking $M_0$ such that*

$$o_1 \cdot o_2 \cdot \ldots \cdot o_m = \lambda(t_1) \cdot \lambda(t_2) \cdot \ldots \cdot \lambda(t_n),$$

*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`

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` `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 `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`).

We can easily extend the definition of validity from sequential plans to generic workflow `Plan`s, by constraining all their sequential traces to be valid.

[4] The empty string $\epsilon$ 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).

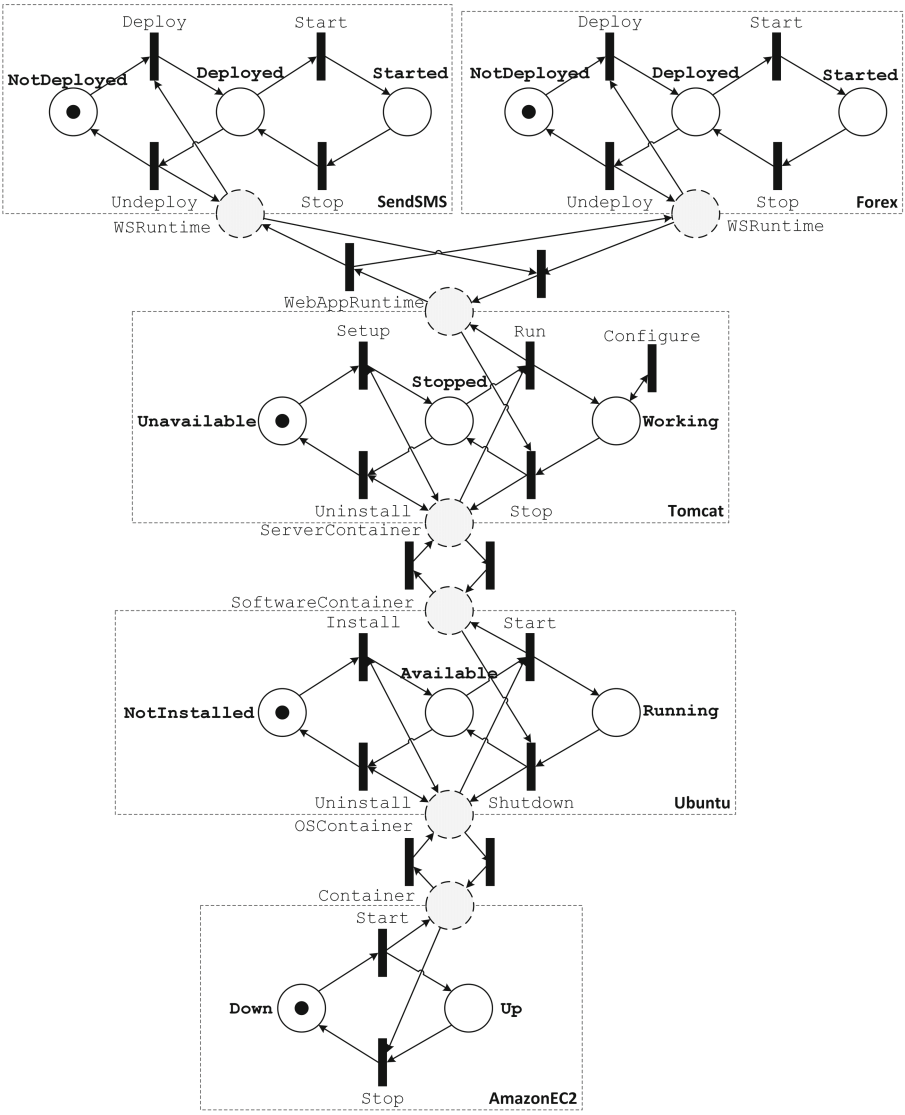



**Fig. 7.** Petri net encoding for the motivating scenario in Sect. 3.

**Definition 10.** *Let $S$ be a* `ServiceTemplate`*, and let $\mathcal{Z}_S$ be its Petri net encoding. A workflow* `Plan` *$P$ is* valid *for $S$ if and only if all its sequential traces are valid sequential plans for $S$ (see Definition 9).*

However, the above Definition 10 does not ensure that all traces end up in the same setting of the `ServiceTemplate`. Two different traces can reach two different markings with a different token assignment for the internal places. This

would mean that, by differently inter-leaving the activities in a workflow `Plan`, the nodes in a `ServiceTemplate` can end up in different states (thus potentially activating different capabilities and assuming different requirements). This is not acceptable in the management of a TOSCA application, as we would expect a `Plan` to have deterministic effects (independently of the inter-leaving of the activities that compose such `Plan`). We thus define the notion of *deterministic* `Plan`s, after introducing that of internally equivalent markings.

**Definition 11.** *Let $\mathcal{Z} = \langle \mathcal{P}, I \rangle$, with $\mathcal{P} = \langle P, T, {}^\bullet\cdot, \cdot^\bullet, M_0 \rangle$, be an open Petri net. Two markings $M_1, M_2 : P \to \mathbb{N}$ are* internally equivalent *($M_1 \equiv_M M_2$) if and only if*

$$\forall p \in P \backslash I . M_1(p) = M_2(p)$$

**Definition 12.** *Let $S$ be a* `ServiceTemplate`*, and let $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, with $\mathcal{P}_S = \langle P_S, T_S, {}^\bullet\cdot, \cdot^\bullet, M_0 \rangle$, be the Petri net encoding of $S$. Let also $P$ be a valid workflow* `Plan` *for $S$. $P$ is also* deterministic *if and only if for each pair $M_1, M_2$ of markings reached by executing two finite, complete*[5] *sequential traces of $P$*

$$M_1 \equiv_M M_2.$$

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 then be directly determined from the marking that is reached performing the corresponding firing sequence. We thus first characterize the states, requirements, and capabilities that are active in a marking (Definition 13), and we then employ such characterization to list the effects of a deterministic `Plan` (Remark 1).

**Definition 13.** *Let $S$ be a* `ServiceTemplate`*, and let $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, with $\mathcal{P}_S = \langle P_S, T_S, {}^\bullet\cdot, \cdot^\bullet, M_0 \rangle$, be the Petri net encoding of $S$. Let also $N_i = \langle S_{N_i}, R_{N_i}, C_{N_i}, O_{N_i}, \mathcal{M}_{N_i} \rangle$, with $\mathcal{M}_{N_i} = \langle \overline{s}, \rho, \gamma, \tau \rangle$, be a node in $S$. Finally, let $M$ be a marking.*

- *The* active states *in $M$ are*

$$A_S^M = \{ s \mid s \in P_S \setminus I_S \wedge M(s) = 1 \}.$$

- *The* assumed requirements *in $M$ are*

$$A_R^M = \{ r \mid M(r) = 0 \wedge r \in b(c, S) \wedge M(r_c) = 1 \}.$$

- *The* offered capabilities *in $M$ are*

$$A_C^M = \{ c \mid M(c) = 1 \vee M(r_c) = 1 \}.$$

*Remark 1.* Let $S$ be a `ServiceTemplate` and let $\mathcal{Z}_S$ be its Petri net encoding. Let also $P$ be a deterministic `Plan`, and let $M_0$ and $M$ be the initial marking and a marking equivalent to the markings reached by performing the (complete) sequential traces of $P$ in $M_0$.

[5] A sequential trace for a `Plan` $P$ is *complete* if and only if its first and last operation correspond to an initial and to a final activity of $P$.

- The requirements that are assumed after $P$ are $A_R^M$ (where the newly assumed ones are $A_R^M \backslash A_R^{M_0}$), while those that are no more assumed are $A_R^{M_0} \backslash A_R^M$.
- The capabilities that are offered after $P$ are $A_C^M$ (where the newly added ones are $A_C^M \backslash A_C^{M_0}$), while those that are no more offered are $A_C^{M_0} \backslash A_C^M$.

Please note that it is possible to consider as initial marking any other (reachable) marking so as to analyze maintenance plans (starting from non-initial states) besides deployment plans. Obviously, the very same properties and techniques also apply in this case.

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 [23] on the associated Petri net. To show it, we first define the notion of *goal*, that is a marking putting exactly one token in the places denoting the states and capabilities that have to be active.

**Definition 14.** *Let $S$ be a* `ServiceTemplate`*, and let $N_i = \langle S_{N_i}, R_{N_i}, C_{N_i}, O_{N_i}, \mathcal{M}_{N_i} \rangle$, with $\mathcal{M}_{N_i} = \langle \overline{s}, \rho, \gamma, \tau \rangle$, be a node in $S$. A* goal *for planning in $\mathcal{Z}_S$ is a pair $G = \langle S_G, C_G \rangle$ such that*

*(a) $S_G \subseteq \bigcup_i S_{N_i}$ is the set of states to be reached, and*
*(b) $C_G \subseteq \bigcup_i C_{N_i}$ is the set of capabilities to be offered.*

*A valid sequential plan $P$ for $S$ reaches the goal $G = \langle S_G, C_G \rangle$ if and only if*

*(a) $\forall s \in S_G . s \in S_{N_i} \Rightarrow s$ is the current state of $N_i$, and*
*(b) $\forall c \in C_G . c \in C_{N_i} \wedge s$ is the current state of $N_i \Rightarrow c \in \gamma(s)$.*

**Theorem 1.** *Let $S$ be a* `ServiceTemplate`*, and let $\mathcal{Z}_S$ be the Petri net encoding of $S$. Finding a valid sequential plan for $S$ that reaches a goal $G$ corresponds to solving a coverability problem in $\mathcal{Z}_S$.*

*Proof.* Let $G = \langle S_G, C_G \rangle$. We can easily build a marking $M_G : P_S \rightarrow \{0, 1\}$ as follows:

$$\forall p \in P_S . M_G(p) = \begin{cases} 1 & \text{if } p \in S_G \\ 1 & \text{if } p = r_c \wedge c \in C_G \\ 0 & \text{otherwise} \end{cases}$$

From the above, it follows that finding a sequential plan that reaches the goal $G$ corresponds to solving the coverability problem for the marking $M_G$. □

**Theorem 2.** *Let $S$ be a* `ServiceTemplate`*, and let $G$ be a goal. Finding a valid sequential plan for $S$ that reaches $G$ can be solved with polynomial space.*

*Proof.* The proof follows from the facts that the Petri net encoding $\mathcal{Z}_S$ of $S$ is safe, that finding a sequential plan in $\mathcal{Z}_S$ that reaches $G$ corresponds to solving a coverability problem, and that coverability in safe Petri nets is PSPACE-complete [12]. □

Another classical notion in the Petri net context that assumes a specific meaning is that of *reversibility* [23]: The Petri net encoding of a `ServiceTemplate` $S$ is *reversible* if and only if it is always possible to softly reset the application, i.e. if whatever (valid) sequence of operations we perform, we can always get back to the initial state of $S$ by performing another (valid) sequence of operations. This is a very convenient property, because it guarantees that it is always possible to generate a sequential plan for any reachable goal from any application state.

**Definition 15.** *Let $S$ be a* `ServiceTemplate`*, and let $\overline{s}_S$ be its initial configuration (i.e., the configuration in which all the management protocols of its nodes are in their initial state). We say that $S$ is* softly resettable *if and only if for each valid sequential plan for $S$*

$$o_1 o_2 ... o_m$$

*there exists a continuation*

$$o_{m+1} o_{m+2} ... o_{m+n}$$

*such that*

$$o_1 o_2 ... o_m o_{m+1} o_{m+2} ... o_{m+n}$$

*is a valid sequential plan for $S$ such that the firing of $o_1 o_2 ... o_m o_{m+1} o_{m+2} ... o_{m+n}$ from $\overline{s}_S$ leads to $\overline{s}_S$.*

**Theorem 3.** *Let $S$ be a* `ServiceTemplate`*, and let $\mathcal{Z}_S$ be the Petri net encoding of $S$.*

$$S \text{ is softly resettable} \Leftrightarrow \mathcal{Z}_S \text{ is reversible.}$$

*Proof.* By Definition 15, $S$ is softly resettable if and only if the following condition holds: ($C$) For each valid sequence $o_1 o_2 ... o_n$, we can always determine a longer valid sequence $o_1 o_2 ... o_m o_{m+1} ... o_{m+n}$ such that by firing it in the initial configuration $\overline{s}_S$ we end up in the same configuration $\overline{s}_S$.

Notice that $\overline{s}_S$ corresponds to the initial marking of the Petri net encoding $\mathcal{Z}_S$, and that a valid sequence of operations corresponds to a firing sequence in $\mathcal{Z}_S$. Thus, condition $C$ corresponds to saying that whatever firing sequence we can perform in the initial marking, we can always find a longer firing sequence that (starts and) ends up in the initial marking. This in turn corresponds to saying that $\mathcal{Z}_S$ is reversible (since whatever marking we can reach with a sequence of firings, we can always come back to the initial marking). □

## 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 [11]. This is witnessed by the proliferation of so-called *configuration management systems*, like Chef (https://www.chef.io/chef/) or Puppet (https://puppetlabs.com/). 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 (e.g., [5,16]). Our approach to protocol specification and analysis brings some similarities for instance with [3,14,22,26], that employ high-level Petri nets for protocol specification, and exploit notions like firability, reachability, and coverability, to analyse such protocols. For instance, [3] employs "numeric" Petri nets to model and analyse communication protocols. Such nets generalize tokens into tuples of variables to model fields in protocol messages, introduce net data variables to store "global values", and associate conditions and operations with transitions to permit checking and editing net variables. As the problem we address is simpler, we do not need a complex system like [3] since we just need to synchronize the management of connected components, by allowing each component to determine whether a needed capability is actually offered. Similar considerations apply to [14,22,26].

A detailed comparison with other existing approaches is beyond the scope of this paper[6]. We focus next on the subset of approaches more closely related to ours, tailored 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 [15]. 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 $\langle$`Unavailable`, {`ServerContainer`}, `Setup`, `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 (https://jujucharms.com/) and Engage [17], 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 [24].

[6] A more detailed discussion on existing approaches exploiting Petri nets for protocol engineering can be found in [13].

To this end, TOSCA offers a rich type system permitting to match, adapt and reuse existing solutions [10]. Since our proposal extends this type system, it can also be exploited to refine existing reuse techniques, like [9,27]. 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 `Node-Type`. 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 [6]. 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 [2,18], 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 automata 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 [4], 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. 4.4. Another interesting direction for future work is to investigate the applicability of more sophisticated fault diagnosis analyses (like [20,21]) 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 [9,27] by including the behaviour information coming from management protocols.

## Appendix

The objective of this appendix is to provide the properties of the Petri net encoding of a `ServiceTemplate` (see Definition 8) that are needed to prove its safeness (see Proposition 1). First, since each node $N_i$ in a `ServiceTemplate` $S$ can be in a unique state, exactly one of the internal places denoting its states contains one token, while the others contain no token. This holds at any given time, and thus in any marking that can be reached from the initial marking of the Petri net encoding of $\mathcal{Z}_S$. In short, (i) each internal place of the net encoding a `ServiceTemplate` contains at most one token. The same holds also for the open places modeling (ii) capabilities and (iii) requirements.

**Lemma 1.** *Let $S$ be a* `ServiceTemplate` *and let $\mathcal{Z}_S = \langle \mathcal{P}_S, I_S \rangle$, with $\mathcal{P}_S = \langle P_S, T_S, \bullet\cdot, \cdot\bullet, M_0 \rangle$, be the Petri net encoding of $S$. Let also $M$ be a marking reachable from the initial marking $M_0$ of $\mathcal{Z}_S$. For each node $N_i = \langle S_{N_i}, R_{N_i}, C_{N_i}, O_{N_i}, \mathcal{M}_{N_i} \rangle$ (with $\mathcal{M}_{N_i} = \langle \overline{s}, \rho, \gamma, \tau \rangle$) in $S$, the following properties hold:*

*(i)* $\exists s' \in S_{N_i}.M(s') = 1 \ \wedge \ \forall s \in S_{N_i}.s \neq s' \Rightarrow M(s) = 0$ *or, equivalently:*

$$\Sigma_{s \in S_{N_i}} M(s) = 1$$

*(ii) Let $s$ be the current state of a node $N_i$ (i.e. $s \in S_{N_i} \wedge M(s) = 1$). For any capability $c \in C_{N_i}$, the number of tokens in the open places $r_c$ and $c$ is:*

$$c \notin \gamma(s) \Leftrightarrow M(c) + M(r_c) = 0$$
$$c \in \gamma(s) \Leftrightarrow M(c) + M(r_c) = 1$$

*(iii) Let $s$ be the current state of a node $N_i$ (i.e. $s \in S_{N_i} \wedge M(s) = 1$). For any requirement $r \in R_{N_i}$ bound to a capability $c$ (i.e., $r \in b(c, S)$), the number of tokens in the open places $r$ and $r_c$ is:*

$$r \notin \rho(s) \Leftrightarrow (M(r) = M(r_c) = 0) \vee (M(r) = M(r_c) = 1)$$
$$r \in \rho(s) \Leftrightarrow M(r) = 0 \wedge M(r_c) = 1$$

*Proof.* The proofs for (i), (ii), and (iii) are listed below.

(i) For each node $N_i$, the places denoting its states are internal to $\mathcal{Z}_S$. Hence, their input and output transitions are not changed by the merge process, which in turn means that only the net transitions (encoding the protocol transitions) of the same node $N_i$ can add/remove tokens to/from them.
By construction, the above mentioned transitions always input exactly one token from an internal place and output exactly one token to an internal place (potentially the same). This guarantees that the total number of tokens in the internal places of a single node cannot change:

$$\Sigma_{s \in S_{N_i}} M(s) = \Sigma_{s \in S_{N_i}} M'(s),$$

where $M'$ is a marking reached by firing a transition in $M$.

The above, along with the fact that the initial marking $M_0$ of $\mathcal{Z}_S$ includes a token only in the places denoting the initial states of the nodes in $S$ (i.e., for each node $N_i$, $\Sigma_{s \in S_{N_i}} M_0(s) = 1$), implies that any sequence of firings starting from the initial marking will preserve exactly one token in the internal places denoting the states of each node.

(ii) First, we show that the property holds in the initial marking $M_0$ of $\mathcal{Z}_S$. According to the definition of management protocols (Definition 3), $\gamma(\overline{s}) = \varnothing$, which means that (in order for the property to hold) the initial marking $M_0$ of the open places must be empty (i.e., for each capability $c$, $M(c) + M(r_c) = 0$). This follows from the construction of $\mathcal{Z}_S$ (Definition 8), thus the property holds for $M_0$.

Since the property holds for the initial marking, we can prove that it holds for every reachable marking, by showing that no transition can invalidate the property. We will thus consider it as invariant.
Consider the capability $c$ of a node $N_i$. The places mentioned in the property (i.e., $c$ and $r_c$) are connected to the $c_\uparrow$ and $c_\downarrow$ transitions, and to the transitions of $N_i$ that input/output a token to/from $c$. These are the only transitions that might affect the invariant, since the transitions connected to the requirements managed by the controller of $c$ cannot change the marking of $c$ nor that of $r_c$.

The $c_\uparrow$ and $c_\downarrow$ transitions cannot affect the invariant, since they do not change the total number of tokens in $c$ and $r_c$. This is because, whenever $c_\uparrow$ fires, it removes one token from $c$, but it also adds one token to $r_c$ (and to all of the other $r_i$ places). Symmetrically, whenever $c_\downarrow$ fires, it removes one token from $r_c$ (and from each of the other $r_i$ places), but it also adds one token to $c$.

Thus, the only transitions that might invalidate the invariant are the transitions of the node $N_i$ that input/output one token to/from $c$. Since all these transitions move a token from a state $s$ to a state $s'$, they can be classified as follows:

(a) $c$ is either provided in both $s$ and $s'$ or in neither of them (i.e., $c \in \gamma(s) \cap \gamma(s') \vee c \notin \gamma(s) \cup \gamma(s')$);
(b) $c$ is provided in $s'$, but it is not provided in $s$ (i.e., $c \in \gamma(s') - \gamma(s)$);

(c) $c$ is provided in $s$, but it is not provided in $s'$ (i.e., $c \in \gamma(s) - \gamma(s')$).

Each of these cases is consistent with the property that we want to prove.

(a) In the first case, transitions do not affect $c$ at all, as (by construction) they are not even connected to $c$. They thus preserve the sum $M(c) + M(r_c)$, as well as the truth value of $c \in \gamma(\cdot)$.

(b) In the second case, transitions lead to a state $s'$ such that $c \in \gamma(s')$, but they also add a token to $c$. If the invariant held before the transition (i.e., $M(c) + M(r_c) = 0$ with $M(s) = 1 \wedge c \notin \gamma(s)$), it also holds after the transition, because the sum becomes $M(c) + M(r_c) = 1$ with $M(s') = 1 \wedge c \in \gamma(s)$.

(c) The third case is precisely the opposite of the second one, since transitions lead to a state $s'$ such that $c \notin \gamma(s')$ and they remove a token from $c$. If the invariant held before the transition (i.e., $M(c) + M(r_c) = 1$ with $M(s) = 1 \wedge c \in \gamma(s)$), then it also holds after the transition. The sum indeed becomes $M(c) + M(r_c) = 1$ with $M(s') = 1 \wedge c \notin \gamma(s)$.

In conclusion, since the invariant holds for $M_0$ and none of the transitions can invalidate it, by induction (over the length of a firing sequence) it holds for any reachable marking.

(iii) The proof of the property follows the same line as the one for (ii). Namely, the property can be proved to hold for any reachable marking by induction over the length of a firing sequence, by showing that it holds for the initial marking $M_0$, and that none of the transitions can invalidate such property.

□

## References


1. de Alfaro, L., Henzinger, T.A.: Interface automata. In: Proceedings of ESEC/FSE-9, pp. 109–120. ACM (2001)
2. Baldan, P., Corradini, A., Ehrig, H., Heckel, R.: Compositional semantics for open Petri nets based on deterministic processes. Math. Struct. Comput. Sci. **15**(01), 1–35 (2005)
3. Billington, J., Wheeler, G.R., Wilbur-Ham, M.C.: PROTEAN: a high-level petri net tool for the specification and verification of communication protocols. IEEE Trans. Softw. Eng. **14**(3), 301–316 (1988)
4. Billington, J., et al.: The petri net markup language: concepts, technology, and tools. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003)
5. Bochmann, G.V., Sunshine, C.A.: A survey of formal methods. In: Green Jr., P.E. (ed.) Computer Network Architectures and Protocols. Applications of Communications Theory, pp. 561–578. Springer, Heidelberg (1982)
6. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Dustdar, S., Leymann, F., Villari, M. (eds.) ESOCC 2015. LNCS, vol. 9306, pp. 19–33. Springer, Heidelberg (2015). http://dx.doi.org/10.1007/978-3-319-24072-5_2
7. Brogi, A., Canciani, A., Soldani, J.: Modelling the behaviour of management operations in TOSCA. Technical report, University of Pisa, July 2015

8. Brogi, A., Canciani, A., Soldani, J., Wang, P.: Modelling the behaviour of management operations in cloud-based applications. In: Moldt, D. (ed.) Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE 2015), CEUR Workshop Proceedings, vol. 1372, pp. 191–205. CEUR-WS.org (2015)
9. Brogi, A., Soldani, J.: Finding available services in TOSCA-compliant clouds. Science of Computer Programming 115–116, 177–198, Special Section on Foundations of Coordination Languages and Software (FOCLASA 2012), Special Section on Foundations of Coordination Languages and Software (FOCLASA 2013) (2016)
10. Brogi, A., Soldani, J., Wang, P.W.: TOSCA in a nutshell: promises and perspectives. In: Villari, M., Zimmermann, W., Lau, K.-K. (eds.) ESOCC 2014. LNCS, vol. 8745, pp. 171–186. Springer, Heidelberg (2014)
11. 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 Gener. Comput. Syst. **25**(6), 599–616 (2009)
12. Cheng, A., Esparza, J., Palsberg, J.: Complexity results for 1-safe nets. In: Shyamasundar, R.K. (ed.) FSTTCS 1993. LNCS, vol. 761, pp. 326–337. Springer, Heidelberg (1993)
13. Cheung, T.Y.: Petri nets for protocol engineering. Comput. Commun. **19**(14), 1250–1257 (1996)
14. Courtiat, J.P., Ayache, J.M., Algayres, B.: Petri nets are good for protocols. SIGCOMM Comput. Commun. Rev. **14**(2), 66–74 (1984)
15. Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a component model for the cloud. Inf. Comput. **239**, 100–121 (2014)
16. Diaz, M.: Modeling and analysis of communication and cooperation protocols using Petri net based models. Comput. Netw. **6**(6), 419–441 (1982)
17. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: Proceedings of PLDI 2012, pp. 263–274. ACM (2012)
18. Kindler, E.: A compositional partial order semantics for Petri net components. In: Azéma, P., Balbo, G. (eds.) ICATPN 1997. LNCS, vol. 1248, pp. 235–252. Springer, Heidelberg (1997)
19. Kopp, O., Binz, T., Breitenbücher, U., Leymann, F.: Winery - modeling tool for TOSCA-based cloud applications. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 700–704. Springer, Heidelberg (2013)
20. Lohmann, N.: Why does my service have no partners? In: Bruni, R., Wolf, K. (eds.) WS-FM 2008. LNCS, vol. 5387, pp. 191–206. Springer, Heidelberg (2009)
21. Lohmann, N., Fahland, D.: Where did i go wrong? In: Sadiq, S., Soffer, P., Völzer, H. (eds.) BPM 2014. LNCS, vol. 8659, pp. 283–300. Springer, Heidelberg (2014)
22. Morgan, E.T., Razouk, R.R.: Interactive state-space analysis of concurrent systems. IEEE Trans. Software Eng. **10**, 1080–1091 (1987)
23. Murata, T.: Petri nets: properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989)
24. OASIS: Topology and Orchestration Specification for Cloud Applications (2013). http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf
25. OASIS: TOSCA Simple Profile in YAML (2014). http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.0/TOSCA-Simple-Profile-YAML-v1.0.pdf
26. Paule, C., Eckert, H.: The NEt Simulation SYstem NESSY: Summary and Example. Ges. fur Mathematik u, Datenverarbeitung (1985)
27. Soldani, J., Binz, T., Breitenbcher, U., Leymann, F., Brogi, A.: ToscaMart: a method for adapting and reusing cloud applications. J. Syst. Softw. **113**, 395–406 (2016)

http://www.springer.com/978-3-662-53400-7

Transactions on Petri Nets and Other Models of Concurrency XI

Koutny, M.; Desel, J.; Kleijn, J. (Eds.)
2016, XVIII, 319 p. 103 illus., Softcover
ISBN: 978-3-662-53400-7