Simulation-based matching of cloud applications

Filippo Bonchi[b], Antonio Brogi[a], Andrea Canciani[a], Jacopo Soldani[a]

[a] *Dept. of Computer Science, University of Pisa, Italy*
[b] *Univ. Lyon, CNRS, ENS de Lyon, UCB Lyon 1, LIP*

Abstract

OASIS TOSCA aims at solving the problem of managing complex applications across heterogeneous clouds by providing a standard, vendor-agnostic language to describe them. TOSCA permits defining a cloud application as an orchestration of typed components, which can be instantiated by matching other TOSCA applications.

In this paper we first present two types of behaviour-aware matching of applications (*exact* and *plug-in*) both based on a notion of simulation. We then extend the notion of plug-in matching by relaxing the notion of simulation to permit matching an operation with a sequence of operations. We also present a coinductive procedure to compute such relaxed simulation, and we formally prove the termination, soundness, and completeness of such procedure.

Keywords: service matching, cloud application, TOSCA, reuse, simulation, coinduction

1. Introduction

Cloud computing has revolutionised IT by permitting to run on-demand distributed applications at a fraction of the cost which was necessary just a few years ago [2]. However, current cloud technologies suffer of a lack of standardization, with different providers offering similar resources in a different manner. How to flexibly manage complex applications across heterogeneous clouds is thus one of the main research problems in the cloud environment [10].

To tackle this issue, OASIS released the *Topology and Orchestration Specification for Cloud Applications* (TOSCA [24]), a standard to describe complex cloud applications, and to support the automation of their management. An application is specified in TOSCA by instantiating component types (possibly by reusing matching applications [13]), by connecting a component's requirements to the capabilities of other components, and by orchestrating the operations of the application components into plans defining the management of the whole application.

Unfortunately, the current specification of TOSCA [24] does not permit to describe the *behaviour* of applications' management operations. More precisely, there is no way to describe the order in which the operations of a component must be invoked, nor how

Email addresses: filippo.bonchi@ens-lyon.fr (Filippo Bonchi), brogi@di.unipi.it (Antonio Brogi), canciani@di.unipi.it (Andrea Canciani), soldani@di.unipi.it (Jacopo Soldani)

Published article: Bonchi F, Brogi A, Canciani A, Soldani J. "Simulation-based matching of cloud applications", Science of Computer Programming, vol. 162, pp. 110-131, 2018. DOI: 10.1016/j.scico.2017.06.001

those operations depend on the requirements or affect the capabilities of that component. As a consequence, existing matchmaking and adaptation techniques (e.g., [12, 28]) are behaviour-unaware, and this may cause problems when substituting a component by reusing a matched application.

In this paper we first recall how TOSCA can be extended to specify the behaviour of management operations by generalising the model we proposed in [7]. Namely, the management protocols of a TOSCA component can be described by means of finite state machines whose states and transitions are associated with conditions on the requirements and capabilities of such component. The objective of those conditions is essentially to define the consistency of the states of a component, and to constrain the executability of its management operations to the satisfaction of its requirements.

We then constrain the existing notions of (syntactic) *exact* and *plug-in* matching in TOSCA [12] to take into account behaviour information in management protocols. More precisely, we define when a desired management protocol can be "simulated" [27] by another management protocol, and we exploit such notion of simulation to constrain exact and plug-in matchings.

We then relax the notion of simulation into that of f-simulation, where f is a function associating each transition in the desired management protocol with a sequence of transitions in the available protocol. This permits to further relax the notion of plug-in matching, so to match operations of a desired component with sequences of operations of an available application. We also describe how flexibly plug-in matched applications can be suitably adapted so to be employed in place of desired components.

Finally, we introduce a coinductive [19] algorithm (called ComputeFs) that permits computing the function f determining an f-simulation between two management protocols. We also coinductively prove that ComputeFs is terminating, sound and complete.

This paper is an extended version of [5], including (i) a more detailed background on TOSCA, (ii) a formalisation of the coinductive algorithm ComputeFs, and (iii) the formal assessment of termination, soundness, and completeness of ComputeFs.

The rest of the paper is organised as follows. Sect. 2 provides the necessary background on TOSCA, while Sect. 3 describes how TOSCA can be extended to specify the behaviour of management operations. Sect. 4.1 recalls the notions of (syntactic) *exact* and *plug-in* matching. Sect. 4.2 introduces the notion of management protocol simulation, and it employs such notion to further constrain the *exact* and *plug-in* matchings. Sect. 4.3 relaxes the notion of simulation into that of f-simulation to further relax *plug-in* matching. Sect. 5 presents the algorithm to compute f-simulations and formally assess its termination, soundness, and completeness. Finally, Sects. 6 and 7 discuss related work and draw some conclusions, respectively.

2. Background: TOSCA

TOSCA [24] is an OASIS standard aimed at enabling the specification of portable cloud applications and the automation of their management. To do so, TOSCA provides a modelling language to describe the structure of a cloud application as a typed topology graph, and its tasks as plans. More precisely, each cloud application is represented as a *service template* (Fig. 1), consisting of a mandatory *topology template* and of optional management *plans*.

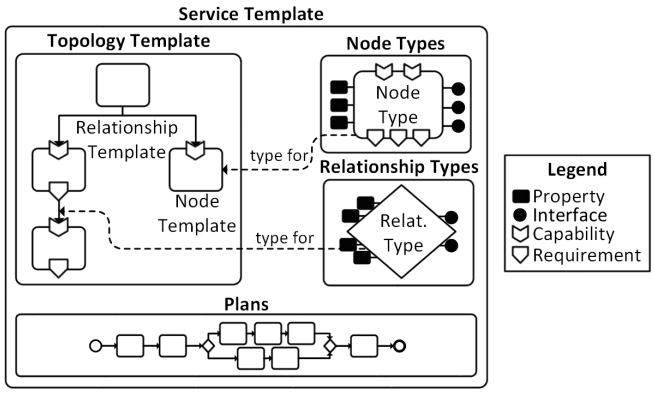

Figure 1: A TOSCA service template.

The topology template is a typed directed graph describing the structure of the composite cloud application. Its nodes (called *node templates*) model the application components, while its edges (called *relationship templates*) model the relations among those components. Node templates and relationship templates are typed by means of *node types* and *relationship types*, respectively. A node type defines (i) the observable properties of an application component, (ii) the possible states of its instances, (iii) its requirements, (iv) the capabilities it offers to satisfy other components' requirements, and (v) its management operations. Relationship types describe the properties of relationships occurring among components. Requirements, capabilities, properties and operations externally exposed by a service template can be described in its *boundary definitions*.

Plans instead permit describing the management aspects of a service template. Each plan is a workflow orchestrating the management operations offered by the application components to address (part of) the management of the whole cloud application.[1]

***Example* 2.1.** We hereby exemplify how a Moodle application can be specified as a TOSCA service template. MoodleApplication (Fig. 2) is composed by six components, each represented by a node template in its topology. Moodle is the front-end of the application and Database is its back-end. The other four components (viz., a Server, a MySQL runtime environment, an Ubuntu virtual machine, and a Debian virtual machine) form the software stacks needed to properly run Moodle and Database.

Requirements and capabilities of components are interconnected by relationship templates, to specify that a component is HostedOn another, or that a component ConnectsTo another. For instance, Moodle has two requirements that must be fulfilled:

- The requirement WebAppRuntime specifies that Moodle has to be hosted on a runtime for web applications. WebAppRuntime is fulfilled by connecting it (with a HostedOn relationship) to the capability WebAppRuntime of Server. This means that, whenever the service template MoodleApplication is deployed, Moodle will be installed and hosted on Server.

[1] A more detailed, self-contained introduction to TOSCA can be found in [13].

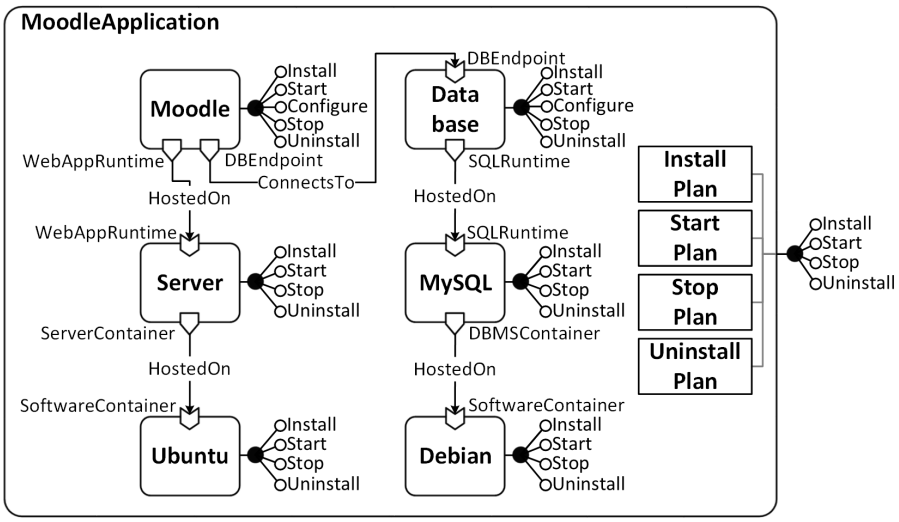

Figure 2: Example of TOSCA service template.

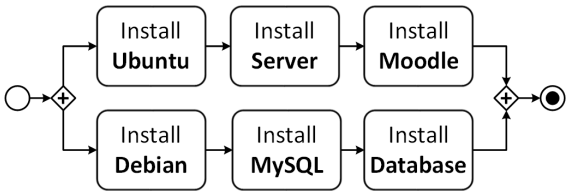

Figure 3: Example of TOSCA plan.

- The requirement DBEndpoint specifies that Moodle needs to know the actual endpoint of the back-end Database. DBEndpoint is hence fulfilled by connecting it (with a ConnectsTo relationship) to the capability DBEndpoint of Database. This means that, whenever the service template MoodleApplication is deployed, a connection between Moodle and Server will be set up.

All nodes forming the topology of MoodleApplication also expose a TOSCA lifecycle interface [25]. The lifecycle interface contains the operations to Install, Start, Stop and Uninstall a component. In the case of Moodle and of Database, the lifecycle interface also contains the operation to Configure them.

A lifecycle interface is also exposed on the boundary definitions of the service template MoodleApplication. Such interface contains the operations to Install, Start, Stop, and Uninstall the whole service template, and the actual implementation of such operations is given by plans orchestrating the management operations of the nodes forming the topology of MoodleApplication. For instance, the operation Install of MoodleApplication is implemented by InstallPlan, which orchestrates the operations Install of the inner nodes as illustrated in Fig. 3. □

3. Management protocols for cloud applications

While a TOSCA node type can be described by means of its states, requirements, capabilities, properties, 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 this section is precisely to show how we proposed (in [7]) to extend TOSCA to specify the behaviour of the management operations of a node type, as well as their relations with its states, requirements, and capabilities.

Essentially, we want to describe whether and how the management operations of a node type N depend on (i) other operations of the same node and/or on (ii) operations of other nodes providing the capabilities that satisfy the requirements of N:

(i) The first kind of dependencies can be easily described by specifying the relationship between states and management operations of N. More precisely, to describe the order with which the operations of N can be executed, we introduce a transition relation τ specifying whether an operation o can be executed in a state s, and which state is reached by executing o in s.

(ii) The second kind 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 node type 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, we permit associating each state s of a node type N with the capabilities provided by N in s, and to explicitly specify which capabilities are maintained during a transition.[2]

Definition 3.1 (Management protocol)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, where S_N, R_N, C_N, and O_N are the sets of its states, requirements, capabilities, and management operations, and $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ is the* management protocol *of N, where:*

- *$\overline{s}_N \in S_N$ is the initial state,*
- *$\rho_N : S_N \to 2^{R_N}$ is a function indicating, for each state $s \in S_N$, which conditions on requirements must hold,*
- *$\chi_N : S_N \to 2^{C_N}$ is a function indicating which capabilities of N are concretely offered in a state $s \in S_N$, and*

[2] The latter is a proper extension that generalises our initial definition of management protocols [7, 9], where we were assuming that all capabilities were maintained during a transition. The extension will be justified by Def. 4.7, which shows why transitions need to predicate on capabilities.

- $\tau_N \subseteq S_N \times 2^{R_N} \times 2^{C_N} \times O_N \times S_N$ *is a set of quintuples modelling the transition relation (i.e.,* $\langle s, H, G, o, s' \rangle \in \tau_N$ *denotes that in state* s*, and if condition* H *holds,* o *is executable and leads to state* s' *— by maintaining the capabilities in* G *during the transition).*

According to Def. 3.1, management protocols allow for operations that have non-deterministic effects (namely, a state may have two outgoing transitions corresponding to the same operation and leading to different states). This form of non-determinism is not acceptable when managing TOSCA applications [13]. We will thus focus on *deterministic* management protocols (i.e., protocols ensuring deterministic effects when performing an operation in a state).

Definition 3.2 (Deterministic management protocol)**.** *Let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ *be the management protocol of a node type* N*.* $\mathcal{M}_N$ *is* deterministic *iff*

$$\forall \langle s_1, H_1, G_1, o_1, s_1' \rangle, \langle s_2, H_2, G_2, o_2, s_2' \rangle \in \tau_N : \big(s_1 = s_2 \wedge o_1 = o_2\big) \Rightarrow s_1' = s_2'.$$

Furthermore, for each transition, the conditions on requirements and capabilities should be coherent with the starting and target states. Namely, the requirements assumed to hold in the starting state, as well as those assumed to hold in the target state, should also be assumed to hold during the transition, to avoid inconsistencies. Analogously, the capabilities that can be maintained during a transition are (at most) those offered by both its starting and target states.

Definition 3.3 (Well-formed management protocol)**.** *Let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ *be the management protocol of a node type* N*.* $\mathcal{M}_N$ *is* well-formed *iff*

$$\forall \langle s, H, G, o, s' \rangle \in \tau_N : \rho_N(s) \cup \rho_N(s') \subseteq H \ \wedge \ G \subseteq \chi_N(s) \cap \chi_N(s').$$

In the following, we consider well-formed, deterministic management protocols.

Finally, we recall (from [7]) the *intensional* operational semantics of the management protocol of a single component (i.e., a TOSCA node type), which models all possible sequences of management operations that can be performed on a component if the conditions on the needed requirements are satisfied by the environment. Formally, the intensional semantics of the management protocol of a node type N can be defined by a labelled transition system over configurations that are the states of N.

Definition 3.4 (Intensional semantics of a management protocol)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node type. The* intensional semantics *of the management protocol* $\mathcal{M}_N$ *of* N *is denoted by a labelled transition system whose set of configurations is* S_N *and whose transition relation is defined by the following inference rule:*

$$\frac{N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle \ \wedge \ \mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle \ \wedge \ \langle s, H, G, o, s' \rangle \in \tau_N}{s \xrightarrow{\langle H, G, o \rangle}_N s'}$$

Intuitively, a transition $s \xrightarrow{\langle H,G,o \rangle}_N s'$ denotes that operation o can be executed on N when N is in state s, and under the hypotheses given by condition H, making N evolve into state s'. During the transition, it is guaranteed that N continues to offer the capabilities in G.

4. Behaviour-aware matching of cloud applications

4.1. Syntactic matching

A TOSCA node type N can be instantiated by substituting it with a service template S if the capabilities, requirements, and operations exposed by S are exactly the same as those of N [24]. In [12], we formalised this with the notion of (syntactic) *exact* matching.[3]

Definition 4.1 (Syntactic exact matching)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, and let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. S* (syntactically) exactly *matches N ($S \equiv N$) iff*

$$R_S = R_N \ \wedge \ C_S = C_N \ \wedge \ O_S = O_N.$$

The notion of (syntactic) *plug-in* matching ($\leqq$) relaxes that of exact matching by permitting to substitute a node type N with a service template S that, intuitively speaking, "requires less" and "offers more" than N. More precisely, S plug-in matches N when all requirements of S are exposed by N, and when all capabilities and operations of N are offered by S.

Definition 4.2 (Syntactic plug-in matching)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, and let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. S* (syntactically) plug-in *matches N ($S \leqq N$) iff*

$$R_S \subseteq R_N \ \wedge \ C_S \supseteq C_N \ \wedge \ O_S \supseteq O_N.$$

In [12], we also showed how a plug-in matched service template can be adapted so to exactly match a desired node type.

***Example* 4.1.** Consider the Server node type and the Apache and Tomcat service templates in Fig. 4. For simplicity, we assume that requirements, capabilities, and operations having the same name satisfy the syntactical matching conditions given in [12].

It is easy to see that the Apache service template syntactically exactly matches Server (viz., Apache $\equiv$ Server), while the same does not hold for the Tomcat service template. Since Tomcat and Server expose the same requirements, and since Tomcat exposes "more" capabilities and operations than Server, we have that Tomcat plug-in matches Server (viz., Tomcat $\leqq$ Server).

The Tomcat service template can hence be adapted to exactly match the Server node type [12]. We define a new service template (Fig. 5) having Tomcat as its only node, and exposing (via its boundary definitions) the same requirements, capabilities, and management operations as the target Server node type. □

The *exact* and *plug-in* matching notions [12] are purely syntactical, and do not take into account the behavioural information of management protocols. In the next sections, we first constrain them by including conditions on such behaviour information. We then provide a more flexible notion of behaviour-aware matching, extending the behaviour-aware *plug-in* matching in order to identify larger sets of service templates that can be adapted so to exactly match a node type.

[3] We model node types and service templates with the same structure, since we abstract from a service template's topology by focusing on its boundaries. For the sake of simplicity, we also assume service templates to be valid [11], and we abstract from purely syntactical matching constraints (e.g., name equivalence, type compatibility) and from some TOSCA concepts (e.g., properties, interfaces) that are not necessary to understand this paper. A detailed definition of TOSCA matching can be found in [12].

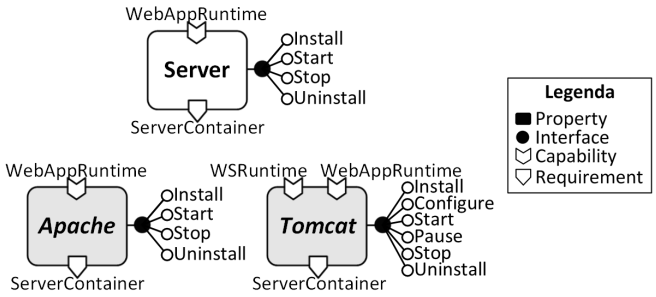

Figure 4: Example of node type and service templates.

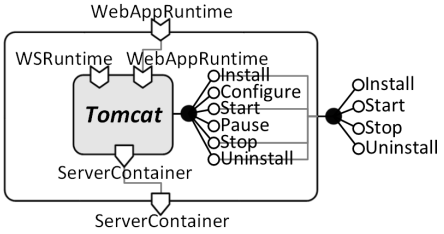

Figure 5: Example of adaptation of a plug-in matched service template.

4.2. Simulation-based matching

Consider a node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$, where S_N, R_N, C_N, and O_N are respectively the sets of its states, requirements, capabilities, and management operations, and where $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ is the *management protocol* of N. Consider also a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$, with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$. In order to constrain the notions of *exact* and *plug-in* matching, we formally define when the management protocol $\mathcal{M}_S$ of S can simulate [27] (the management behaviour defined by) the management protocol $\mathcal{M}_N$ of N.

Intuitively speaking, $\mathcal{M}_N$ is simulated by $\mathcal{M}_S$ if and only if the initial state of $\mathcal{M}_N$ is simulated by the initial state of $\mathcal{M}_S$. A state $s_N \in S_N$ is simulated by a state $s_S \in S_S$ if and only if (a) the requirements needed by s_N include all those needed by s_S, (b) the capabilities offered by s_N are included in those offered by s_S, and (c) for each transition leading from s_N to s'_N, there is a "compatible" transition starting from s_S (i.e., a transition performing the same operation o, not needing additional requirements, providing at least the same capabilities, and leading to a state s'_S that simulates s'_N).

Definition 4.3 (Simulation of management protocols)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$. Let also $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template, with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$.*

A state $s_N \in S_N$ is simulated *by $s_S \in S_S$ ($s_N \sqsubseteq s_S$) iff*

(a) $\rho_N(s_N) \supseteq \rho_S(s_S)$,

(b) $\chi_N(s_N) \subseteq \chi_S(s_S)$, and

(c) $s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N$
implies
$\exists s_S \xrightarrow{\langle H_S, G_S, o\rangle}_S s'_S : H_N \supseteq H_S \wedge G_N \subseteq G_S \wedge s'_N \sqsubseteq s'_S$.

A management protocol $\mathcal{M}_N$ *is* simulated *by a management protocol* $\mathcal{M}_S$ *(*$\mathcal{M}_N \sqsubseteq \mathcal{M}_S$*) iff* $\overline{s}_N \sqsubseteq \overline{s}_S$.

The notion of simulation permits us to constrain that of exact matching (Def. 4.1). Namely, to check whether a service template S *exactly* matches a node type N, we now check whether S syntactically exactly matches N, and whether the management protocol of S simulates that of N.

Definition 4.4 (Exact matching)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ *be a node type, and let* $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ *be a service template.* S exactly *matches* N *(*$S \equiv_b N$*) iff*

$$S \equiv N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S.$$

Analogously, to check whether a service template S *plug-in* matches a node type N, we check whether S syntactically plug-in matches N, and whether the management protocol of S simulates that of N.

Definition 4.5 (Plug-in matching)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ *be a node type, and let* $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ *be a service template.* S plug-in *matches* N *(*$S \leqq_b N$*) iff*

$$S \leqq N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S.$$

***Example* 4.2.** Example 4.1 showed that the Tomcat service template syntactically plug-in matches the Server node type. Consider now the management protocols in Fig. 6, where $\mathcal{M}_{\mathsf{Server}}$ is the management protocol for Server, while $\mathcal{M}_{\mathsf{Tomcat}}$ and $\mathcal{M}'_{\mathsf{Tomcat}}$ are two (alternative) management protocols for Tomcat.

It is easy to see that $\mathcal{M}_{\mathsf{Server}} \sqsubseteq \mathcal{M}_{\mathsf{Tomcat}}$, since Unavailable $\sqsubseteq$ NotInstalled. It follows that (with $\mathcal{M}_{\mathsf{Tomcat}}$) Tomcat plug-in matches Server (i.e., Tomcat $\leqq_b$ Server), and that Tomcat can still be adapted as shown in Fig. 5 to exactly match Server. The adaptation now also ensures that the operations of the adapted service template have the same behaviour as those of the desired node type.

The same does not hold if the management protocol of Tomcat is $\mathcal{M}'_{\mathsf{Tomcat}}$. For instance, by performing Install in their initial states, $\mathcal{M}'_{\mathsf{Tomcat}}$ and $\mathcal{M}_{\mathsf{Server}}$ respectively reach the states Installed and Stopped, and Installed $\not\sqsubseteq$ Stopped. This is because there is no transition starting from Installed that corresponds to the operation Start of Server.

Instead, if the operation Install of Server was corresponding to the sequencing of the operations Install and Configure of Tomcat, the aforementioned problem would have not been raised. Tomcat would have indeed reached its state Configured, where it can fire the operation Start. This means that a less strict definition of operation matching should allow Tomcat to match Server (with $\mathcal{M}'_{\mathsf{Tomcat}}$ as management protocol). □

4.3. Flexible simulation-based matching

We hereby extend the notion of plug-in matching to permit identifying larger sets of service templates that can be adapted to exactly match a desired node type N =

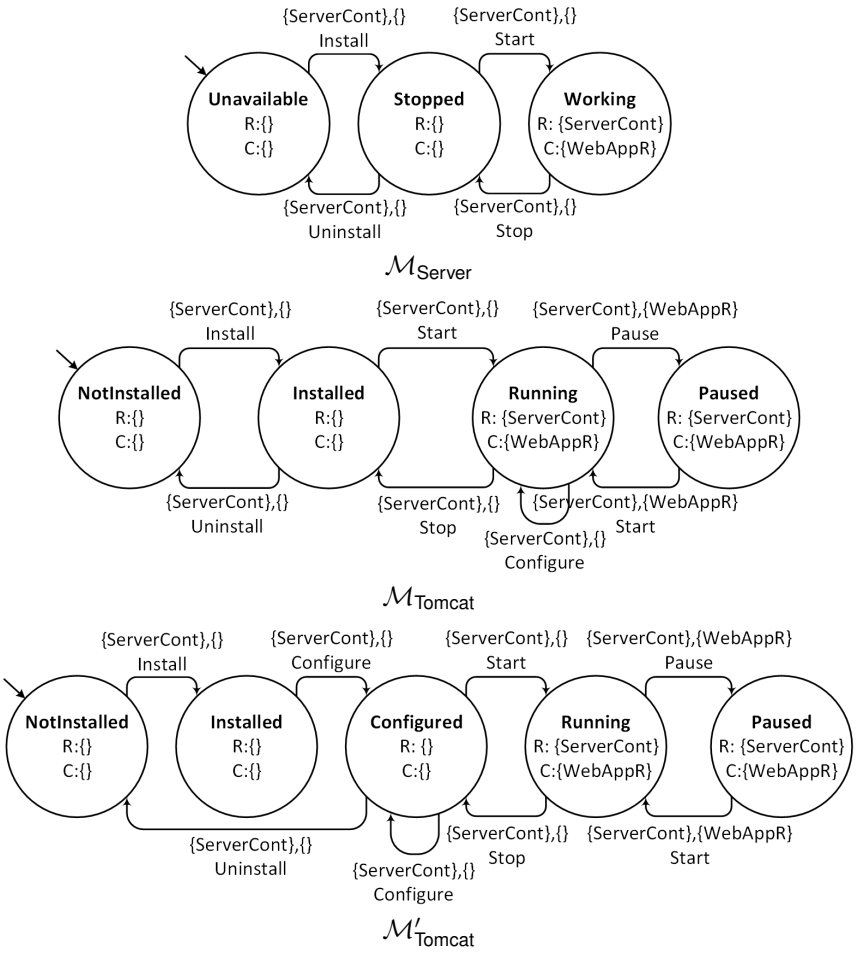

Figure 6: Example of management protocols.

$\langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. More precisely, we relax the definition of plug-in matching so that, given a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$, we permit matching (and substituting) operations in O_N with sequences of operations in O_S, based upon their effects on states, requirements and capabilities.

We first relax the notion of simulation between management protocols (Def. 4.3), by allowing to simulate each transition of a target management protocol $\mathcal{M}_N$ with sequences of transitions of an available management protocol $\mathcal{M}_S$. To do so, we first extend the intensional semantics of $\mathcal{M}_S$, by adding the transitions which permit to remain in the same state by performing an empty sequence ϵ of operations (without changing the conditions on requirements and capabilities), and which permit moving from a state to another by performing non-empty sequences of operations. While for singleton sequences the rule is trivial, for sequences of at least two operations we need the following rule: If w_1 permits transiting from state s to state s'' by assuming the requirements in H_1 and providing the capabilities in G_1, and if w_2 permits transiting from state s'' to state s' by assuming the requirements in H_2 and providing the capabilities in G_2, then $w_1 w_2$ permits transiting from s to s' by assuming $H_1 \cup H_2$ and by providing $G_1 \cap G_2$.

Definition 4.6 (n-step intensional semantics)**.** *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. The* n-step intensional semantics *of the management protocol $\mathcal{M}_S$ of S is modelled by a labelled transition system whose set of configurations is S_S and whose transition relation is defined by the following inference rules:*

$$\frac{-}{s \xRightarrow{\langle \rho_S(s), \chi_S(s), \epsilon \rangle}_S s} \qquad \frac{s \xrightarrow{\langle H,G,o \rangle}_S s'}{s \xRightarrow{\langle H,G,o \rangle}_S s'} \qquad \frac{s \xRightarrow{\langle H_1,G_1,w_1 \rangle}_S s'' \ \wedge \ s'' \xRightarrow{\langle H_2,G_2,w_2 \rangle}_S s'}{s \xRightarrow{\langle H_1 \cup H_2, G_1 \cap G_2, w_1 w_2 \rangle}_S s'}$$

Remark 4.1. The transition system $\Rightarrow_S$ (from Def. 4.6) effectively performs the reflexive and transitive closure of the transition system $\rightarrow_S$ (from Def. 3.4). One can readily check that whenever $\rightarrow_S$ is well-formed also $\Rightarrow_S$ is well-formed. This ensures that all the intermediates states that are reached during the execution of a transition $s \xRightarrow{\langle H,G,w \rangle}_S s'$ offer at least capabilities G and require at most requirements H. This meets the intuition behind the label G introduced in Sect. 3: G are all the capabilities that are maintained available *during* a transition. □

According to Def. 4.6, the transition system $\Rightarrow_S$ generates infinite branching whenever the corresponding protocol features some loops. In order to obtain a finitary description of it, we restrict to consider only *minimal* sequences of operations.

Definition 4.7 (Minimal n-step intensional semantics)**.** *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. The* minimal n-step intensional semantics *of the management protocol $\mathcal{M}_S$ of S is modelled by a labelled transition system whose set of configurations is S_S and whose transition relation is defined by the following inference rule:*

$$\frac{s \xRightarrow{\langle H,G,w \rangle}_S s' \ \wedge \ \nexists H_1 \subseteq H, G_1 \supseteq G, w_1 w_2 = w, w_2 \neq \epsilon : s \xRightarrow{\langle H_1,G_1,w_1 \rangle}_S s'}{s \bullet\!\!\Rightarrow^{\langle H,G,w \rangle}_S s'}$$

Remark 4.2. While the transition relation $\Rightarrow_S$ is in general infinite, $\bullet\!\!\Rightarrow_S$ is always finite. This is because the constraint $\nexists$ prevents loops. One can indeed readily check that any

sequence w of transitions that returns to the same state is replaced with a ϵ transition (since ϵ is a prefix of w, and since a ϵ transition has more relaxed requirements and preserves more capabilities than a sequence w of transitions[4]).

The transition system $\bullet\!\Rightarrow_S$ is also computable, as the hypotheses only involve operation sequences that are strictly shorter. Hence, the transition system $\bullet\!\Rightarrow_S$ can be built with an algorithm enumerating the operation sequences ordered by increasing length up to a maximum length equal to the number of states in the management protocol of S. Please recall that $\bullet\!\Rightarrow_S$ avoids loops, and this allows us to stop enumerating sequences longer than the number of states in the management protocol of S. □

We now relax the notion of simulation (Def. 4.3) into that of f-simulation, where $f : S_N \times S_S \times O_N \to O_S^*$ is a function associating each transition in the target management protocol $\mathcal{M}_N$ with a (possibly empty) sequence of transitions in the available management protocol $\mathcal{M}_S$.

Intuitively speaking, $\mathcal{M}_N$ can be f-simulated by $\mathcal{M}_S$ if and only if the initial state of $\mathcal{M}_N$ can be f-simulated by the initial state of $\mathcal{M}_S$. A state $s_N \in S_N$ is in turn f-simulated by a state $s_S \in S_S$ if and only if (a) the requirements needed by s_N contain all those needed by s_S, (b) the capabilities offered by s_N are contained in those offered by s_S, and (c) for each transition starting from s_N, there is a transition in $\bullet\!\Rightarrow_S$ starting from s_S, not needing additional requirements, providing at least the same capabilities, and leading to a state s'_S that in turn f-simulates s'_N.

Definition 4.8 (f-simulation of management protocols)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ be a node type, with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N\rangle$. Let also $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ be a service template, with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S\rangle$.*

A state $s_N \in S_N$ is f-simulated *by a state $s_S \in S_S$ ($s_N \sqsubseteq_f s_S$) iff the following conditions hold.*

(a) $\rho_N(s_N) \supseteq \rho_S(s_S)$,

(b) $\chi_N(s_N) \subseteq \chi_S(s_S)$, *and*

(c) $s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N$
implies
$\exists s_S \bullet\!\xrightarrow{\langle H_S, G_S, f(s_N, s_S, o)\rangle}_S s'_S : H_N \supseteq H_S \wedge G_N \subseteq G_S \wedge s'_N \sqsubseteq s'_S$.

A management protocol $\mathcal{M}_N$ is f-simulated *by a management protocol $\mathcal{M}_S$ ($\mathcal{M}_N \sqsubseteq_f \mathcal{M}_S$) iff $\overline{s}_N \sqsubseteq_f \overline{s}_S$.*

Remark 4.3. A similar notion could be defined by replacing the transition system $\bullet\!\Rightarrow_S$ with the transition system $\Rightarrow_S$ (Def. 4.6). This would put weaker constraints on f, as $\Rightarrow_S$ is much larger than $\bullet\!\Rightarrow_S$. Nonetheless any f satisfying the weaker simulation constraints would have an associated f' fulfilling the stricter ones. We have selected $\bullet\!\Rightarrow_S$ to have a decidable f-simulation, since the transitions system $\bullet\!\Rightarrow_S$ is finite and computable (see Remark 4.2). □

[4]The latter condition directly follows from the well-formedness of management protocols (Def. 3.3) and from the rules defining $\Rightarrow_S$ (Def. 4.6).

It is easy to see that the notion of f-simulation supports a more *flexible* form of matching than simulation. It indeed permits matching the operations of the target node type N with (different) sequences of operations of the available service template S.

Definition 4.9 (Flexible plug-in matching)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node type, and let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ be a service template. S* flexibly plug-in *matches N ($S \lesssim_b N$) iff*

$$R_S \subseteq R_N \ \wedge \ C_S \supseteq C_N \ \wedge \ \mathcal{M}_N \sqsubseteq_f \mathcal{M}_S.$$

Property 4.1 (Relation between $\leqq_b$ and $\lesssim_b$)**.** *If an available service template S* plug-in *matches a node type N, then S also* flexibly plug-in *matches N, i.e. $\leqq_b \subseteq \lesssim_b$.*

Proof. Consider a service template S and a node type N, and suppose that

$$S \leqq_b N.$$

By definition of $\leqq_b$ (Def. 4.5), we have that

$$S \leqq N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S.$$

According to definition of $\leqq_b$ (Def. 4.2), the above means that

$$R_S \subseteq R_N \ \wedge \ C_S \supseteq C_N \ \wedge \ O_S \supseteq O_N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S,$$

which obviously implies that

$$R_S \subseteq R_N \ \wedge \ C_S \supseteq C_N \ \wedge \ \mathcal{M}_N \sqsubseteq \mathcal{M}_S.$$

Let us now denote with Id the identity function mapping each operation o on itself. By Defs. 4.3 and 4.8, we have that $\sqsubseteq \equiv \sqsubseteq_{Id}$, which in turns means that

$$R_S \subseteq R_N \ \wedge \ C_S \supseteq C_N \ \wedge \ \mathcal{M}_N \sqsubseteq_{Id} \mathcal{M}_S.$$

According to Def. 4.9, the above means that

$$S \lesssim_b N.$$

□

It is worth noting that, while the adaptation technique for (syntactically) plug-in matched service templates [12] can be directly applied to requirements and capabilities, management operations need now to be adapted by taking into account sequences. We hence map the operations exposed on the boundaries of the adapted service template onto plans composing the operation of the flexibly plug-in matched service template. More precisely, since each operation to be matched can be associated with a different sequence according to f (and depending on the states of the target node type and matched service template), each operation exposed by the adapted service template is now associated with a conditional workflow encoding all the mappings given by f.

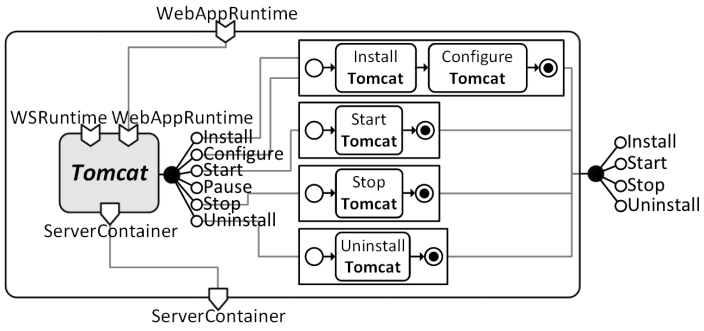

Figure 7: Example of adaptation of a flexibly plug-in matched service template.

***Example* 4.3.** Consider again the Server node type and the Tomcat service template in Fig. 4. One can now readily check that Tomcat flexibly plug-in matches Server (i.e., Server $\lesssim_b$ Tomcat), even if we consider the management protocols $\mathcal{M}_{\mathsf{Server}}$ and $\mathcal{M}'_{\mathsf{Tomcat}}$ in Fig. 6. This is because $\mathcal{M}_{\mathsf{Server}}$ is f-simulated by $\mathcal{M}'_{\mathsf{Tomcat}}$, where f is the identity function, except that for mapping the operation Install of Server with the sequencing of the operations Install and Configure of Tomcat.

It follows that we can adapt Tomcat as shown in Fig. 7. Namely, we create a new service template containing Tomcat as its only node, and exposing on its boundary definitions the features of the node type to be matched (i.e., Server). We then map requirements and capabilities as shown in [12]. Finally, we implement each operation of the adapted service template with workflow plans built according to the mappings given by f. Namely, the Install operation is implemented by a sequential plan which invokes the operations Install and Configure of Tomcat. Each other operation is implemented by plans containing a single invocation to the homonym operation of Tomcat (e.g., Start is implemented by a plan which only invokes the Start operation of Tomcat). □

It is worth noting that Example 4.3 shows a static translation: Each operation exposed on the boundaries of the adapted service template is implemented by a plan that is independent from the current states of the target node type and of the matched service template. According to the definition of $\sqsubseteq_f$ (Def. 4.8) this is not always the case, since f may depend on their current states. In the next section we present an algorithm which computes *all* possible f, from which it is trivial to extract a static translation (if it exists). If no such translation exists, the adapter would need some additional logic, namely it should include some conditional statements to track the state of the node type and/or of the service template.

5. Computing an f-simulation

We hereby present an algorithm to compute f-simulations (Sect. 5.1), and we formally assess its termination, soundness, and completeness (Sect. 5.2).

5.1. The algorithm ComputeFs

Consider a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ and a node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_S \rangle$, where $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$ and $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$ are the cor-

responding management protocols. According to Def. 4.9, to check whether S flexibly plug-in matches N we need to check whether $\mathcal{M}_S$ f-simulates $\mathcal{M}_N$ (i.e., $\mathcal{M}_S \sqsubseteq_f \mathcal{M}_N$).

Let us denote with $W_S \subseteq O_S^*$ set of all minimal operation sequences in $\mathcal{M}_S$ (see Def. 4.7). We now present an algorithm (called ComputeFs — Algorithm 1) capable of finding all functions $f : S_N \times S_S \times O_N \to W_S$ such that $\mathcal{M}_S \sqsubseteq_f \mathcal{M}_N$. Intuitively speaking, the algorithm starts by permitting to map each operation in O_N with any sequence of operations in W_S, and iteratively refines the mapping by removing the mappings leading to states that do not f-simulate (for any f). This process continues until the mapping cannot be refined any more.

More precisely, ComputeFs employs a $l \times m \times n$ matrix F, where l is the number of states in S_N (i.e., $l = |S_N|$), m is the number of states in S_S (i.e., $m = |S_S|$), and n is the number of operations in O_N (i.e., $n = |O_N|$). Each entry $F[i, j, o]$ stores the set of all words $w \in W_S$ such that, for some f, mapping (i, j, o) into w would potentially allow j to f-simulate i (i.e., $i \sqsubseteq_f j$). Intuitively, the matrix F represents an element in the lattice of all functions $S_N \times S_S \times O_N \to 2^{W_S}$, and the algorithm consists of a greatest fixpoint computation on such lattice (i.e., F starts from the top of the lattice, and each refinement makes it become a lower element — until it cannot be lowered any more).

Initially, there is no information about f-simulation, and ComputeFs can only check whether two states $i \in S_N$ and $j \in S_S$ are compatible in terms of requirements and capabilities (i.e., whether $\rho_N(i) \supseteq \rho_S(j) \wedge \chi_N(i) \subseteq \chi_S(j)$ — line 3). If this is the case, then j is a candidate to simulate i, and ComputeFs maps each operation $o \in O_N$ with any operation sequence in W_S (i.e., $F[i, j, o] \leftarrow W_S$ — line 4). Otherwise, there is no way to simulate i with j, and thus there is no way to map the operations in O_N onto sequences in W_S (lines 5-6).

Remark 5.1. ComputeFs initialises the matrix F as follows:

$$F[i, j, o] = \begin{cases} W_S & \text{if } \rho_N(i) \supseteq \rho_S(j) \wedge \chi_N(i) \subseteq \chi_S(j) \\ \varnothing & \text{otherwise} \end{cases}$$

(where $i \in S_N$, $j \in S_S$, and $o \in O_N$). □

F is then iteratively refined by removing all the mappings that lead to states that do not f-simulate (lines 7-14). At each step, after storing the previously computed F in $\widehat{F}$ (line 8), ComputeFs computes a new value W to be assigned to $F[i, j, o]$, for each $i \in S_N$, $j \in S_S$, and $o \in O_N$. W is initially set to $\varnothing$ (line 10). Then, for each w assigned to $F[i, j, o]$ at the previous refinement step (i.e., for each $w \in \widehat{F}[i, j, o]$), ComputeFs checks whether o and w lead to states that were candidates for f-simulation at the previous refinement step, i.e. if i can go in i' with o, then j can go in j' with w, and j' is a candidate to simulate i' (i.e., $\forall o' \in O_N.\widehat{F}[i', j', o'] \neq \varnothing$)[5]. If this is the case, w is added to W (line 12). The resulting W is then assigned to $F[i, j, o]$ (line 13).

Remark 5.2. One can readily check that, at each refinement step, $F[i, j, o]$ is obtained by restricting $\widehat{F}[i, j, o]$ (i.e., the value of $F[i, j, o]$ computed at the previous refinement step) as follows:

$$F[i, j, o] = \{w \in \widehat{F}[i, j, o] \mid \forall i \xrightarrow{\langle H_N, G_N, o\rangle}_N i'.\exists j \bullet\!\!\xrightarrow{\langle H_S, G_S, w\rangle}_S j' : H_N \supseteq H_S \wedge G_N \subseteq G_S \wedge \forall o' \in O_N.\widehat{F}[i', j', o'] \neq \varnothing\}$$

[5]Such check is actually implemented by the function LeadsToCandidate (lines 17-31).

Algorithm 1 ComputeFs. Computing all functions f determining a f-simulation between the management protocols of a node type N and of a service template S.

```
1:  function ComputeFs(S,N)
2:      for all i ∈ S_S, j ∈ S_N do
3:          if ρ_N(i) ⊇ ρ_S(j) ∧ χ_N(i) ⊆ χ_S(j) then
4:              for all o ∈ O_N do F[i,j,o] ← W_S
5:          else
6:              for all o ∈ O_N do  F[i,j,o] ← ∅
7:      repeat
8:          F̂ ← F
9:          for all i ∈ S_S, j ∈ S_N, o ∈ O_N do
10:             W ← ∅
11:             for all w ∈ F̂[i,j,o] do
12:                 if LeadsToCandidate(w, F̂, S, N) then W ← W ∪ {w}
13:             F[i,j,o] ← W
14:     until F ≠ F̂
15:     return F
16:
17: function LeadsToCandidate(w, F, S, N)
18:     for all i --⟨H_N,G_N,o⟩-->_N i' do
19:         exists ← false
20:         for all j •--⟨H_S,G_S,w⟩-->_S j' do
21:             if H_N ⊇ H_S ∧ G_N ⊆ G_S then
22:                 isCandidate ← true
23:                 for all o' ∈ O_N do
24:                     if F[i',j',o'] = ∅ then
25:                         isCandidate ← false
26:                         break
27:                 if isCandidate then
28:                     exists ← true
29:                     break
30:         if not(exists) then return false
31:     return true
```

(where $i \in S_N$, $j \in S_S$, and $o \in O_N$). □

The iterative refinement process stops when the matrix F cannot be refined any more (i.e., when $F = \widehat{F}$ — line 14). This is guaranteed to happen, because at every step each entry $F[i, j, o]$ either shrinks or stays the same. By definition, when $F = \widehat{F}$, we reached the maximum fixpoint F. The latter is the output returned by ComputeFs (line 15). Given such F:

- If there is at least one operation in O_N that cannot be mapped in the starting states (i.e., $\exists o \in O_N.F[\overline{s}_N, \overline{s}_S, o] = \varnothing$), then there is no function f such that the starting states f-simulate (i.e., $\overline{s}_S \not\sqsubseteq_f \overline{s}_N$). This in turn implies that $\mathcal{M}_S$ does not

f-simulate $\mathcal{M}_N$ (i.e., $\mathcal{M}_S \not\sqsubseteq_f \mathcal{M}_N$).

- Otherwise, we can extract one of the functions f (such that $\mathcal{M}_S \sqsubseteq_f \mathcal{M}_N$) by simply selecting one of the possible mappings for each operation $o \in O_N$ and for each pair of states $(i, j) \in S_N \times S_S$.

***Example* 5.1.** Consider again the Tomcat service template and the Server node type, whose management protocols $\mathcal{M}'_{\mathsf{Tomcat}}$ and $\mathcal{M}_{\mathsf{Server}}$ are in Fig. 6. We now show that ComputeFs permits determining a function f such that $\mathcal{M}'_{\mathsf{Tomcat}} \sqsubseteq_f \mathcal{M}_{\mathsf{Server}}$.[6]

ComputeFs initially builds the matrix in Fig. 8 by checking whether the states of $\mathcal{M}'_{\mathsf{Tomcat}}$ and $\mathcal{M}_{\mathsf{Server}}$ are compatible in terms of requirements and capabilities. If a state of $\mathcal{M}'_{\mathsf{Tomcat}}$ is compatible with a state of $\mathcal{M}_{\mathsf{Server}}$ (such as in the case of NotInstalled and Unavailable), the corresponding cell is filled by mapping each operation of Server to the set of all minimal operation sequences in Tomcat (denoted by $\mathsf{W}_{\mathsf{Tomcat}}$). Otherwise, the cell is left empty (such as in the case of NotInstalled and Working).

The matrix in Fig. 8 is then iteratively refined by removing all operation mappings that lead to states that do not f-simulate. The matrix obtained after the first refinement step is displayed in Fig. 9. Consider, for instance, the initially available mappings for the operation Install of Server when Tomcat is NotInstalled and Server is Unavailable (Fig. 8). All such mappings (but those to empty sequence ϵ, and to the sequences Install and Install ·Configure) lead to states that are not candidate to f-simulate (viz., to states whose corresponding cells are empty in the matrix in Fig. 8). The only mappings maintained in the matrix in Fig. 9 are hence ϵ, Install and Install·Configure.

By applying a second refinement step to the matrix in Fig. 9, all mappings remain unchanged, meaning that we already reached the greatest fixpoint. As all operations do have mappings in the starting states NotInstalled and Unavailable, we can conclude that there exist a set of functions $f : S_{\mathsf{Server}} \times S_{\mathsf{Tomcat}} \times O_{\mathsf{Server}} \to O^*_{\mathsf{Tomcat}}$ such that $\mathcal{M}'_{\mathsf{Tomcat}} \sqsubseteq_f \mathcal{M}_{\mathsf{Server}}$. We can hence pick one of such functions by selecting one mapping for each operation in each pair of states. For instance, we can pick a function f defined as follows:

$$f(\_,\_,o) = \begin{cases} \mathsf{Install} \cdot \mathsf{Configure} & o = \mathsf{Install} \\ \mathsf{Start} & o = \mathsf{Start} \\ \mathsf{Stop} & o = \mathsf{Stop} \\ \mathsf{Uninstall} & o = \mathsf{Uninstall} \end{cases}$$

□

5.2. *Properties of* ComputeFs

We hereby assess termination, soundness and completeness of the algorithm ComputeFs (Algorithm 1).

5.2.1. *Termination of* ComputeFs

The termination of ComputeFs follows trivially from its rules for initialising and refining the matrix F (which are recapped in Remark 5.1 and 5.2, respectively).

[6] An example showing that ComputeFs can also be exploited to check that two management protocols do not f-simulate is in Appendix A

Tomcat \ Server	Unavailable	Stopped	Working
NotInstalled	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	
Installed	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	
Configured	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	
Running			Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}
Paused			Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}

Figure 8: Initial matrix computed by ComputeFs when determining a function f such that $\mathcal{M}'_{\mathsf{Tomcat}} \sqsubseteq_f \mathcal{M}_{\mathsf{Server}}$. White cells correspond to states that are candidate to f-simulate, grey cells correspond to states that do not f-simulate.

Tomcat \ Server	Unavailable	Stopped	Working
NotInstalled	Install → {ε,Install,Install·Config} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → {Install·Config·Start, Install·Config·Start·Pause} Stop → W_{Tomcat} Uninstall → {ε,Install,Install·Config}	
Installed	Install → {ε,Config,Config·Uninst} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → {Config·Start, Config·Start·Pause} Stop → W_{Tomcat} Uninstall → {ε,Config, Config·Unistall}	
Configured	Install → {ε,Uninst,Uninst·Install} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → {Start,Start·Pause} Stop → W_{Tomcat} Uninstall → {ε,Uninstall, Uninstall·Pause}	
Running			Install → W_{Tomcat} Start → W_{Tomcat} Stop → {Stop} Uninstall → W_{Tomcat}
Paused			Install → W_{Tomcat} Start → W_{Tomcat} Stop → {Start·Stop} Uninstall → W_{Tomcat}

Figure 9: Refinement of the matrix in Fig. 8 obtained by removing all mappings that lead to states that are not candidate to f-simulate. Refined mappings are thicker, unchanged mappings are lighter.

Proposition 5.1 (Termination of ComputeFs)**.** ComputeFs *always terminates.*

Proof. The algorithm consists in an iterative refinement process, which stops whenever the matrix F cannot be refined any more, i.e. $F = \widehat{F}$. This is guaranteed to happen, because of the following facts:

- All entries of the matrix F are initialised with finite sets (viz., either they contain the set W_S of all minimal operation sequences, or they contain the empty set — Remark 5.1).
- At every step, each entry $F[i, j, o]$ either shrinks or stays the same (Remark 5.2), and $F[i, j, o]$ is lower-bounded by $\varnothing$.

□

5.2.2. Soundness and completeness of ComputeFs

To illustrate soundness and completeness of ComputeFs, we first need to set the stage with some preliminary notions. Namely, we need to introduce the operator Ψ_f (Def. 5.1), and to illustrate that the relation of f-simulation is a post-fixpoint of such operator (Lemma 5.1). We also need to introduce the endomap Φ (Def. 5.2), and to show that ComputeFs actually computes the greatest fixpoint of such endomap.

Consider a service template $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ and a node type $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$, whose management protocols are $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S\rangle$ and $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N\rangle$, respectively. The notion of f-simulation is defined (in Def. 4.8) as a relation such that, for all pairs $(s_N, s_S) \in S_N \times S_S$, satisfies the following constraint:

$$
\begin{aligned}
s_N \sqsubseteq_f s_S := & \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
& \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
& (\forall s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N, \exists s'_S . s_S \overset{\langle H_S, G_S, f(s_N, s_S, o)\rangle}{\Longrightarrow}_S s'_S \wedge \\
& \qquad H_N \supseteq H_S \wedge \\
& \qquad G_N \subseteq G_S \wedge \\
& \qquad s'_N \sqsubseteq s'_S).
\end{aligned}
$$

The same relation can be defined as a post-fixpoint of the operator Ψ_f, which is defined as follows.

Definition 5.1 (Operator Ψ_f)**.** *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$ be a service template (with $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S\rangle$), and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ be a node type (with*

$\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$*). The* operator Ψ_f *is defined as follows:*

$$\begin{aligned}\Psi_f(R) := \{(s_N, s_S) \in & R \mid \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\ & \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\ & (\forall s_N \xrightarrow{\langle H_N, G_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle H_S, G_S, f(s_N, s_S, o) \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\ & \qquad H_N \supseteq H_S \wedge \\ & \qquad G_N \subseteq G_S \wedge \\ & \qquad (s'_N, s'_S) \in R)\}\end{aligned}$$

Lemma 5.1. *The relation of f-simulation (viz.,* $\sqsubseteq_f$*) is a post-fixpoint of the operator* Ψ_f*. Namely:*

$$\sqsubseteq_f = \Psi_f(\sqsubseteq_f).$$

Proof. The thesis directly follows from the definitions of $\sqsubseteq_f$ and of Ψ_f (Def. 4.8 and Def. 5.1, respectively). □

In Sect. 5.1, we mentioned that the algorithm essentially consists of a greatest fixpoint computation on the lattices of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$. We hereby formalise this intuition by introducing a monotone endomap Φ on the aforementioned lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$, and by showing that ComputeFs actually computes the greatest fixpoint of Φ.

Definition 5.2 (Endomap Φ)**.** *Let* $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S \rangle$ *be a service template (with* $\mathcal{M}_S = \langle \overline{s}_S, \rho_S, \chi_S, \tau_S \rangle$*), and let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node type (with* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N \rangle$*). Let also* $W_S \subseteq O_S^*$ *be the set of all minimal operation sequences in* $\mathcal{M}_S$ *(Def. 4.7). We define the* endomap Φ *on the lattice of functions* $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ *as follows:*

$$\begin{aligned}\Phi(F)(s_N, s_S, o) := \{w \in & F(s_N, s_S, o) \mid \\ & \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\ & \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\ & (\forall s_N \xrightarrow{\langle H_N, G_N, o \rangle}_N s'_N, \exists s'_S . s_S \overset{\langle H_S, G_S, w \rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\ & \qquad H_N \supseteq H_S \wedge \\ & \qquad G_N \subseteq G_S \wedge \\ & \qquad \forall o', F(s'_N, s'_S, o') \neq \varnothing)\}\end{aligned}$$

Lemma 5.2. *The algorithm presented in Sect. 5.1 consists in computing of the greatest fixpoint of the endomap* Φ*.*

Proof. Consider the rules for initialising and updating the matrix F, which are formalised in Remark 5.1 and Remark 5.2. By definition of Φ:

- The initial matrix F is just $\Phi(\top)$, where $\top$ is the greatest element of the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ (i.e., a function assigning to any triple i, j, o the whole set W_s), and
- Each refinement step computes a matrix F such that $F = \Phi(\widehat{F})$.

The above, along with the fact that ComputeFs stops whenever $F = \widehat{F}$ (and that ComputeFs eventually terminates — Proposition 5.1), proves the thesis. □

We now have all the notions that are needed to coinductively [19] prove that ComputeFs (Algorithm 1) is sound and complete.

Proposition 5.2 (Soundness and Completeness of ComputeFs)**.** ComputeFs *is sound and complete.*

Proof. By Lemma 5.1 and Lemma 5.2, we respectively have that:

- The relation $\sqsubseteq_f$ is a post-fixpoint of the operator Ψ_f.
- ComputeFs consists in computing of the greatest fixpoint of the endomap Φ.

Given the above, to prove soundness and completeness of ComputeFs we can focus on Φ and Ψ_f, by showing how they are related. More precisely, we can exploit the operator Ψ_f to prove the following facts:

(a) If there is an f for which the states of a node type N are f-simulated by those of a service template S, then there is a non-empty fixpoint for the endomap Φ built on the corresponding lattice, and

(b) if Φ has a non-empty fixpoint, then it is always possible to extract from it a function f such that the states of N are f-simulated by those of S.

The above listed conditions (a) and (b) are proved by the following Lemmas 5.3 and 5.4, respectively. □

Lemma 5.3. *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$. Let also $W_S \subseteq O_S^*$ be the set of all minimal operation sequences in $\mathcal{M}_S$ (Def. 4.7), and Φ be an endomap on the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ built as shown in Def. 5.2.*

Consider a state $s_S \in S_S$ and a state $s_N \in S_N$. If there exists a function f for which $s_N \sqsubseteq_f s_S$, then there is a non-empty fixpoint for the endomap Φ on F. In formulas, $\forall s_N \in S_N, s_S \in S_S, o \in O_N$

$$\Phi(F_f)(s_N, s_S, o) = F_f(s_N, s_S, o)$$

where $F_f(s_N, s_S, o) = \{f(s_N, s_S, o) \mid s_N \sqsubseteq_f s_S\}$.

Proof. The definition of $F_f(s_N, s_S, o)$ naturally partitions the proof in two cases, i.e. (a) $F_f(s_N, s_S, o) = \varnothing$, and (b) $F_f(s_N, s_S, o) = \{f(s_N, s_S, o)\} \neq \varnothing$.

(a) Assume that $F_f(s_N, s_S, o) = \varnothing$. By definition of Φ

$$\Phi(F_f)(s_N, s_S, o) \subseteq F_f(s_N, s_S, o).$$

Since $F_f(s_N, s_S, o) = \varnothing$

$$\Phi(F_f)(s_N, s_S, o) \subseteq \varnothing,$$

which trivially implies that

$$\Phi(F_f)(s_N, s_S, o) = \varnothing.$$

From the above, and since $F_f(s_N, s_S, o) = \varnothing$, we have that $\forall o \in O_N$

$$\Phi(F_f)(s_N, s_S, o) = F_f(s_N, s_S, o).$$

(b) On the other hand, if $F_f(s_N, s_S, o) = \{f(s_N, s_S, o)\} \neq \varnothing$, then the definition of F_f ensures that

$$s_N \sqsubseteq_f s_S.$$

By expanding the definition of $\sqsubseteq_f$ we have that

$$\begin{aligned}
&\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
&\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
&(\forall s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N, \exists s'_S . s_S \overset{\langle H_S, G_S, f(s_N, s_S, o)\rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
&\qquad\qquad H_N \supseteq H_S \wedge \\
&\qquad\qquad G_N \subseteq G_S \wedge \\
&\qquad\qquad s'_N \sqsubseteq_f s'_S.
\end{aligned}$$

Since $s'_N \sqsubseteq_f s'_S \Rightarrow \forall o', F_f(p', q', o') = \{f(p', q', o')\} \neq \varnothing$, the above can be rewritten as follows:

$$\begin{aligned}
&\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\
&\chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\
&(\forall s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N, \exists s'_S . s_S \overset{\langle H_S, G_S, f(s_N, s_S, o)\rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\
&\qquad\qquad H_N \supseteq H_S \wedge \\
&\qquad\qquad G_N \subseteq G_S \wedge \\
&\qquad\qquad \forall o' \in O_N, F_f(s'_N, s'_S, o') \neq \varnothing.
\end{aligned}$$

The above predicate is proved to be true, and this means that we can write the

following equivalence (for every $o \in O_N$):

$$\begin{aligned}\{f(s_N, s_S, o)\} = \{f(s_N, s_S, o) \mid\ & \\ & \rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\ & \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\ & (\forall s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N, \exists s'_S . s_S \overset{\langle H_S, G_S, f(s_N,s_S,o)\rangle}{\bullet\!\!\dashrightarrow}_S s'_S \wedge \\ & \qquad H_N \supseteq H_S \wedge \\ & \qquad G_N \subseteq G_S \wedge \\ & \qquad \forall o', F_f(p', q', o') \neq \varnothing\},\end{aligned}$$

which by definition of Φ (Def. 5.2) means that (for every $o \in O_N$)

$$\{f(s_N, s_S, o)\} = \Phi(F_f)(s_N, s_S, o)$$

Finally, since $F_f = \{f(s_N, s_S, o)\}$, we obtain that (for every $o \in O_N$)

$$F_f = \Phi(F_f)(s_N, s_S, o)$$

□

Lemma 5.4. *Let $S = \langle S_S, R_S, C_S, O_S, \mathcal{M}_S\rangle$, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$. Let also Ψ_f be the operator defined in Def. 5.1, $W_S \subseteq O_S^*$ be the set of all minimal operation sequences in $\mathcal{M}_S$ (Def. 4.7), and Φ be an endomap on the lattice of functions $F\colon S_N \times S_S \times O_N \to 2^{W_S}$ built as shown in Def. 5.2.*

If Φ has a non-empty fixpoint, then it is possible to extract from it a function f for which $s_N \sqsubseteq_f s_S$ (with $s_N \in S_N$ and $s_S \in S_S$). Formally, assuming that

(i) $\forall s_N \in S_N, s_S \in S_S, o \in O_N . \Phi(F)(s_N, s_S, o) = F(s_N, s_S, o)$, *and*

(ii) $f_F : S_N \times S_S \times O_N \to O_S^*$
such that
$\forall s_N \in S_N, s_S \in S_S, o \in O_N . F(s_N, s_S, o) \neq \varnothing \Rightarrow f(s_N, s_S, o) \in F(s_N, s_S, o)$,

we have that

$$\sqsubseteq_F \subseteq \Psi_{f_F}(\sqsubseteq_F)$$

where $s_N \sqsubseteq_F s_S := \forall o \in O_N, F(s_N, s_S, o) \neq \varnothing$.

Proof. Consider a state $s_N \in S_N$ and a state $s_S \in S_S$, and suppose that

$$s_N \sqsubseteq_F s_S.$$

By definition of $\sqsubseteq_F$, we have that

$$\forall o \in O_N . F(s_N, s_S, o) \neq \varnothing.$$

The above, along with the hypoteses on f, implies that

$$\forall o \in O_N . f(s_N, s_S, o) \in F(s_N, s_S, o) \subseteq \Phi(F)(s_N, s_S, o).$$

By definition of Φ, the above can be rewritten as follows

$$\begin{aligned}\forall o \in O_N.\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\ \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\ (\forall s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N, \exists s'_S \in S_S.s_S \overset{\langle H_S, G_S, f(s_N,s_S,o)\rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\ H_N \supseteq H_S \wedge \\ G_N \subseteq G_S \wedge \\ \forall o', F(s'_N, s'_S, o') \neq \varnothing),\end{aligned}$$

which, by definition of $\sqsubseteq_F$, can be in turn rewritten as follows

$$\begin{aligned}\forall o \in O_N.\rho_N(s_N) \supseteq \rho_S(s_S) \wedge \\ \chi_N(s_N) \subseteq \chi_S(s_S) \wedge \\ (\forall s_N \xrightarrow{\langle H_N, G_N, o\rangle}_N s'_N, \exists s'_S \in S_S.s_S \overset{\langle H_S, G_S, f(s_N,s_S,o)\rangle}{\bullet\!\!\Longrightarrow}_S s'_S \wedge \\ H_N \supseteq H_S \wedge \\ G_N \subseteq G_S \wedge \\ s'_N \sqsubseteq_F s'_S).\end{aligned}$$

From the above, and because of the definition of Ψ_{f_F}, we have that

$$(s_N, s_S) \in \Psi_{f_F}(\sqsubseteq_F),$$

from which it follows the thesis we wanted to prove. □

6. Related work

While the matching between service templates and node types is indicated in the TOSCA primer [25] as a way to instantiate TOSCA node types, no definition of matching is given in either TOSCA [24] or in its primer [25].

A first (formal) definition of matching for TOSCA has been given in [12], where we proposed four definitions of matching (*exact*, *plug-in*, *flexible*, and *white-box*) between TOSCA service templates and node types, each identifying larger sets of service templates that can be adapted to exactly match a node type. While exact and plug-in matching are purely syntactical, flexible and white-box matching exploit ontologies to check whether two management operations are equivalent. In [12] we also showed how a non-exactly matched service template can be adapted so to exactly match a desired node type. The matchmaking and adaptation approach presented in this paper differs from that in [12]: The semantics of management operations is now given by management protocols [7] (rather than by ontologies), and with the notion of f-simulation we can substitute a desired operation with sequences of available operations, based upon the effects on states, requirements and capabilities.

The problem of how to match existing software components has been extensively studied in recent years. Many approaches are ontology-aware, like for instance the matchmaker for OWL-S services described by Klusch et al. [20]. Other approaches are behaviour-aware, like the behavioural congruence for OWL-S services defined by Bonchi et al. [6], or the heuristic black-box matching described by Eshuis and Grefen [16]. The main difference between the aforementioned approaches and ours is the type of information considered when matching single nodes. The matching levels considered by Klusch et al. [20], by Bonchi et al. [6], and by Eshuis and Grefen [16] are all defined in terms of input and output data, while we consider also technology requirements and capabilities.

Cavallaro et al. [15] propose an approach to automatically replace services based upon their functional interface and behaviour models. Cavallaro et al. [15] focus on many-to-many mappings among service operations, since such mappings have to hold in whatever state of the service. Our approach is more flexible in the sense that it is capable of mapping a single operation to different operation sequences depending on the state in which such operation is invoked. Furthermore, Cavallaro et al. [15] generate adapter scripts that have to be passed to a proxy any time the corresponding operations are invoked. Our approach instead adapts operations once for all (by translating the function f into a set of TOSCA plans).

Reussner et al. [26] describe how to adapt components based on parametric contracts, which permit modifying their interfaces depending on context properties in a way potentially more expressive than ours. Like us, Reussner et al. [26] exploit finite state machines to model interaction protocols. However, the approach by Reussner et al. [26] differs from ours since it does not make the relation between context properties and protocols, thus loosing information on what it is concretely reachable in a composite environment explicit.

Closely related approaches are also those by Motahari Nezhad et al. [23], by Inverardi and Tivoli [18], and by Bennaceur and Issarny [3], even if they propose approaches to synthesize mediators among service and service clients, while we focus on matching. Our work shares with the approach by Motahari Nezhad et al. [23] (and previous papers by the same authors) the idea of exploiting, at the same time, functional interfaces and behaviour models to match a service's operation with multiple operations of another service. Our function f (Def. 4.8) is indeed pretty similar to the *interface mapping* proposed by Motahari Nezhad et al. [23]. However, our approach is fully automated while that by Motahari Nezhad et al. [23] is semi-automated.

The approach by Inverardi and Tivoli [18] shares its baselines with that by Motahari Nezhad [23], and synthesizes adapters in the form of interaction protocols. Bennaceur and Issarny [3] instead exploit ontologies and constraint programming to infer one-to-one, one-to-many, and many-to-many correspondences between components' interfaces, and synthesize adapters in the form of labelled transition systems. Our approach synthesises adapters that are considerably simpler than those produced with the approaches by Inverardi and Tivoli [18] and Bennaceur and Issarny [3], as they just consist in the determined f functions (Def. 4.8).

It is also worth noting that our notion of f-simulation extends the notion of one-to-many operation mapping proposed by Bennaceur and Issarny [3]. Furthermore, despite the algorithm proposed by Bennaceur and Issarny [3] deals also with many-to-many mappings, its correctness is validated only by means of various use cases. We instead formally prove soundness and completeness of our algorithm (see Sect. 5.2).

Summing up, to the best of our knowledge, ours is the first fully automated approach for reusing TOSCA application components, which takes into account both functional and extra-functional features of TOSCA application components, and which relies on the widely accepted idea of exploiting behaviour models to match operations, and on behaviour simulation [27] to go beyond non-relevant operation mismatches.

A similar problem has been faced in the area of algebraic development techniques. For instance, Martins et al. [21, 22] show that the standard notion of signature morphism (which can be thought morally as our notion of syntactic plug-in matching) is often not flexible enough to deal with the problems of software reuse and refinement. In analogy with our work, the Martin et al. propose a notion of refinement based on the more flexible *logical interpretations*.

Action refinement has been also extensively studied in the context of concurrency theory, see e.g., Aceto [1], Van Glabbeek and Goltz [29], Gorrieri and Rensink [17]. In this setting action refinement is far more sophisticated than the one provided by our translation function f: a single action can usually be translated into a set of parallel processes, while f simply refines an action by a sequence of more fine-grained actions. At the level of behavioural equivalences and pre-orders this usually forces to leave the interleaving semantics and consider truly concurrent semantics (see e.g, Van Glabbeek and Goltz [29]). Another crucial simplification in our setting, which is however motivated by the TOSCA specification, is the absence of non-determinism. This simplification, which is sometimes referred as sequential action refinement, has been presented as the starting point for more elaborated theories (such as those by Aceto [1] and by Gorrieri and Rensink [17]), while in our work it covers a key role justified by applications.

It is worth to also discuss how f-simulation relates to *stuttering* [14] and *branching* [30] bisimulations. Stuttering bisimulation is defined on Kripke structures, where labels appear on states but not on transitions, and therefore there is no need for a function for translating transition labels. Branching bisimulation is instead defined on labelled transition systems, and permits matching a transition only with another transition having the same label, followed by a sequence of silent steps. It thus not requires any translation function which behaves as f, i.e. which maps a (desired) labelled transition to sequences of (available) labelled transitions. Furthermore, in both stuttering and branching bisimulations, the intermediate states occurring in a sequence of transitions should satisfy some equivalence constraints, while in f-simulation these states are completely hidden behind the definition of $\Rightarrow$ (Def. 4.6). This abstraction is safe in our context, since the interactions among components relies just on requirements and capabilities, which are annotated in the transitions and checked by the definition of f-simulation (see Remark 4.1). Moreover, since we consider a preorder rather than an equivalence, the additional transitions of intermediate states do not play any role.

7. Conclusions

In this paper, we first recalled how to formally model the behaviour of management operations in TOSCA applications by means of management protocols [7]. It is worth highlighting that such model builds upon, but is not limited to, TOSCA. It can indeed be adapted to other languages for specifying cloud applications, and more in general to

any stateful behaviour model of systems that describe states, requirements, capabilities, and operations.

We also introduced a notion of management protocol simulation, and we exploited it to constrain the (syntactical) *exact* and *plug-in* matching of [12] to take also into account the management behaviour of TOSCA applications. We then relaxed the notion of simulation into that of f-simulation, where f is a function associating each transition of a desired management protocol with a sequence of transitions of an available protocol. On that basis, we further constrained the notion of plug-in matching, by permitting to match the operations of a desired component with sequences of operations of an available application. We also described how to compute the function f determining the f-simulation, and how (flexibly) plug-in matching applications can be suitably adapted to be employed in place of desired components.

We see different possible extensions of this work. First, we intend to extend the proof-of-concept implementation of the syntactic matching that we proposed in [12] to support the behaviour-aware matching illustrated in this paper, and to integrate it in the open-source OpenTOSCA ecosystem [4]. We also plan to exploit such implementation to experimentally validate our approach.

We also plan to refine and extend our approaches for matching, adapting, and reusing TOSCA applications. On the one hand, we plan to constrain the topology fragment matchmaking approach that we proposed in [28] by including the behaviour information of management protocols. On the other hand, we intend to investigate other weaker notions of simulation to further relax our behaviour-aware matching notions.

Finally, it is worth noting that management protocols have been recently extended to permit modelling how nodes behave in presence of faults [8]. Essentially, fault-aware management introduce a new transition relation that model the explicit fault handling of a node. We plan to extend our notions of simulation (as well as the algorithm to compute f-simulations) to cope also with fault-handling transitions, and hence to permit matching cloud applications by taking into account both their normal and faulty behaviour.

Acknowledgments

Work partly supported by the project *Through the fog* (PRA_2016_64) funded by the Univ. of Pisa, and by *LABEX MILYON* (ANR-10-LABX-0070) of Univ. of Lyon, within the program *Investissements d'Avenir* (ANR-11-IDEX-0007) operated by the French National Research Agency (ANR).

References

[1] Luca Aceto. *Action refinement in process algebras*, volume 3. Cambridge University Press, 1992.

[2] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, 2010.

[3] Amel Bennaceur and Valérie Issarny. Automated synthesis of mediators to support component interoperability. *Software Engineering, IEEE Transactions on*, 41(3):221–240, 2015.

[4] Tobias Binz, Uwe Breitenbücher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA – a runtime for TOSCA-based cloud applications. In *Service-Oriented Computing*, volume 8274 of *LNCS*, pages 692–695. Springer, 2013.

[5] Filippo Bonchi, Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Behaviour-aware matching of cloud applications. In *10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, July 17-19, 2016*, pages 117–124. IEEE Computer Society, 2016.

[6] Filippo Bonchi, Antonio Brogi, Sara Corfini, and Fabio Gadducci. A net-based approach to web services publication and replaceability. *Fundamenta Informaticae*, 94(3-4):305–330, 2009.

[7] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Modelling and analysing cloud application management. In Schahram Dustdar, Frank Leymann, and Massimo Villari, editors, *Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings*, volume 9306 of *Lecture Notes in Computer Science*, pages 19–33. Springer International Publishing, 2015.

[8] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. Fault-aware application management protocols. In Marco Aiello, Broch Einar Johnsen, Schahram Dustdar, and Ilche Georgievski, editors, *Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings*, volume 9846 of *Lecture Notes in Computer Science*, pages 219–234. Springer International Publishing, 2016.

[9] Antonio Brogi, Andrea Canciani, Jacopo Soldani, and Pengwei Wang. A Petri net-based approach to model and analyze the management of cloud applications. In Maciej Koutny, J́org Desel, and Jetty Kleijn, editors, *Transactions on Petri Nets and Other Models of Concurrency XI*, volume 9930 of *Lecture Notes in Computer Science*, pages 28–48. Springer Berlin Heidelberg, 2016.

[10] Antonio Brogi, José Carrasco, Javier Cubo, Francesco D'Andria, Ahmad Ibrahim, Ernesto Pimentel, and Jacopo Soldani. EU project SeaClouds - adaptive management of service-based applications across multiple clouds. In Markus Helfert, Frédéric Desprez, Donald Ferguson, Frank Leymann, and Víctor Méndez Muñoz, editors, *CLOSER 2014 - Proceedings of the 4th International Conference on Cloud Computing and Services Science, Barcelona, Spain, April 3-5, 2014*, pages 758–763. SciTePress, 2014.

[11] Antonio Brogi, Antonio Di Tommaso, and Jacopo Soldani. Validating TOSCA application topologies. In Luís Ferreira Pires, Slimane Hammoudi, and Bran Selic, editors, *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.*, pages 667–678. SciTePress, 2017.

[12] Antonio Brogi and Jacopo Soldani. Finding available services in TOSCA-compliant clouds. *Science of Computer Programming*, 115–116:177–198, 2016.

[13] Antonio Brogi, Jacopo Soldani, and PengWei Wang. TOSCA in a nutshell: Promises and perspectives. In Massimo Villari, Wolf Zimmermann, and Kung-Kiu Lau, editors, *Service-Oriented and Cloud Computing: Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings*, volume 8745 of *Lecture Notes in Computer Science*, pages 171–186, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[14] Michael C. Browne, Edmund M. Clarke, and Orna Grümberg. Characterizing finite kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59(1–2):115 – 131, 1988.

[15] Luca Cavallaro, Elisabetta Di Nitto, and Matteo Pradella. An automatic approach to enable replacement of conversational services. In *Proceedings of the 7th International Joint Conference on Service-Oriented Computing*, ICSOC-ServiceWave '09, pages 159–174. Springer-Verlag, 2009.

[16] Rik Eshuis and Paul Grefen. Structural matching of bpel processes. In *Proceedings of ECOWS '07*, pages 171–180. IEEE, 2007.

[17] Roberto Gorrieri and Arend Rensink. Action refinement. In *Handbook of process algebra*, pages 1047–1147. 2001.

[18] Paola Inverardi and Massimo Tivoli. Automatic synthesis of modular connectors via composition of protocol mediation patterns. In *Proceedings of the ICSE'13*, pages 3–12. IEEE, 2013.

[19] Bart Jacobs and Jan Rutten. A tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin*, 62:62–222, 1997.

[20] Matthias Klusch, Benedikt Fries, and Katia Sycara. Owls-mx: A hybrid semantic web service matchmaker for owl-s services. *Web Semantics: Science, Services and Agents on the WWW*, 7(2):121–133, 2009.

[21] Manuel A Martins, Alexandre Madeira, and Luís Soares Barbosa. Refinement via interpretation. In *Software Engineering and Formal Methods, 2009 Seventh IEEE International Conference on*, pages 250–259. IEEE, 2009.

[22] Manuel A. Martins, Alexandre Madeira, and Luís Soares Barbosa. The role of logical interpretations in program development. *Logical Methods in Computer Science*, 10(1), 2014.

[23] Hamid Reza Motahari Nezhad, Guang Yuan Xu, and Boualem Benatallah. Protocol-aware matching of web service interfaces for adapter development. In *Proceedings of the 19th International Conference on World Wide Web*, WWW, pages 731–740. ACM, 2010.

[24] OASIS. Topology and Orchestration Specification for Cloud Applications. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf`, 2013.

[25] OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Primer. `http://docs.oasis-open.org/tosca/tosca-primer/v1.0/tosca-primer-v1.0.pdf`, 2013.

[26] Ralf H. Reussner, Steffen Becker, and Viktoria Firus. Component Composition with Parametric Contracts. In *Net.ObjectDays*, pages 155–169, 2004.

[27] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge University Press, 2011.

[28] Jacopo Soldani, Tobias Binz, Uwe Breitenbücher, Frank Leymann, and Antonio Brogi. ToscaMart: A method for adapting and reusing cloud applications. *Journal of Systems and Software*, 113:395–406, 2016.

[29] Rob Van Glabbeek and Ursula Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4):229–327, 2001.

[30] Rob J Van Glabbeek and W Peter Weijland. Branching time and abstraction in bisimulation semantics. *Journal of the ACM (JACM)*, 43(3):555–600, 1996.

Appendix A. Example of non-simulating management protocols

The objective of this appendix is to complement Example 5.1 by showing what happens when applying the algorithm ComputeFs to two management protocols that cannot f-simulate (for any f).

***Example* A.1.** Consider again the node type Server, whose management protocol $\mathcal{M}_{\mathsf{Server}}$ is in Fig. 6. Consider also the service template Tomcat, and suppose that its management protocol $\mathcal{M}''_{\mathsf{Tomcat}}$ is that in Fig. A.10 (which differs from the management protocol $\mathcal{M}'_{\mathsf{Tomcat}}$ in Fig. 6 only because of the absence of the transition leading from Running to Configured by executing the operation Stop). We now show that ComputeFs permits checking that there is no function f such that $\mathcal{M}''_{\mathsf{Tomcat}} \sqsubseteq_f \mathcal{M}_{\mathsf{Server}}$.

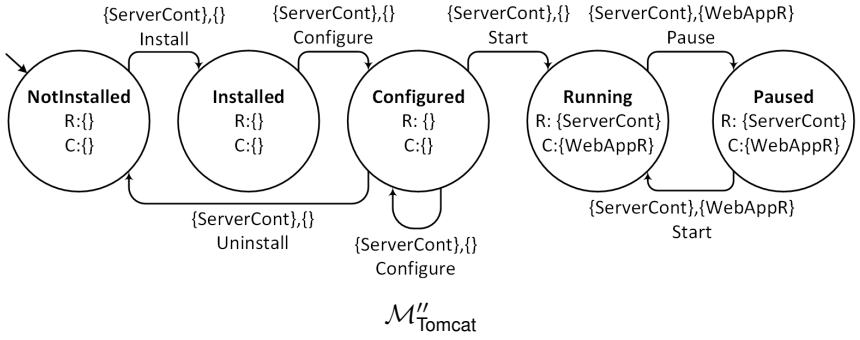

Figure A.10: Another example of management protocol.

ComputeFs initially builds the matrix in Fig. A.11 by checking whether the states of $\mathcal{M}''_{\mathsf{Tomcat}}$ and $\mathcal{M}_{\mathsf{Server}}$ are compatible in terms of requirements and capabilities. If a state of $\mathcal{M}''_{\mathsf{Tomcat}}$ is compatible with a state of $\mathcal{M}_{\mathsf{Server}}$ (such as in the case of the initial states NotInstalled and Unavailable), the corresponding cell is filled by mapping each operation of Server to any operation sequence in Tomcat (denoted by $\mathsf{W}_{\mathsf{Tomcat}}$). Otherwise, the cell is left empty (such as in the case of NotInstalled and Working).

ComputeFs then iteratively refines the matrix in Fig. A.11 by removing all operation mappings that lead to states that do not f-simulate. The matrix obtained after the first refinement step is displayed in Fig. A.12. It is worth noting that two pairs of states (namely, Running and Working, and Paused and Working) are no more candidate to f-simulate. In both cases, this is because there is no sequence of operations of Tomcat leading to a state that is candidate to f-simulate the state reached by executing Stop when Server is in state Working.

Similar considerations apply to the second and third refinement steps, whose resulting matrices are displayed in Figs. A.13 and A.14, respectively. The matrix in Fig. A.14 cannot be refined any more (as no cell is containing operation mappings to be refined). The initial state NotInstalled of Tomcat is *not* candidate to f-simulate the initial state Unavailable of Server, which means that there is no function f such that $\mathcal{M}''_{\mathsf{Tomcat}} \sqsubseteq_f \mathcal{M}_{\mathsf{Server}}$ (viz., $\mathcal{M}''_{\mathsf{Tomcat}}$ does not f-simulate $\mathcal{M}_{\mathsf{Server}}$). □

Tomcat \ Server	**Unavailable**	**Stopped**	**Working**
NotInstalled	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	
Installed	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	
Configured	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	
Running			Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}
Paused			Install → W_{Tomcat} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}

Figure A.11: Initial matrix computed by ComputeFs when determining a function f such that $\mathcal{M}''_{\mathsf{Tomcat}} \sqsubseteq_f \mathcal{M}_{\mathsf{Server}}$. White cells correspond to states that are candidate to f-simulate, grey cells correspond to states that do not f-simulate.

Tomcat \ Server	**Unavailable**	**Stopped**	**Working**
NotInstalled	Install → {ε,Install,Install·Config} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → {Install·Config·Start, Install·Config·Start·Pause} Stop → W_{Tomcat} Uninstall → {ε,Install,Install·Config}	
Installed	Install → {ε,Config,Config·Uninst} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → {Config·Start, Config·Start·Pause} Stop → W_{Tomcat} Uninstall → {ε,Config, Config·Unistall}	
Configured	Install → {ε,Uninst,Uninst·Install} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}	Install → W_{Tomcat} Start → {Start,Start·Pause} Stop → W_{Tomcat} Uninstall → {ε,Uninstall, Uninstall·Pause}	
Running			
Paused			

Figure A.12: Refinement of the matrix in Fig. A.11 obtained by removing all mappings that lead to states that are not candidate to f-simulate. Refined operation mappings are thicker, unchanged operation mappings are lighter.

Tomcat \ Server	**Unavailable**	**Stopped**	**Working**
NotInstalled	Install → {ε,Install,Install·Config} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}		
Installed	Install → {ε,Config,Config·Uninst} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}		
Configured	Install → {ε,Uninst,Uninst·Install} Start → W_{Tomcat} Stop → W_{Tomcat} Uninstall → W_{Tomcat}		
Running			
Paused			

Figure A.13: Refinement of the matrix in Fig. A.12 obtained by removing all mappings that lead to states that are not candidate to f-simulate.

Tomcat \ Server	**Unavailable**	**Stopped**	**Working**
NotInstalled			
Installed			
Configured			
Running			
Paused			

Figure A.14: Refinement of the matrix in Fig. A.13 obtained by removing all mappings that lead to states that are not candidate to f-simulate.