# Fault-aware management protocols for multi-component applications

Antonio Brogi[a], Andrea Canciani[a], Jacopo Soldani[a,*]

[a] *Department of Computer Science, University of Pisa*

**Abstract**

Nowadays, applications are composed by multiple heterogeneous components, whose management must be suitably coordinated by taking into account inter-component dependencies and potential failures. In this paper, we first present *fault-aware management protocols*, which allow to model the management behaviour of application components, and we then illustrate how such protocols can be composed to analyse and automate the overall management of a multi-component application. We also show how to recover applications that got stuck because a fault was not handled properly, or because a component is behaving differently than expected. To illustrate the feasibility of our approach, we present Barrel, a proof-of-concept application that permits editing and analysing fault-aware management protocols in multi-component applications. We also discuss the usefulness of Barrel by showing how it was fruitfully exploited it in a concrete case study and in a controlled experiment.

## 1. Introduction

How to automatically manage composite applications is currently one of the major concerns of enterprise IT [1]. This holds especially when deploying such applications on cloud platforms, as the efficient exploitation of cloud computing peculiarities strictly depends on the degree of automation of the management of applications [2].

Composite applications typically integrate various heterogeneous components. Hence, the deployment, configuration, enactment, and termination of the components forming a composite application must be suitably coordinated, by also taking into account all dependencies occurring among application components. As the number of components grows, or the need to reconfigure them becomes more frequent, application management becomes more and more time-consuming and error-prone [1].

Topology graphs provide a convenient representation of the structure of composite applications [3]. The nodes in a topology graph represent the components of an application, while its oriented arcs represent the dependencies among such components. More precisely, each node models an application component by describing its requirements, the operations to manage it, and the capabilities it features (to satisfy the requirements of other nodes). Arcs model inter-component dependencies by associating the requirements of a node with capabilities featured by other nodes.

The management behaviour of topology nodes can be specified by means of *management protocols*, as we illustrated in [4]. Each node can be equipped with its own management protocol, which is a finite state machine whose states and transitions are enriched with conditions on the requirements and capabilities of such node. Conditions on states permit defining which requirements of a node must be satisfied in a state, as well as which capabilities the node actually provides in such state. Conditions on transitions instead define which additional requirements must be satisfied to actually execute a management operation in a state. Given that topology graphs connect the requirements of a node with the capabilities that can satisfy them, and assuming that a requirement is satisfied only when the corresponding capability is actually provided, the management behaviour of a composite application can be derived by composing the management protocols of its nodes – according to the dependencies defined in its topology.

However, management protocols (as per [4]) do not cope with the potential occurrence of faults. This limits their applicability, as an application component may be affected by faults caused by another component on which it relies (e.g., a component may be shutdown or uninstalled while other components are relying on its capabilities). Faults and fault handling must indeed be considered when managing complex composite applications [5].

In this paper, we introduce *fault-aware management protocols*, which extend those in [4] by also allowing to model how nodes react to faults. We then show how this permits analysing and automating the management of composite applications in a fault-resilient manner.

We illustrate how to derive the fault-aware management behaviour of a composite application by combining the fault-aware management protocols of the nodes in its topology. We then show how such behaviour can be exploited to check the validity of a plan orchestrating the management of an application, and to determine its effects (e.g., which application configuration is reached by

[*] Corresponding author (`soldani@di.unipi.it`).

Published article: Brogi A, Canciani A, Soldani J. "Fault-aware management protocols for multi-component applications", Journal of Systems and Software, Volume 139, 2018, Pages 189-210.
DOI: 10.1016/j.jss.2018.02.005

executing it, or whether it may generate faults while being executed). We also show how this permits automatically determining plans that can accomplish specific management goals (e.g., reaching a desired application configuration, or restoring it after a fault occurred). Also, based on the observation that the actual behaviour of an application component may be different from that specified in its fault-aware management protocol (e.g., due to non-deterministic bugs [6]), we show how to deal with misbehaving components. Namely, we show how to model the unexpected behaviour of an application component by automatically completing its fault-aware management protocol, and we illustrate how this permits analysing the (worst possible) effects of misbehaving components on the rest of an application. We also present a solution to automatically recover applications that are stuck because of a misbehaving component and/or because a fault was not properly handled.

Finally, we show how we integrated fault-aware management protocols in the OASIS standard TOSCA (*Topology and Orchestration Specification for Cloud Applications* [7]). We first introduce Barrel, a proof-of-concept, web-based application that permits editing fault-aware management protocols of the components forming TOSCA applications, and analysing the management of such applications. We then discuss the usefulness of Barrel by showing how it can be fruitfully exploited to validate and automate the (fault-aware) management of a concrete case study, and by means of a controlled experiment.

This paper is an extended version of [8]. The fault-aware management protocols defined in this paper are a strict generalisation of those in [8], and such generalisation eases the specification of fault handling transitions. Also, this paper extends [8] by characterising the properties that must be satisfied by management protocols, by formalising the notion of hard recovery in composite applications, by providing a proof-of-concept implementation to edit and analyse fault-aware management protocols (viz., Barrel), and by discussing the usefulness of our approach by means of a case study and of a controlled experiment.

The rest of the paper is organised as follows. Sect. 2 provides an example motivating the need for fault-aware management protocols, which are formally introduced in Sect. 3. Sect. 4 shows how fault-aware management protocols can be automatically composed to analyse the management of a composite application in presence of faults. Sect. 5 introduces Barrel, while Sect. 6 discusses its usefulness through a case study and a controlled experiment. Finally, Sects. 7 and 8 discuss related work and draw some concluding remarks, respectively.

## 2. Motivating scenario

Consider the toy application in Fig. 1, composed by a web-based *Frontend* that exploits a *Backend* API to serve its functionalities. Both the *Frontend* and the *Backend* are hosted on an *Nginx* server, which is in turn deployed on a *Ubuntu* operating system.

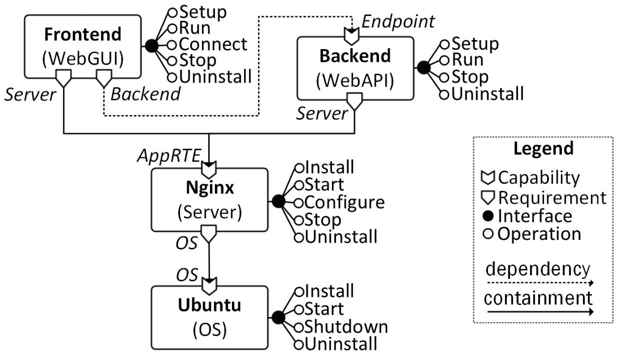


Figure 1: Topology of the application in our motivating scenario (depicted according to the TOSCA graphical notation [9]).

Interdependencies between application components are also explicitly represented in Fig 1. Each dependency is modelled with a relationship connecting each requirement of each node with the capability satisfying such requirement (e.g., the requirements *Server* and *Backend* of *Frontend* are connected with the capability *AppRTE* of *Nginx* and with the capability *Endpoint* of *Backend*, respectively). Relationships can describe "vertical" containment dependencies, which indicate that a component is contained in another (e.g., *Server* is hosted on *Nginx*), or "horizontal" dependencies, which indicate that a component just requires another (without stating that the former is contained in the latter — e.g., *Frontend* must connect to the *Endpoint* offered by *Backend* to work properly).

Suppose that we wish to orchestrate the deployment of our application with a dedicated management plan. Since the represented application topology does not include any management protocol for its components, one may produce invalid management plans. For instance, while Fig. 2 illustrates three seemingly valid plans, only (c) is a valid plan. Plan (a) is not valid since the operation *Configure* of *Nginx* cannot be executed before *Nginx* itself is running, while plan (b) is not valid since *Frontend* must always wait for *Backend* to be running to *Connect* to it.

Consider now the application configuration reached by executing the plan (c) in Fig. 2. All nodes are deployed, started, and properly connected each other (i.e., all components are in their *running* state). It may happen that:

(i) *Backend* is stopped by executing its *Stop* operation. This results in destroying the connection between *Frontend* and *Backend*, hence faulting *Frontend*, which can no more exploit the *Backend* to answer to the requests of its clients. If this is the case, re-starting *Backend* would not be enough, as the connection between *Frontend* and *Backend* would need to be re-established.

(ii) *Nginx* unexpectedly crashes. This generates a fault also in *Frontend* and *Backend*, viz., in the nodes con-

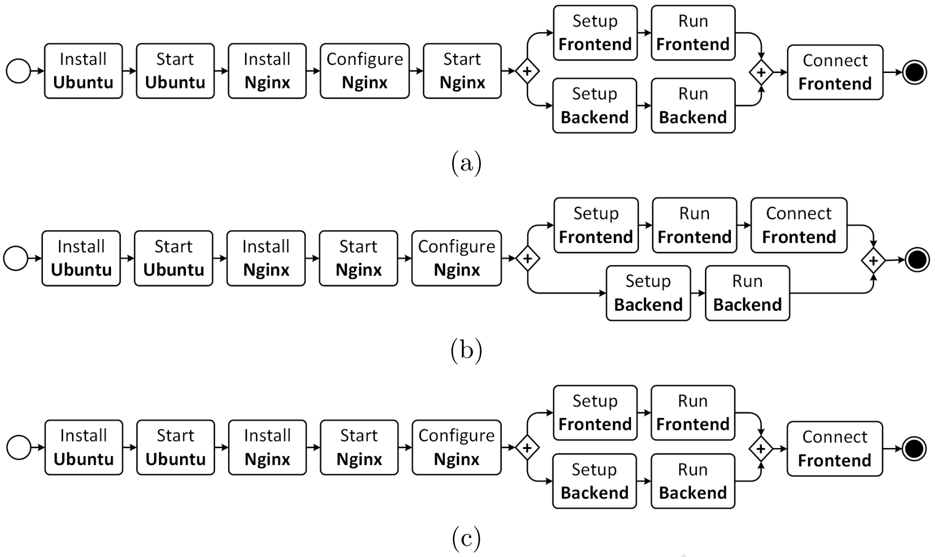


Figure 2: Example of (workflow) plans orchestrating the deployment of the composite application in our motivating scenario.

tained in the *Nginx* server. Such nodes are indeed suddenly killed, and potentially enter in an inconsistent state that makes them unusable from there onwards.

The failures in both (i) and (ii) are due to nodes that stop providing their capabilities, even if other nodes actually rely on them on them to continue to work. The fault in (i) is caused by the execution of a management operation, which stops a node while other are depending on its capabilities. Even worse is the case of (ii), where faults are due to a node that behaves unexpectedly (viz., it unpredictably fails[1]).

In summary, while the validity of plans can be manually verified, this is a time-consuming and error-prone process. To automate the verification of the validity of plans, as well as the generation of valid plans reaching given application configurations, we need an explicit representation of the management protocols of the nodes appearing in the topology of a composite application. Such management protocols have to take into account the possibility of faults to occur, and should permit reacting to them to recover the desired configuration of an application.

## 3. Fault-aware management protocols

Most of the available languages for modelling composite applications allow to indicate the states, requirement, capabilities, and management operations of the nodes building the topology of a composite application (e.g., enterprise topology graphs [3], GENTL [10], TOSCA [7]). We hereby propose *fault-aware management protocols*, which permit specifying the management behaviour of the nodes composing an application, i.e. the behaviour of a node's management operations, their relations with states, requirement, and capabilities, and how a node reacts to the occurrence of a fault.

### *3.1. Definition of fault-aware management protocols*

Consider an application component, and let $N$ be the node modelling such component. To describe the management behaviour of $N$, we need to specify whether/how each management operation of $N$ depends on other management operations (i) of the same node $N$ or (ii) of the nodes that provide capabilities used to satisfy the requirements of $N$. Fault-aware management protocols permit specifying (i) and (ii) as follows.

(i) The first kind of dependencies is described by relating the management operations of $N$ with its states. The order of execution of the operations of $N$ is described by a transition relation $\tau$. The latter specifies whether a management operation $o$ can be performed in a state $s$, and which state is reached by performing $o$ in $s$.

(ii) The second kind of dependencies is specified by associating (possibly empty) sets of requirements with transitions and states. The requirements associated with a transition $t$ must be satisfied to perform $t$, while those associated with a state of $N$ must continue to be satisfied in order for $N$ to continue to work properly. As requirements are satisfied when the corresponding capability is provided, the requirements associated with transitions and states actually indicate which capabilities must be offered (by other nodes) to perform a transition or to continue to reside in a state.

[1] Component failures can be detected via monitoring (e.g., by exploiting watchdogs or heartbeat services). This is particularly useful for misbehaviour of components and unpredictable failures due to non-deterministic bugs [6]. We shall not delve into details, as component monitoring is outside of the scope of this paper.

The description is completed by associating each state $s$ of $N$ with the capabilities actually provided by $N$ in $s$.

Fault-aware management protocols also allow to indicate how $N$ reacts when a fault occurs, i.e. when $N$ is in a state assuming some requirements to be satisfied, and some other node stops providing the capabilities satisfying such requirements. This is described by a transition relation $\varphi$ that models the explicit fault handling of $N$ by specifying how $N$ changes its state from $s$ to $s'$ when some of the requirements it assumes in $s$ stop being satisfied.

**Definition 1** (Fault-aware management protocols)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ be a node, where $S_N$, $R_N$, $C_N$, and $O_N$ are the finite sets of its states, requirements, capabilities, and management operations. $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N\rangle$ is a finite state machine defining the* fault-aware 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 which requirements must hold in each state $s \in S_N$,*
- *$\chi_N : S_N \to 2^{C_N}$ is a function indicating which capabilities of $N$ are offered in each state $s \in S_N$,*
- *$\tau_N \subseteq S_N \times 2^{R_N} \times O_N \times S_N$ is a set of quadruples modelling the transition relation, viz., $\langle s, P, o, s'\rangle \in \tau_N$ denotes that in state $s$, and if the requirements in $P$ are satisfied*[2]*, $o$ is executable and leads to state $s'$, and*
- *$\varphi_N \subseteq S_N \times S_N$ is a set of pairs modelling the fault handling for a node, viz., $\langle s, s'\rangle \in \varphi_N$ denotes that the node will change its state from $s$ to $s'$ if some of the requirements in $\rho_N(s) - \rho_N(s')$ stops being satisfied*[3]*.*

*Example* 1. The fault-aware management protocols of the nodes in our motivating scenario are shown in Fig. 3.

Consider $\mathcal{M}_{Nginx}$, viz., the fault-aware management protocol of *Nginx*. The initial state of *Nginx* is *NotInstalled*, where *Nginx* does not require nor provide anything. The *OS* requirement is instead assumed to (continue to) be satisfied in the states *Installed* and *Started*. If *OS* is faulted, then *Nginx* goes back to its initial state (hence needing to be re-installed and re-started). The *AppRTE* capability is concretely provided by *Nginx* only in the *Started* state. Finally, all management operations of *Nginx* can be executed only if the *OS* requirement is satisfied.

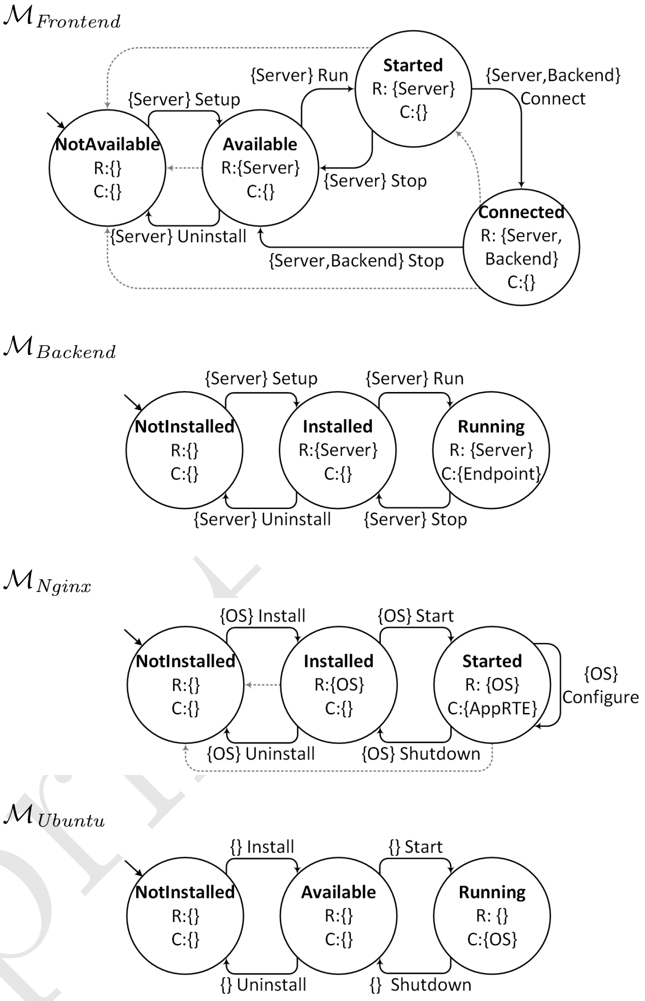


Figure 3: Fault-aware management protocols of the nodes in our motivating example (Fig. 1). Solid arrows represent $\tau$, while dashed arrows represent $\varphi$.

Consider now $\mathcal{M}_{Backend}$, viz., the fault-aware management protocol of *Backend*. One can readily observe that $\mathcal{M}_{Backend}$ is somehow "incomplete": *Backend* assumes the *Server* requirement to (continue to) be satisfied in the *Installed* and *Running* states. However, $\mathcal{M}_{Backend}$ does not indicate what happens if such requirement is faulted (since the corresponding capability stops being provided). Sect. 3.3 shows how to deal with such a kind of' "incomplete" management protocols, by illustrating how to automatically complete them (by adding transitions for default handling all unhandled faults). □

It is worth observing that fault-aware management protocols (as per Def. 1) allow (i) the conditions on requirements of transitions to be inconsistent with respect to those of their source and target states, and (ii) operations to have non-deterministic effects when applied in a state. Additionally, as faults are not going to be propagated syn-

[2] The requirements $P$ needed to perform a transition $\langle s, P, o, s'\rangle \in \tau_N$ obviously need to include those needed in the starting state $s$ and in the target state $s'$. The relationship between $\rho_N(s)$, $\rho_N(s')$, and $P$ is formalised in Sect. 3.2.

[3] The current formalisation of fault-aware management protocols strictly generalises the one we proposed in [8], which required to explicitly indicate the set $F$ of faulted requirements that are handled by a fault handling transition. Indeed, a fault handling transition $\langle s, s'\rangle \in \varphi_N$ now handles all possible sets $F$ of faulted requirements such that $\varnothing \subset F \subseteq \rho_N(s) - \rho_N(s')$, hence easing the specification of fault-aware management protocols (especially because it significantly reduces the amount of fault handling transitions that need to be specified to ensure race-freedom while handling failures — see Sect. 3.2).

chronously (see Sect. 4), (iii) the simultaneous removal of multiple requirements should have the same effect on a node as any sequential removal of the same requirements.

To inhibit (i)-(iii), we assume that fault-aware management protocols enjoy some basic properties (viz., well-formedness, determinism, and race-freedoom), which are presented in the following section.

### *3.2. Characterising fault-aware management protocols*

We now show how to formally define the constraints to ensure well-formedness, determinism and race-freedom of fault-aware management protocols.

A management protocol is *well-formed* if the conditions on requirements of each transition $t$ are consistent with the source and target states. This means that (i) the requirements assumed to hold in the source state of $t$, as well as those assumed to hold in its target state, must be assumed to hold also during the transition, to avoid inconsistencies, and (ii) faults only affect requirements that are assumed in a state, and the handling of such faults should lead to states where faulted requirements are no more assumed and where no additional capabilities are provided.

**Definition 2** (Well-formedness of fault-aware management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, and let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ *be its fault-aware management protocol.* $\mathcal{M}_N$ *is* well-formed *iff*

*(i)* $\forall \langle s, P, o, s' \rangle \in \tau_N :$
$\rho_N(s) \cup \rho_N(s') \subseteq P$*, and*

*(ii)* $\forall \langle s, s' \rangle \in \varphi_N :$
$\rho_N(s') \subset \rho_N(s) \ \wedge \ \chi_N(s') \subseteq \chi_N(s)$

It can be easily verified that all protocols in Fig. 3 are well-formed, since (i) whatever transition $t \in \tau_*$ we consider, the set of requirements needed to fire $t$ is the union of the requirements assumed in the source and target states of $t$, and since (ii) whatever transition $f \in \varphi_*$ we take, the target state of $f$ assumes less requirements and provides less capabilities with respect to the source state of $f$.

A management protocol is also *deterministic* if (i) management operations have deterministic effects when applied in a state (viz., a state cannot have two outgoing transitions corresponding to the same operation and leading to different states). Additionally, (ii) fault handling transitions have to be uniquely determined by the sets of requirements that are no more satisfied. Notice that, since a fault handling transition $\langle s, s' \rangle \in \varphi_N$ handles the fault of some of the requirements in $\rho(s) - \rho(s')$, condition (ii) checks that there are no fault handling transitions outgoing from a state and leading to two states that assume the same set of requirements.

**Definition 3** (Determinism of fault-aware management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, and let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ *be its well-formed fault-aware management protocol.* $\mathcal{M}_N$ *is also* deterministic *iff*

*(i)* $\forall \langle s_1, P_1, o_1, s_1' \rangle, \langle s_2, P_2, o_2, s_2' \rangle \in \tau_N :$
$(s_1 = s_2 \wedge o_1 = o_2) \Rightarrow s_1' = s_2'$.

*(ii)* $\forall \langle s_1, s_1' \rangle, \langle s_2, s_2' \rangle \in \varphi_N :$
$\big(s_1 = s_2 \wedge \rho_N(s_1') = \rho_N(s_2')\big) \Rightarrow s_1' = s_2'$.

It can be easily verified that all protocols in Fig. 3 are deterministic since there is no pair of transitions which start from the same source state and lead to different states by (i) applying the same operation or (ii) handling the same faulted requirements.

Finally, it is worth noting that faults may not propagate synchronously, i.e. when a capability is removed, the nodes assuming the requirements satisfied by such capability eventually detect the removal, but in the meanwhile other capabilities might disappear (potentially raising other faults to be handled). For this reason, (the fault handling in) management protocols should be *race-free*, which means that the simultaneous removal of multiple requirements should have the same effect on a node as any sequential removal of the same requirements, if no operations are executed on the node in the meantime. More precisely, (i) the relation $\varphi_N$ has to be transitive, (ii) if the fault of a set of requirements can be handled in a state $s$, then the same faulted requirements have to be handled in all states that can be reached from $s$ and that have not yet handled it, (iii) if the removal of two sets of requirements is handled in a state, then the removal of their union has to be handled in the same state, and (iv) the same must hold for their intersection.

**Definition 4** (Race-freedom of fault-aware management protocols)**.** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, and let* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ *be its management protocol.* $\mathcal{M}_N$ *is* race-free *iff*

*(i)* $\forall \langle s, s' \rangle, \langle s', s'' \rangle \in \varphi_N :$
$\langle s, s'' \rangle \in \varphi_N$

*(ii)* $\forall \langle s, s' \rangle, \langle s, s'' \rangle \in \varphi_N :$
$\rho_N(s') \supsetneq \rho_N(s'') \Rightarrow \langle s', s'' \rangle \in \varphi_N$

*(iii)* $\forall \langle s, s' \rangle, \langle s, s'' \rangle \in \varphi_N :$
$\exists \langle s, s''' \rangle \in \varphi_N : \rho_N(s''') \subseteq \rho_N(s') \cap \rho_N(s'')$

*(iv)* $\forall \langle s, s' \rangle, \langle s, s'' \rangle \in \varphi_N \ \wedge \ \rho_N(s) \supsetneq \rho_N(s') \cup \rho_N(s'') :$
$\exists \langle s, s''' \rangle \in \varphi_N : \rho_N(s''') \supseteq \rho_N(s') \cup \rho_N(s'')$

Intuitively, the rules in Def. 4 ensure that, as long as no operation occurs in-between, handling faults always leads to the same state, even if novel faults occur in the meantime. This can be easily observed by looking at Fig. 4:

*(i)* In a state $s$, handling the fault of a requirement $r_2$ and then the fault of a requirement $r_1$ should lead to the same state $s''$ as simultaneously handling both faults. If this was not the case, depending on whether the fault of $r_1$ occurs before or after handling the fault of $r_2$, the node may end in different states (hence meaning that its protocol is not race-free).

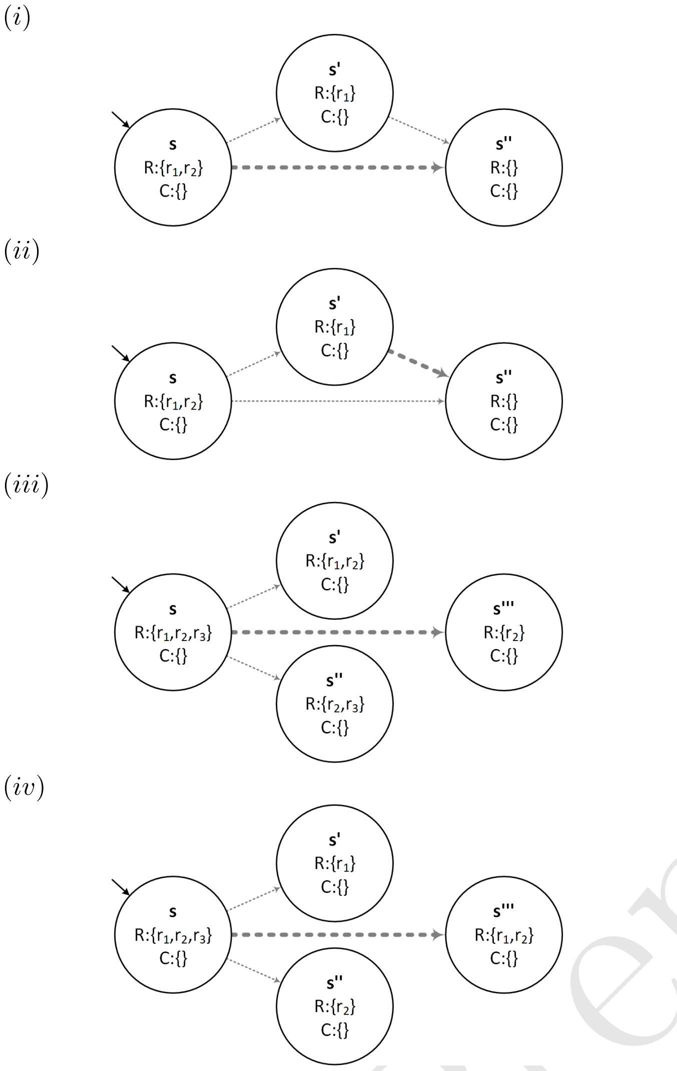


Figure 4: Examples of fault-aware management protocols illustrating the meaning of rules (*i*), (*ii*), (*iii*) and (*iv*) of Def. 4. Lighter arrows satisfy the pre-conditions of each rule, while thicker arrows satisfy their post-condition.

(*ii*) For similar reasons, if the faults of two requirements $r_1$ and $r_2$ can be handled simultaneously by leading a node from state $s$ to state $s''$, and if the fault of $r_2$ can also be handled "per se" by leading the same node from state $s$ to state $s'$, then it should be possible to handle the fault of $r_1$ in state $s'$ and its handler should end in $s''$. This would again ensure that, independently on whether the fault of $r_1$ occurs before or after the handling of the fault of $r_2$, the node ends in the same state $s''$.

(*iii*) If there exist two different transitions handling the fault of a requirement $r_1$ and that of a requirement $r_2$ in a state $s$, then there should be a third transition simultaneously handling the fault of both requirements in $s$. If this is not the case, then the node may not be capable of handling the simpler faults of either $r_1$ or $r_2$ and their combination (viz., the fault of both $r_1$ and $r_2$) in a deterministic way.

(*iv*) Suppose that there exist two fault handling transitions outgoing from a state $s$, one simultaneously handling the fault of both requirements $r_1$ and $r_3$ and the other simultaneously handling the fault of both requirements $r_2$ and $r_3$. If this is the case, then there should be a third transition handling the fault of only $r_3$. Otherwise, the node may not be capable of handling the simpler fault of $r_3$ in a deterministic way (as it would have to non-deterministically choose one between the two fault handling transitions in our hypothesis).

*Remark* 1. More in general, conditions (i) and (ii) in Def. 4 ensure that the simultaneous removal of multiple requirements have the same effect on a node as their two different sets of requirements have the same effect on a node as any sequential removal of the same requirements (if no operations are executed in the meantime). Condition (iii-iv) instead ensure that a node can deterministically choose how to handle simple faults and their compositions. All conditions (i-iv) are hence needed to ensure that fault-aware management protocols are race-free[4]. □

It can be easily verified that all protocols in Fig. 3 are race-free.

### 3.3. Completing fault-aware management protocols

As illustrated by Example 1, the management protocol of a node may leave unspecified how the component will behave in case some requirements stop being fulfilled in some states. To explicitly model that (by also preserving well-formedness, determinism, and race-freedom) management protocols can be completed by adding default transitions for all unhandled faults. All such transitions lead a node to its "sink" state $s_{\lightning}$ that requires and provides nothing (as this models the —worst-case– scenario where a node stops interacting with the other nodes in an application because of an unhandled fault).

**Definition 5** (Completing fault-aware management protocols). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is its fault-aware management protocol. The management protocol $\mathcal{M}_N$ can be* completed *by replacing $S_N$, $\rho_N$, $\chi_N$ and $\varphi_N$ with*

$$\begin{aligned} S'_N =\ & S_N \cup \{s_{\lightning}\}, \\ \rho'_N\ & s.t.\ \forall s \in S_N : \rho'_N(s) = \rho_N(s)\ \wedge\ \rho'_N(s_{\lightning}) = \varnothing, \\ \chi'_N\ & s.t.\ \forall s \in S_N : \chi'_N(s) = \chi_N(s)\ \wedge\ \chi'_N(s_{\lightning}) = \varnothing,\ and \\ \varphi'_N =\ & \varphi_N \cup \{\langle s, s_{\lightning} \rangle \mid s \in S_N \wedge \rho_N(s) \neq \varnothing\ \wedge \\ & \qquad\qquad \nexists \langle s, s' \rangle \in \varphi_N : \rho_N(s') = \varnothing\} \end{aligned}$$

*where $s_{\lightning} \notin S_N$.*

[4] Notice also that partially specified $\varphi$ relations can be automatically completed by applying the rules in Def. 4.

In the following we will assume fault-aware management protocols to be automatically completed as defined above. Intuitively speaking, this will ensure that composite applications will always be able to propagate whatever fault of their nodes, as from each state of a node $N$ it will always be possible to react to the removal of any of its requirements (through one of the transitions in the original $\varphi_N$, or through those introduced in $\varphi'_N$).

*Example* 2. The fault-management protocol of *Backend* ($\mathcal{M}_{Backend}$ in Fig. 3) is completed as illustrated in Fig. 5. We include a new state ($Backend_{↯}$), and two new transitions handling the fault of the *Server* requirement in the states *Installed* or *Running* by making *Backend* end in $Backend_{↯}$. The latter acts as a "sink", as while there are transitions allowing *Backend* to end in the $Backend_{↯}$ state, there is no transition outgoing from such state.

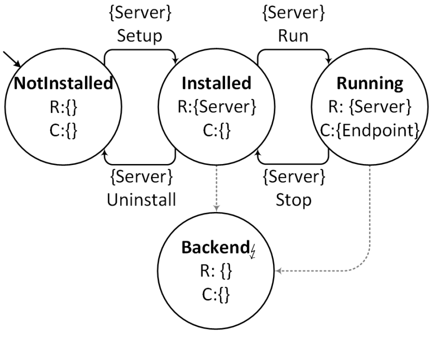


Figure 5: Completion of the fault-aware management protocol $\mathcal{M}_{Backend}$ (Fig. 3), obtained by applying the rules in Def. 5.

All other management protocols in Fig. 3 are completed in a simpler way. Each protocol indeed handles all potential faults, hence making their completion only requiring to add a sink state to each of them (viz., the state $Frontend_{↯}$ is added to the management protocol of *Frontend*, $Nginx_{↯}$ to that of *Nginx*, and $Ubuntu_{↯}$ to that of *Ubuntu*). □

## 4. Analysing application management

In this section we illustrate how to analyse and automate the management of composite applications in a fault-resilient manner. Namely, we show how the fault-aware management behaviour of a composite application can be determined by composing the protocols of its nodes according to the application topology (Sect. 4.1). We then describe how to determine whether a plan orchestrating the management of a composite application is valid, which are the effects of a management plan, and how this also permits finding plans to achieve specific goals (Sect. 4.2).

Even if application components are described by fault-aware management protocols, the actual behaviour of components may differ from the described one (e.g., because of non-deterministic bugs [6]). In Sect. 4.3 we show how the unexpected behaviour of a component can be modelled by automatically completing its management protocols, and how this permits analysing the (worst possible) effects of a misbehaving component on the rest of an application. We also illustrate a way to hard recover applications that are stuck because a fault was not properly handled, or because of misbehaving components (Sect. 4.4).

### *4.1. Management behaviour of an application*

We hereby show how to determine the fault-aware management behaviour of an application by composing the fault-aware management protocols of its components. To simplify the formalisation, we exploit some shorthand notations to denote generic composite applications, the nodes in their topology, and the connections among the requirements and capabilities of such nodes (e.g., to denote that *AppRTE* is the capability connected to the *Server* requirements in our motivating scenario — Fig. 1).

**Notation 1.** *We denote with $A = \langle T, b\rangle$ a generic composite application, where $T$ is the finite set of nodes in the application topology, and where the connections among nodes is described by a (total)* binding *function*

$$b : \bigcup_{N \in T} R_N \to \bigcup_{N \in T} C_N$$

*associating each requirement of each node with the capability that satisfies such requirement.*

*Remark* 2. For simplicity, and without loss of generality, we assume that the names of states, requirements, capabilities, and operations of a node are all disjoint. We also assume that, given two different nodes in a topology, the names of their states, requirements, capabilities, and operations are naturally disjoint. □

Formally, the semantics of the management protocols in a composite application $A = \langle T, b\rangle$ can be defined by a labelled transition system over configurations that denote the states of the nodes in $T$. Intuitively, $G \xrightarrow{o}_A G'$ is a transition denoting that operation $o$ can be executed (on a node) in $A$ when the "global" state of $A$ is $G$, making $A$ evolve into the new global state $G'$. Hence, we first need to formally define the notion of *global state* for a composite application. The latter is essentially a set $G$ containing only the current state of each of the nodes forming a composite application.

**Definition 6** (Global state)**.** *Let $A = \langle T, b\rangle$ be a composite application, and let us denote with $\langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ the tuple corresponding to a node $N \in T$. A* global state *$G$ of $A$ is a set of states such that:*

$$G \subseteq \bigcup_{N \in T} S_N \wedge \forall N \in T : |G \cap S_N| = 1$$

*We denote by $\overline{G}$ the initial global state of $A$, where each node in $T$ is in its initial state (viz., $\overline{G} = \bigcup_{N \in T} \overline{s}_N$).*

*Remark* 3. The condition $\forall N \in T : |G \cap S_N| = 1$ in Def. 6 ensures that a global state $G$ contains exactly one state for each node $N$ in $T$ (which is actually the current state of $N$ in $G$). This is because a node cannot be in two or more different states at the same time. □

We also define a function $F$ to denote the set of *pending faults* in $G$, namely the set of requirements assumed in $G$ despite their corresponding capabilities are not provided. To do so, we need some shorthand notations to indicate the requirements assumed and the capabilities provided in a global state, and the set of capabilities bound to a given set of requirements.

**Notation 2.** *Let $G$ be a global state of a composite application $A = \langle T, b\rangle$. We denote with $\rho(G)$ the set of requirements that are assumed to hold by the nodes in $T$ when $A$ is in $G$, with $\chi(G)$ the set of capabilities that are provided by such nodes in $G$, and with $b(R)$ the set of capabilities bound to the requirements in $R$. Formally:*

$$\begin{aligned}\rho(G) &= \bigcup_{N \in T} \{\rho_N(s) \mid s \in G \wedge s \in S_N\},\\ \chi(G) &= \bigcup_{N \in T} \{\chi_N(s) \mid s \in G \wedge s \in S_N\}, \text{ and}\\ b(R) &= \bigcup_{r \in R} \{b(r)\}.\end{aligned}$$

**Definition 7** (Pending faults)**.** *Let $A = \langle T, b\rangle$ be a composite application, and let $G$ be a global state of $A$. The set $F(G)$ of* pending faults *in $G$ is defined as follows:*

$$F(G) = \{r \in \rho(G) \mid b(r) \notin \chi(G)\}.$$

The management behaviour of a composite application $A = \langle T, b\rangle$ is then given by a labelled transition system, whose configurations are the global states of $A$. The transition system is characterised by two simple inference rules, *op* for operation execution and *fault* for fault propagation. The former permits executing a management operation $o$ on a node $N \in T$ only if there are no pending faults and all the requirements needed by $N$ to perform $o$ are satisfied (by the capabilities provided by other nodes in $T$). The latter illustrates how to execute fault handling transitions when there are pending faults.

**Definition 8** (Management behaviour of a composite application)**.** *Let $A = \langle T, b\rangle$ be a composite application, and let us denote with $\langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ the tuple corresponding to a node $N \in T$. Let also $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N\rangle$. The* management behaviour *of $A$ is modelled by a labelled transition system whose configurations are the global states of $A$, and whose transition relation is defined by the following inference rules:*

$$\frac{\begin{array}{c} s \in G \qquad \langle s, P, o, s'\rangle \in \tau_N \\ F(G) = \varnothing \qquad b(P) \subseteq \chi(G)\end{array}}{G \xrightarrow{o} (G - \{s\}) \cup \{s'\}} \; \textit{(op)}$$

$$\frac{\begin{array}{c} s \in G \qquad \langle s, s'\rangle \in \varphi_N \qquad \rho_N(s') \subseteq \rho_N(s) - F(G) \\ \nexists \langle s, s''\rangle \in \varphi_N . \rho_N(s') \subsetneq \rho_N(s'') \subseteq \rho_N(s) - F(G)\end{array}}{G \xrightarrow{\bot} (G - \{s\}) \cup \{s'\}} \; \textit{(fault)}$$

The *op* rule indicates how to update the global state of an application $A$ when a node $N$ executes a transition $\langle s, P, o, s'\rangle \in \tau_N$. A transition $\langle s, P, o, s'\rangle$ can be executed only if both following conditions hold:

- $F(G) = \varnothing$, i.e. there are no pending faults in $G$, and
- $b(P) \subseteq \chi(G)$, i.e. all requirements needed to perform the transition are satisfied in $G$.

The actual execution of $\langle s, P, o, s'\rangle$ updates the current state of $N$ from $s$ to $s'$, hence requiring the global state of $A$ to be updated (viz., $G' = (G - \{s\}) \cup \{s'\}$), and potentially triggering faults to be handled (if $F(G') \neq \varnothing$).

The *fault* rule instead models fault propagation, by indicating how to update the global state of an application $A$ when executing the fault handling transition $\langle s, s'\rangle$ of a node $N$. Such transition can be performed only if both following conditions hold:

- $\rho_N(s') \subseteq \rho_N(s) - F(G)$, which ensures that $\langle s, s'\rangle$ handles all faults pending in $G$ and affecting $N$. The target state $s'$ indeed does not assume any of the requirements in the set $F(G)$ of pending faults (since $\rho_N(s')$ is contained in the set difference $\rho_N(s) - F(G)$).
- $\nexists \langle s, s''\rangle \in \varphi_N . \rho(s') \subsetneq \rho(s'') \subseteq \rho(s) - F(G)$, which ensures that, among all executable fault handling transitions, $\langle s, s'\rangle$ is the transition whose target state $s'$ assumes the biggest set of requirements. In this way, the fault handling transition is guaranteed to handle all the faults on the node, while at same time minimising the amount of requirements that stop being assumed (even though the corresponding capabilities continue to be provided).

By executing $\langle s, s'\rangle$ the current state of $N$ changes from $s$ to $s'$, hence requiring to update the global state of $A$ accordingly (viz., $G' = (G - \{s\}) \cup \{s'\}$). Also, the faults of the requirements in $\rho_N(s) - \rho_N(s')$ are not pending any more, and new faults may be triggered (if $F(G') \neq \varnothing$).

*Example* 3. Fig. 6.(a) shows the evolution of a global state $G$ when executing the operation *Stop* of *Backend* when all components are up and running. The pre-conditions of the *op* rule are all satisfied, as there are no pending faults in the starting $G$, and since all requirements needed to execute the considered transition are satisfied. By executing the transition, $G$ is updated by changing the current state of

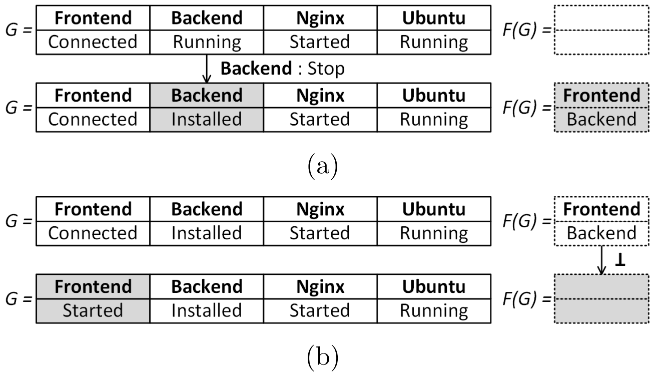

G =
Frontend | Backend | Nginx | Ubuntu
Connected | Running | Started | Running
F(G) =
Backend : Stop
G =
Frontend | Backend | Nginx | Ubuntu
Connected | Installed | Started | Running
F(G) = Frontend | Backend
(a)
G =
Frontend | Backend | Nginx | Ubuntu
Connected | Installed | Started | Running
F(G) = Frontend | Backend
⊥
G =
Frontend | Backend | Nginx | Ubuntu
Started | Installed | Started | Running
F(G) =
(b)

Figure 6: Examples of execution of the rules (a) *op* and (b) *fault* in Def. 8. For reasons of readability, global states and pending faults are represented as tables associating node names with their actual state and with their faulted requirements, respectively. Transitions are displayed as labelled arrows, and each update due to the execution of a transition is highlighted in grey in the target global state.

*Backend* (from *Running* to *Installed*), and a new fault is triggered. *Frontend* is indeed assuming its requirement *Backend* to be satisfied, but *Backend* is not providing the corresponding capability any more.

The triggered fault can be handled by applying the *fault* rule, as illustrated in Fig. 6.(b). *Frontend* actually has two transitions that can handle the fault of the requirement *Backend* when it is in state *Connected*, leading to its states *Started* and *NotAvailable*, respectively (see $\mathcal{M}_{\text{Frontend}}$ in Fig. 3). As the state *Started* is assuming more requirements than the state *NotAvailable*, the *fault* rule mandates to execute the transition leading to *Started*, hence updating the global state $G$ as shown in Fig. 6.(b). □

### *4.2. Analysing management plans*

The management behaviour of a composite application $A$ (Def. 8) lays the foundations for analysing the management of $A$. The management behaviour of $A$ indeed permits defining the notion of *validity* for sequences of management operations and for management plans as follows.

**Notation 3.** *Given a composite application $A = \langle T, b \rangle$, we denote with $P_A$ a management plan for A, viz., a workflow orchestrating the management operations of the nodes in $T$ to carry out a management task for A.*

**Definition 9** (Valid plan)**.** *Let $A = \langle T, b \rangle$ be a composite application. The sequence $o_1 o_2 ... o_n$ of management operations in $A$ is* valid *in a global state $G_0$ of $A$ iff*

$$\exists G_1, G_2, ...G_n : G_0 \overset{o_1}{\longmapsto} G_1 \overset{o_2}{\longmapsto} G_2 \overset{o_3}{\longmapsto} \dots \overset{o_n}{\longmapsto} G_n$$

*where*

$$\frac{G \xrightarrow{o} G'}{G \overset{o}{\mapsto} G'}(o) \qquad \frac{G \overset{o}{\mapsto} G' \quad G' \xrightarrow{\perp} G''}{G \overset{o}{\mapsto} G''}(\perp)$$

*A management plan $P_A$ is* valid *in $G_0$ iff all its sequential traces are valid in $G_0$.*

*Remark* 4. The second rule ($\perp$) of the transition system $\overset{o}{\mapsto}$ in Def. 9 permits focusing on the transitions corresponding to the execution of operations (by abstracting from all transitions $G' \xrightarrow{\perp} G''$ that handle faults possibly raised after the execution of an operation $o$). □

In case we wish to ensure that no fault is raised while executing a management plan, we should require such plan to be *fault-free* too.

**Definition 10** (Fault-free plan)**.** *Let $A$ be a composite application, and let $G$ be a global state. Let also $P_A$ be a valid management plan in $G$. $P_A$ is* fault-free *if no fault handling transition $G \xrightarrow{\perp} G'$ is performed in any of its sequential traces.*

*Example* 4. Consider the management plans in Fig. 2. One can readily check that plan (c) is valid and fault-free, since all its sequential traces are valid sequences of operations, and since none of its sequential traces is executing a fault handling transition.

One of the sequential traces of plan (c) is illustrated in Fig. 7. The figure shows how the global state $G$ of the motivating scenario changes while executing the operations in the trace, and how the set $F(G)$ of pending faults remains empty throughout the execution of the whole trace.

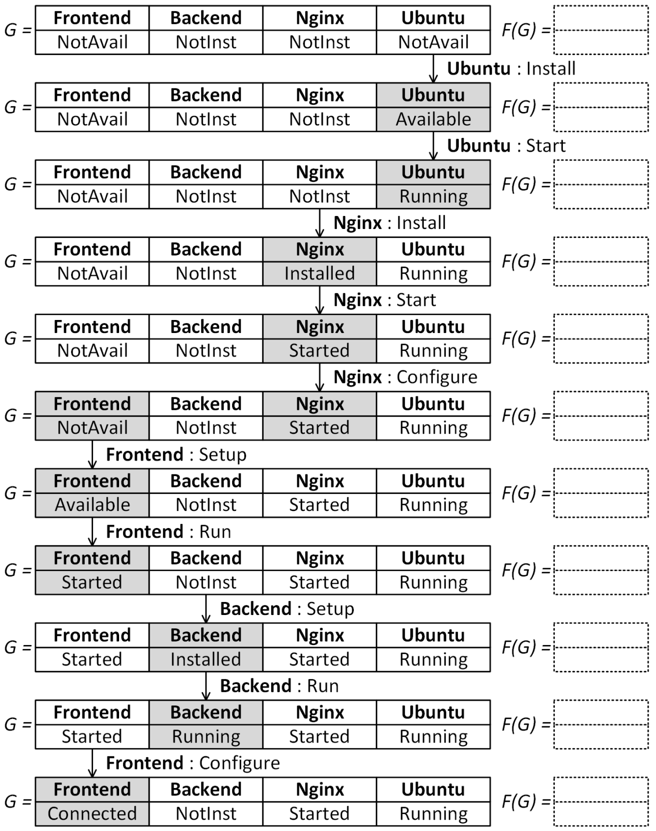

Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | NotAvail
Ubuntu : Install
NotAvail | NotInst | NotInst | Available
Ubuntu : Start
NotAvail | NotInst | NotInst | Running
Nginx : Install
NotAvail | NotInst | Installed | Running
Nginx : Start
NotAvail | NotInst | Started | Running
Nginx : Configure
NotAvail | NotInst | Started | Running
Frontend : Setup
Available | NotInst | Started | Running
Frontend : Run
Started | NotInst | Started | Running
Backend : Setup
Started | Installed | Started | Running
Backend : Run
Started | Running | Started | Running
Frontend : Configure
Connected | NotInst | Started | Running
G =
F(G) =

Figure 7: Evolution of the global state $G$ and of the set of pending faults $F(G)$ according to a valid sequential trace of plan (c) in Fig. 2.

Conversely, plans (a) and (b) in Fig. 2 are not valid since their traces are not valid. More precisely, plan (a) is not valid since all its sequential traces produce the derivation shown in Fig. 8, and *Nginx:Configure* cannot be executed in the reached global state (because it requires *Nginx* to be in state *Working*, instead of *Stopped*).

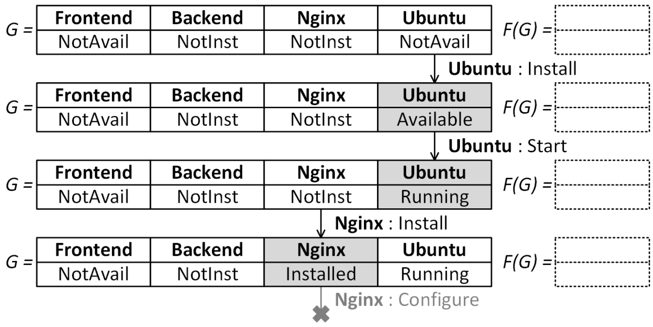

G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | NotAvail
F(G) =
Ubuntu : Install
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | Available
F(G) =
Ubuntu : Start
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | Running
F(G) =
Nginx : Install
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | Installed | Running
F(G) =
Nginx : Configure

Figure 8: Initial evolution of the global state $G$ and of the set of pending faults $F(G)$ according to plan (a) in Fig. 2.

On the other hand, plan (b) is not valid since some of its sequential traces (such as that shown in Fig. 9) reach a global state where *Frontend:Configure* cannot be executed. The latter is because *Frontend:Configure* requires the capability satisfying the *Backend* requirement of *Frontend* to be actually provided, but that capability is not provided when the node *Backend* is not in state *Running* (see $\mathcal{M}_{Frontend}$ and $\mathcal{M}_{Backend}$ in Fig. 3). □

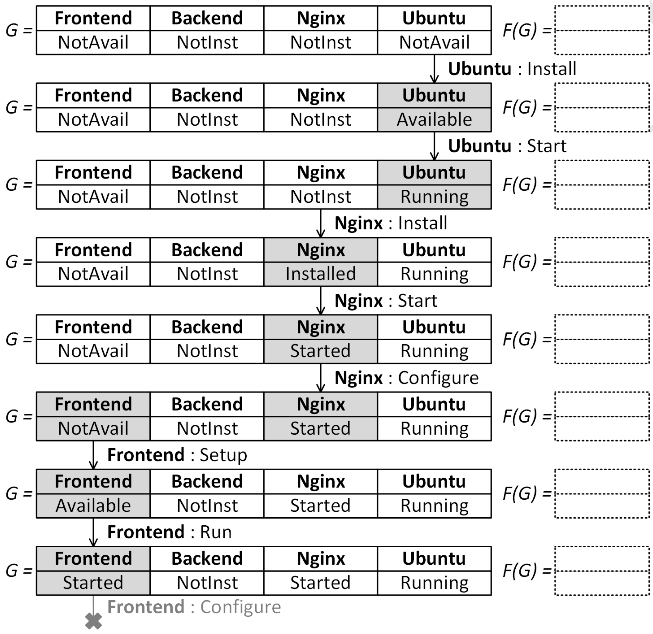

G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | NotAvail
F(G) =
Ubuntu : Install
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | Available
F(G) =
Ubuntu : Start
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | NotInst | Running
F(G) =
Nginx : Install
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | Installed | Running
F(G) =
Nginx : Start
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | Started | Running
F(G) =
Nginx : Configure
G =
Frontend | Backend | Nginx | Ubuntu
NotAvail | NotInst | Started | Running
F(G) =
Frontend : Setup
G =
Frontend | Backend | Nginx | Ubuntu
Available | NotInst | Started | Running
F(G) =
Frontend : Run
G =
Frontend | Backend | Nginx | Ubuntu
Started | NotInst | Started | Running
F(G) =
Frontend : Configure

Figure 9: Initial evolution of the global state $G$ and of the set of pending faults $F(G)$ according to plan (b) in Fig. 2.

The management behaviour of composite applications (Def. 8) can be exploited not only for checking the validity and fault-freedom of management plans, but also for various other purposes. For instance, as there is no assurance that all the sequential traces of a management plan end in the same global state, it is interesting to characterise *deterministic* plans.

**Definition 11** (Deterministic plan)**.** *Let $G$ be a global state of a composite application $A$. A valid management plan $P_A$ is* deterministic *from $G$ iff all its sequential traces end in the same global state.*

*Remark* 5. An obvious approach for checking validity, fault-freedom or determinism of management plans is a visit of the graph associated with the transition system modelling the management behaviour of an application (Def. 8). The constraints on fault-aware management protocols (which are assumed to be well-formed, deterministic and race-free), along with the way they are combined, ensure that such a graph is *finite*, hence guaranteeing its visit to terminate. □

It is also interesting to understand what happens to an application $A$ when executing a valid plan $P_A$. For instance, one may wish to determine the actual states of the components of $A$ after executing $P_A$, which capabilities they provide, and which requirements they assume to be satisfied. Such information can be excerpted directly from the global state(s) reached by the sequential traces of $P_A$.

Moreover, the problem of finding whether there is a deployment plan which starts from the initial global state $\overline{G}$ and achieves a specific management goal (e.g., bringing an application to a certain global state, or making some capabilities available) can be solved with a breadth-first search of the graph of reachable global states. The same approach also works in the case of generic management plans (i.e., plans starting from a generic global state $G$), and it permits finding the sequential plans (if any) allowing to reach a certain goal from whatever starting $G$.

*Remark* 6. Deployment/management plans are automatically determined by visiting the graph of reachable global states, viz., the graph associated with the transition system of the management behaviour of an application (Def. 8). Their validity is hence ensured "by construction", as each path in the graph of reachable global states corresponds to a valid sequence of operations. □

With the above approach, we can also characterise an interesting property that may be exhibited by a composite application $A$. If the initial global state $\overline{G}$ can be reached from any $G$ that is reachable from $\overline{G}$ itself, then all valid sequences of management operations are reversible (as we can always find another sequence of management operations leading back to $\overline{G}$). This in turn means that we can always (soft) reset the application $A$.

**Definition 12** (Soft-resettability)**.** *Let $A$ be a composite application, and let $\overline{G}$ be its initial global state. We say that $A$ is* softly resettable *iff for each global state $G$ reached by executing a valid sequence of operations from $\overline{G}$, there exists a valid sequence of operations from $G$ whose execution leads back to $\overline{G}$.*

The above is a very convenient property, because it guarantees that it is always possible to generate a plan for any reachable goal from any application state.

One can readily check that the application in our motivating scenario (Fig. 1) is not softly resettable, since *Backend* can enter its sink state $Backend_{\lightning}$ (Fig. 5) and there is no way to get it back to its initial state. A way to hard recover applications that are stuck because of a node entering in its sink state is presented in Sect. 4.4.

### *4.3. Modelling and analysing "the unexpected"*

The analyses presented in the previous sections assume that application components behave according to their specified management protocols. However, the actual behaviour of an application component may be different from that modelled in its fault-aware management protocol, e.g., because of non-deterministic bugs [6]. We hereby illustrate a way to deal with such a kind of situations.

The unexpected behaviour of application components can be naturally modelled with fault-aware management protocols. This indeed only requires to add a "crash" operation $\lightning$ to each node, and to automatically complete its fault-aware management protocol with "crash" transitions leading such node to its sink state.

**Definition 13** (Fault-aware management protocols with unexpected behaviour)**.** *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, where $\mathcal{M}_N = \langle \overline{s}_N,\ \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ is its fault-aware management protocol. The management behaviour of $N$ can be extended to include* unexpected behaviour *by replacing $O_N$ and $\tau_N$ with:*

- $O'_N = O_N \cup \{\lightning\}$*, and*
- $\tau'_N = \tau_N \cup \{\langle s, \rho(s), \lightning, s_{\lightning} \rangle \mid s \in S_N\}$*.*

*Remark* 7. The construction rules in Def. 13 permit automatically extending fault-aware management protocols by preserving determinism, well-formedness, and race-freedom of fault-aware management protocols. In particular, to ensure well-formedness, each transition $\langle s, \rho(s), \lightning, s_{\lightning} \rangle$ can be fired only if the requirements in $\rho(s)$ are satisfied. Notice that this is not a restriction since such requirements are satisfied in $s$ by definition of fault-aware management protocols (Def. 1). □

By applying the analyses presented in Sect. 4.2 to fault-aware management protocols including unexpected behaviour (Def. 13), we can analyse and automate the management of a composite application also in presence of misbehaving components. The $\lightning$ transitions indeed model the case of a node behaving unexpectedly, by leading such node to its sink state. As we (pessimistically) assume that a node is no more offering any of its capabilities when it enters its sink state, by firing a $\lightning$ transition we can analyse the (worst possible) effects of a misbehaving node on the rest of an application. The latter indeed only requires to observe the changes that occur to the global state of a composite application after firing a $\lightning$ transition.

*Example* 5. Consider again the fault-aware management protocol of *Backend* (viz., $\mathcal{M}_{Backend}$ in Fig. 3), completed by adding its sink state $Backend_{\lightning}$ as discussed in Example 2. The extension described in Def. 13 simply consists in automatically including a set of "crash" transitions that start from the states *NotInstalled*, *Installed*, and *Running*, and which lead to the $Backend_{\lightning}$ state[5]. The resulting protocol is depicted in Fig. 10.

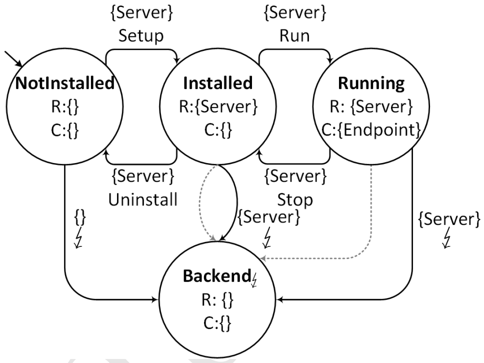
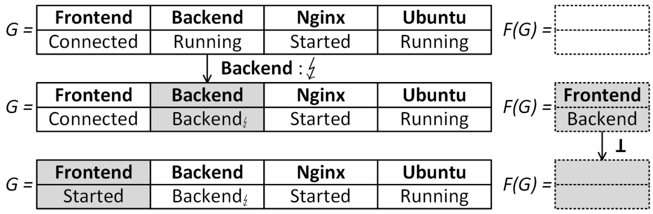


Figure 10: Extension of the fault-aware management protocol $\mathcal{M}_{Backend}$ in Fig. 5, which includes all $\lightning$ transitions modelling the unexpected behaviour of *Backend*.

The above extension allows to analyse the (worst-possible) effects of a misbehaving *Backend* on the rest of the application in our motivating scenario. For instance, we can determine the effects of a "crashing" *Backend* when the whole application is up and running. By executing the operation $\lightning$ of *Backend*, the global state of the application is changed as shown in Fig. 11. Namely, the state of *Backend* is updated, by also filling the set of pending faults with the *Backend* requirement of *Frontend* (since *Frontend* is assuming such requirement in its *Connected* state, but the corresponding capability is no more provided by *Backend*). The pending fault is then consumed by a $\perp$-transition, which updates the state of *Frontend*. □


G =

| Frontend | Backend | Nginx | Ubuntu |
|---|---|---|---|
| Connected | Running | Started | Running |

F(G) =

Backend : ↯

G =

| Frontend | Backend | Nginx | Ubuntu |
|---|---|---|---|
| Connected | Backend↯ | Started | Running |

F(G) =

| Frontend |
|---|
| Backend |

⊥

G =

| Frontend | Backend | Nginx | Ubuntu |
|---|---|---|---|
| Started | Backend↯ | Started | Running |

F(G) =

Figure 11: Example of evolution of the global state $G$ and of the set of pending faults $F(G)$ after injecting a fault in *Backend*.

We now need a solution to recover composite applications that got stuck because one or more of their components are behaving unexpectedly. From the global state

[5] The fault-aware management protocols of the other nodes in our motivating scenario (i.e., *Frontend*, *Nginx* and *Ubuntu*) can be extended analogously.

reached after injecting a failure (by executing a $\lightning$ transition), we may indeed wish to determine a "recovery" plan allowing to reach a given recovery goal (e.g., bringing the application back to the global state in which the failure was injected). Such a plan cannot be determined by directly applying the analyses presented in Sect. 4.2 (e.g., by visiting the graph associated with the labelled transition system modelling the management behaviour of a composite application). This is because, after injecting a crash in a node, the latter is stuck in its sink state (since no transition outgoes from such state). However, fault-aware management protocols can be naturally extended to permit automatically generating recovery plans, as we will see in the next section.

### *4.4. Hard recovering stuck applications*

We hereby illustrate how to automatically determine plans allowing to recover application that are stuck because one of more of their nodes entered their sink state (e.g., because a fault was handled by a transition that led to the node's sink state, or because the node was behaving unexpectedly and a crashing transition $\lightning$ was fired).

The underlying idea is quite simple. If a node $N$ is stuck in its sink state, then the only way to "hard" reset $N$ is to reset the node $N'$ in which $N$ is contained (i.e., by the node in which it is installed or deployed). Indeed, by resetting $N'$, all the nodes that are contained in $N'$ (including the stuck node $N$) are forcibly reset to their initial state. This in turn results in unlocking $N$, hence allowing to recover it (e.g., by re-executing the operations to install and start it).

We hereby show how to automatically extend the modelling of an application so that hard recovery plans can be naturally determined with a visit of the graph associated with the transition system defined by the (extended) management behaviour of the application.

**Notation 4.** *For the sake of readability, we shall denote by* $\mathsf{cont}(N)$ *the node in which* $N$ *is contained (e.g., in our motivating example,* $\mathsf{cont}(\mathit{Frontend}) = \mathit{Nginx}$*). If* $N$ *is not contained in any other node, then* $\mathsf{cont}(N)$ *is not defined (e.g., in our motivating example,* $\mathsf{cont}(\mathit{Ubuntu}) = \bot$*).*

Our aim is to permit hard resetting a node $N \in T$ whenever it is stuck in its sink state, by restarting the node $\mathsf{cont}(N)$ that contains $N$. This can be obtained with our analysis approach, provided that the modelling of the application is updated as follows: The application topology is extended by explicitly representing node containment, and the fault-aware management protocols of the application components are updated to permit forcibly resetting container nodes whenever needed.

To explicitly represent node containment, we model an application $A = \langle T, b\rangle$ into an application $A' = \langle T', b'\rangle$, where $T'$ and $b'$ are built as follows:

- Each node $N$ in the topology $T$ is equipped with a capability $\mathsf{alive}^{\mathsf{c}}_N$ whose purpose is to "attest" whether $N$ is still installed to the nodes it contains. If $N$ is contained in another node $N'$, then $N$ is also equipped with a requirement $\mathsf{alive}^{\mathsf{r}}_N$ whose purpose is to permit checking whether its container $N'$ is still installed.

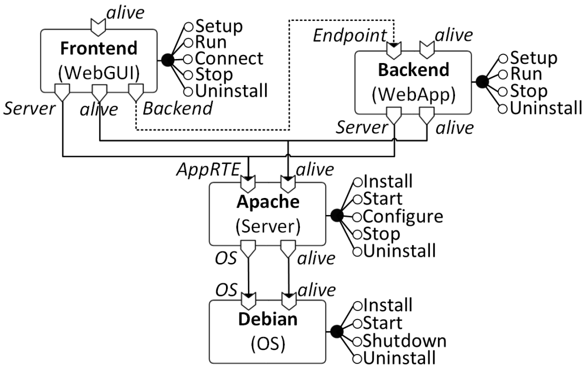


Figure 12: Motivating scenario: updated topology.

- The function $b$ is updated by adding the bindings among the newly introduced requirements and capabilities. Each requirement $\mathsf{alive}^{\mathsf{r}}_N$ is bound to the capability $\mathsf{alive}^{\mathsf{c}}_{N'}$, where $N'$ is the node containing $N$ (viz., $N' = \mathsf{cont}(N)$).

*Example* 6. In our motivating scenario (Fig. 1) the above construction results in updating the application topology as illustrated in Fig. 12. All nodes (but *Ubuntu*) are equipped with *alive* requirements and capabilities. *Ubuntu* is only provided with an *alive* capability. Then, since *Frontend* and *Backend* are contained in *Nginx*, the *alive* requirements of *Frontend* and *Backend* are connected with the *alive* capability of *Nginx*. Additionally, since *Nginx* is contained in *Ubuntu*, the *alive* requirement of *Nginx* is connected to the *alive* capability of *Ubuntu*. □

The updated application topology permits to container nodes to attest that they continue to be installed (by providing their $\mathsf{alive}^{\mathsf{c}}$ capability), and to contained nodes to check whether they continue to be installed (by assuming their $\mathsf{alive}^{\mathsf{r}}$ requirement). This requires to update the management protocol $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N\rangle$ of each node $N \in T$ by substituting it with $\mathcal{M}'_N = \langle \overline{s}_N, \rho'_N, \chi'_N, \tau'_N, \varphi'_N\rangle$, which is built as follows:

- All states in $S_N$ (but the initial one) can be reached by $N$ only if the container of $N$ continues to be installed. Hence, the function $\rho_N$ is updated by making all states (but the initial one) assuming the requirement $\mathsf{alive}^{\mathsf{r}}_N$ in addition to the requirements they already assume.
- Whenever $N$ is not in its initial state, it can be considered as "alive" (as it is ensured that it has performed some operation to get there). To attest this fact, the function $\chi_N$ is updated by making all states (but the initial one) providing the $\mathsf{alive}^{\mathsf{c}}_N$ capability in addition to those they already provide.

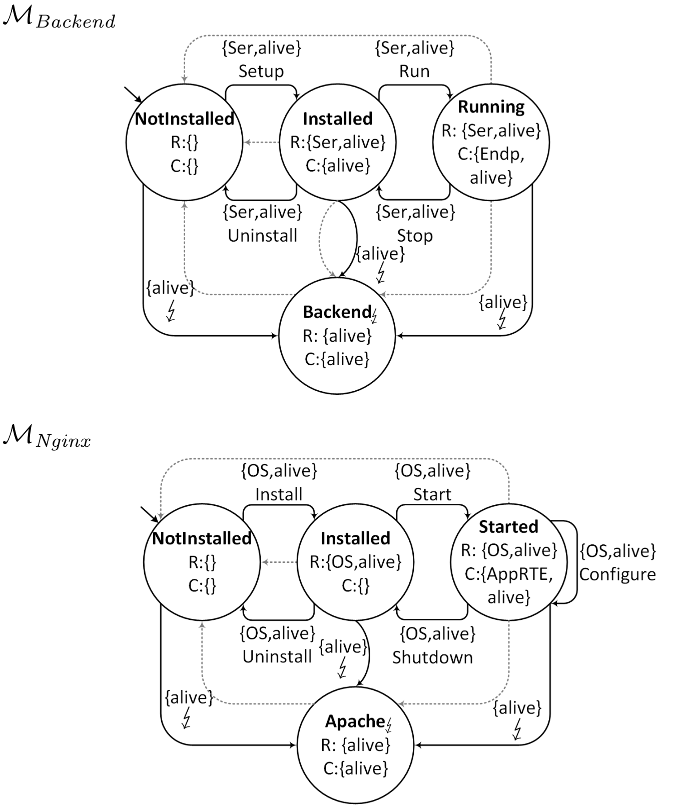


Figure 13: Extension of the fault-aware management protocols $\mathcal{M}_{Backend}$ and $\mathcal{M}_{Nginx}$ obtained by enabling hard recovery. The extension of $\mathcal{M}_{Frontend}$ and $\mathcal{M}_{Ubuntu}$ is similar.

- Each transition in $\tau_N$ requires the container of $N$ (if any) to be alive. This means that each transition in $\tau_N$ has to constrain its executability to the satisfaction of the requirement $\mathsf{alive}^{\mathsf{r}}_N$.
- Finally, the fault handling relation $\varphi_N$ has to be extended to handle the potential fault of the $\mathsf{alive}^{\mathsf{r}}_N$ requirement (if any). If such requirement stops being satisfied, this means that the node in which $N$ is contained has been reset, which in turns means that also $N$ has been (hard) reset. Hence, $\varphi_N$ has to be extended by adding all transitions handling the fault of the requirement $\mathsf{alive}^{\mathsf{r}}_N$ by making $N$ go back to its initial state $\overline{s}_N$.

*Example* 7. The fault-aware management protocols in our motivating scenario can be updated as shown in Fig. 13, which illustrates the updated protocols of *Nginx* and *Backend*. Both protocols are such that the corresponding nodes assume their requirement *alive* to be satisfied (i.e. they assume their containers to continue to be installed) in each of their states but the initial one. They are also providing their capability *alive* in such states, to attest to the nodes they contain (i.e., *Nginx* contains *Frontend* and *Backend*, while *Backend* is not containing any node) that they continue to be there.

Notice that, whenever the requirement *alive* of *Nginx* or *Backend* is faulted, the corresponding node returns to its initial state. Hence, if the node containing *Nginx* is uninstalled, then *Nginx* is uninstalled along with it. The same holds for *Backend*, which is forcibly reset to its initial state whenever its container node is uninstalled. □

The construction explained above is formalised by the following definition, which provides the rules to enable hard recovery in $A = \langle T, b\rangle$.

**Definition 14** (Enabling hard recovery). *Let* $A = \langle T, b\rangle$ *be a composite application, and let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle$ *be a node in* $T$*, with* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N\rangle$*. To* enable hard recovery*,* $A$ *is adapted into a new composite application* $A' = \langle T', b'\rangle$*, where* $T'$ *and* $b'$ *are built according to the following construction rules.*

$T'$ *is built as follows:*

$$\langle S_N, R_N, C_N, O_N, \mathcal{M}_N\rangle \in T \Rightarrow \langle S_N, R'_N, C'_N, O_N, \mathcal{M}'_N\rangle \in T'$$

*where*

$$R'_N = \begin{cases} R_N \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \textit{if } \mathsf{cont}(N) \neq \bot \\ R_N & \textit{otherwise} \end{cases}$$

$$C'_N = C_N \cup \{\mathsf{alive}^{\mathsf{c}}_N\}$$

*and where* $\mathcal{M}'_N = \langle \overline{s}_N, \rho'_N, \chi'_N, \tau'_N, \varphi'_N\rangle$ *is built as follows*

$$\rho'_N(s) = \begin{cases} \rho_N(s) \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \textit{if } s \neq \overline{s}_N \ \wedge \ \mathsf{cont}(N) \neq \bot \\ \rho_N(s) & \textit{otherwise} \end{cases}$$

$$\chi'_N(s) = \begin{cases} \chi_N(s) \cup \{\mathsf{alive}^{\mathsf{c}}_N\} & \textit{if } s \neq \overline{s}_N \\ \chi_N(s) & \textit{otherwise} \end{cases}$$

$\tau'_N = \{\langle s_1, P, o, s_2\rangle \mid \langle s_1, P', o, s_2\rangle \in \tau_N\}$*, with*

$$P' = \begin{cases} P \cup \{\mathsf{alive}^{\mathsf{r}}_N\} & \textit{if } \mathsf{cont}(N) \neq \bot \\ P & \textit{otherwise} \end{cases}$$

$$\varphi'_N = \varphi_N \cup \{\langle s, \overline{s}_N\rangle \mid s \in S_N - \{\overline{s}_N\} \ \wedge \ \mathsf{alive}^{\mathsf{r}}_N \in \rho'_N(s)\}$$

$b'$ *is built as follows:*

$$b'(r) = \begin{cases} \mathsf{alive}^{\mathsf{c}}_{N'} & \textit{if } r = \mathsf{alive}^{\mathsf{r}}_N \textit{ and } \mathsf{cont}(N) = N' \\ b(r) & \textit{otherwise} \end{cases}$$

*Remark* 8. The construction rules for enabling hard recovery (Def. 14) are fully constructive, and they preserve determinism, well-formedness and race-freedom of fault-aware management protocols. Hence, the modelling of a composite application can be automatically updated to enable/disable hard recovery in such application. Such update is done seamlessly and transparently to the owner of the application. □

*Remark* 9. Hard recovery permits recovering a node that is stuck (independently from the kind of failure that made it become stuck), provided that such node is contained in another node. Hence, it cannot be exploited to recover nodes that are not contained in any other node (like *Ubuntu* in our motivating example — Fig. 1). □

The construction rules enabling hard recovery (Def. 14), combined with the analyses presented in Sect. 4.2, permit analysing the management behaviour of a composite application by also considering the possibility of hard resetting a node $N$, to unlock the nodes contained in $N$ that are stuck in their sink states. For instance, the notion of validity (Def. 9) can be reused to check whether a hard recovery plan is valid, and it is also possible to analyse the effects of a hard recovery plan. More interestingly, we can automatically determine a plan recovering the desired global state of an application by simply visiting the graph associated with the labelled transition system modelling the application's management behaviour.

*Example* 8. Consider again our motivating scenario (Fig. 1), and suppose that the application is stuck in the global state reached in Fig. 11. By updating the modelling of the application as illustrated in Examples 6 and 7, it is possible to plan the (hard) recovery of the application from such "stuck" global state.

The only way to let *Backend* exit from its $Backend_{\lightning}$ state is to remove its *alive* requirement. This corresponds to making *Nginx* stop providing its *alive* capability, which in turn makes *Nginx* go back to its initial state. This can be obtained by executing the operations *Shutdown* and *Uninstall* of *Nginx* itself, and it results in resetting also the *Frontend*, which goes back to its initial state. We can then (re-)execute the operations to *Install* and *Start Nginx*, to *Setup* and *Run Backend* and *Frontend*, and to *Connect Frontend* (to *Backend*).

The above listed operations permit building the hard recovery plan in Fig. 14. It can be trivially verified that such plan is valid in the "stuck" global state reached in Fig. 11, and that it permits recovering the application by making all its nodes be up and running again. As we already mentioned, the above recovery plan can be automatically determined by visiting the graph associated with the transition system defined by management behaviour of the application (provided that the modelling of the application has been automatically updated according to the construction rules in Def. 14 — see Examples 6 and 7). □

## 5. Proof-of-concept implementation

We hereby present BARREL, an interactive tool for editing and analysing fault-aware management protocols in composite applications specified in TOSCA[6] (*Topology and Orchestration Specification for Cloud Applications*).

### 5.1. Implementation of BARREL

BARREL is implemented as a web-based application[7], which can be run in any modern web browser (e.g., Microsoft Edge, Google Chrome or Mozilla Firefox). It is composed by two main components, namely a graphical user interface and a back-end (Fig. 15).

The graphical user interface of BARREL is implemented as a HTML5/CSS3 page (*index.html*), which is dynamically populated by a set of JavaScript scripts (*Barrel-react.js*, *Barrel-visualiser.js*, *Barrel-analyser.js*, etc.). The scripts exploit the *ReactJS* library (`https://reactjs.org/`) and the support offered by the back-end of BARREL to reactively update the content and functionalities offered by the GUI.

The back-end of BARREL is implemented with a set of TypeScript modules. Their purpose is essentially to implement all the business logic needed to (i) import/export TOSCA applications, (ii) edit the fault-aware management protocols of their components, and (iii) analyse their management behaviour.

(i) *CSAR.ts* and *TOSCA.ts* implement the logic for importing and exporting TOSCA applications. More precisely, *CSAR.ts* provides the functionalities needed for importing and exporting a CSAR (*Cloud Service ARchive* [7]), which is the standard packaging for TOSCA applications. *TOSCA.ts* instead implements a parser for the TOSCA application specification contained within an imported CSAR.

(ii) *ManagementProtocols.ts* implements a TypeScript class modelling the fault-aware management protocol of an application component. It also allows to edit the fault-aware management protocol of a component, by providing methods that permit updating the sets of requirements/capabilities assumed/provided in a state, and for adding and removing transitions corresponding to the execution of a management operation or to the handling of some faults.

(iii) *Analysis.ts* implements a generic support for all the analyses described in Sect. 4. More precisely, it implements a TypeScript class modelling a composite application, and which permits simulating its management behaviour (e.g., by allowing to check whether a node can perform an operation or whether there are settling handlers for pending faults, and to simulate the execution of the corresponding transitions). *Analysis.ts* also permits determining all the global states that can be reached by an application, as well as the plans allowing to move from each reachable global state to each other[8].

[6] Interested readers can find in [7] the specification of TOSCA, and in [11] a self-contained introduction to TOSCA.

[7] A running instance of BARREL can be accessed at `http://di-unipi-socc.github.io/barrel/`. The source code of BARREL is publicly available on GitHub at `https://github.com/di-unipi-socc/barrel`.

[8] More precisely, *Analysis.ts* permits creating a matrix whose $(i, j)$-th element is the next step on the shortest path from the reachable global state $i$ to the reachable global state $j$, which can then be exploited to reconstruct the shortest path from a starting global state to a target global state. The matrix, as well as the shortest path from a global state to another, are obtained by implementing the Floyd-Warshall algorithm [12].

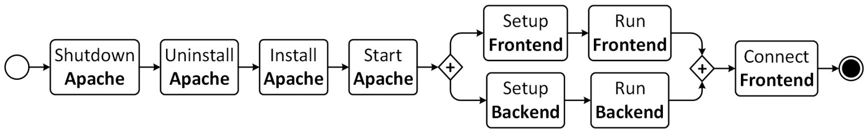


Figure 14: Example of hard recovery plan.

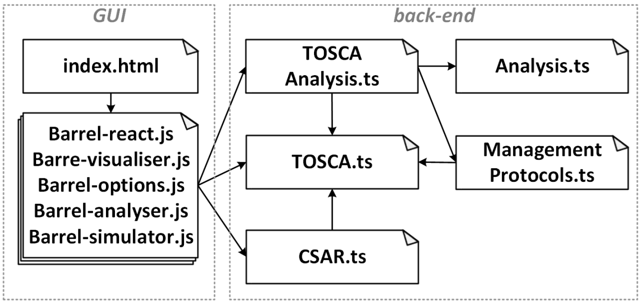


Figure 15: The architecture of the prototype of Barrel. Dashed rectangles represent the main components of Barrel, solid rectangles represent implemented modules, and arrows represent dependencies between modules (by indicating whether a component directly exploits functionalities provided by another component).

*TOSCAAnalysis.ts* then integrates the functionalities provided by *TOSCA.ts*, *ManagementProtocols.ts* and *Analysis.ts* to permit analysing the management behaviour of TOSCA applications.

Barrel is already partially integrated with the OpenTOSCA open-source ecosystem [13, 14, 9]. Indeed, Barrel can process CSARs developed with the visual editor Winery [9], and it produces CSARs that can be edited in Winery, offered by the self-service portal Vinothek [14], and executed by the TOSCA engine OpenTOSCA [13].

Of course, despite Winery, Vinothek and OpenTOSCA can get as input CSARs exported from Barrel, they are not yet capable of processing the behaviour information specified by fault-aware management protocols. This is because Barrel exploits a backward-compatible extension of TOSCA that includes fault-aware management protocols, which is not yet supported by the tools in the OpenTOSCA open-source environment.

### *5.2. How to use* Barrel

As TOSCA applications are shipped within CSAR packages, the very first step is to take an existing CSAR (developed with Winery [9], for instance), and then to import it in Barrel to edit and analyse the management protocols of the TOSCA application it packages. CSAR packages can be imported in Barrel by clicking on the *CSAR* option in the navigation bar of Barrel. Once a CSAR is loaded, the *Visualise*, *Edit*, and *Analyse* panes become selectable in the navigation bar (and the *Visualise* pane is selected by default).

***Visualising applications.*** The *Visualise* pane graphically displays the application specification contained in the imported CSAR (Fig. 16.(a)).

The name of the application is placed in the top-left corner of the *Visualise* pane. The application topology is visualised in the left-hand side of the pane, by drawing all nodes composing such topology, their requirements and capabilities (over and under each node, respectively), and all relationships binding a requirements of a node with a capability of another node. Further information about each node (such as the node type or the management operations it offers) is listed in the table placed in the right-hand side of the *Visualise* pane.

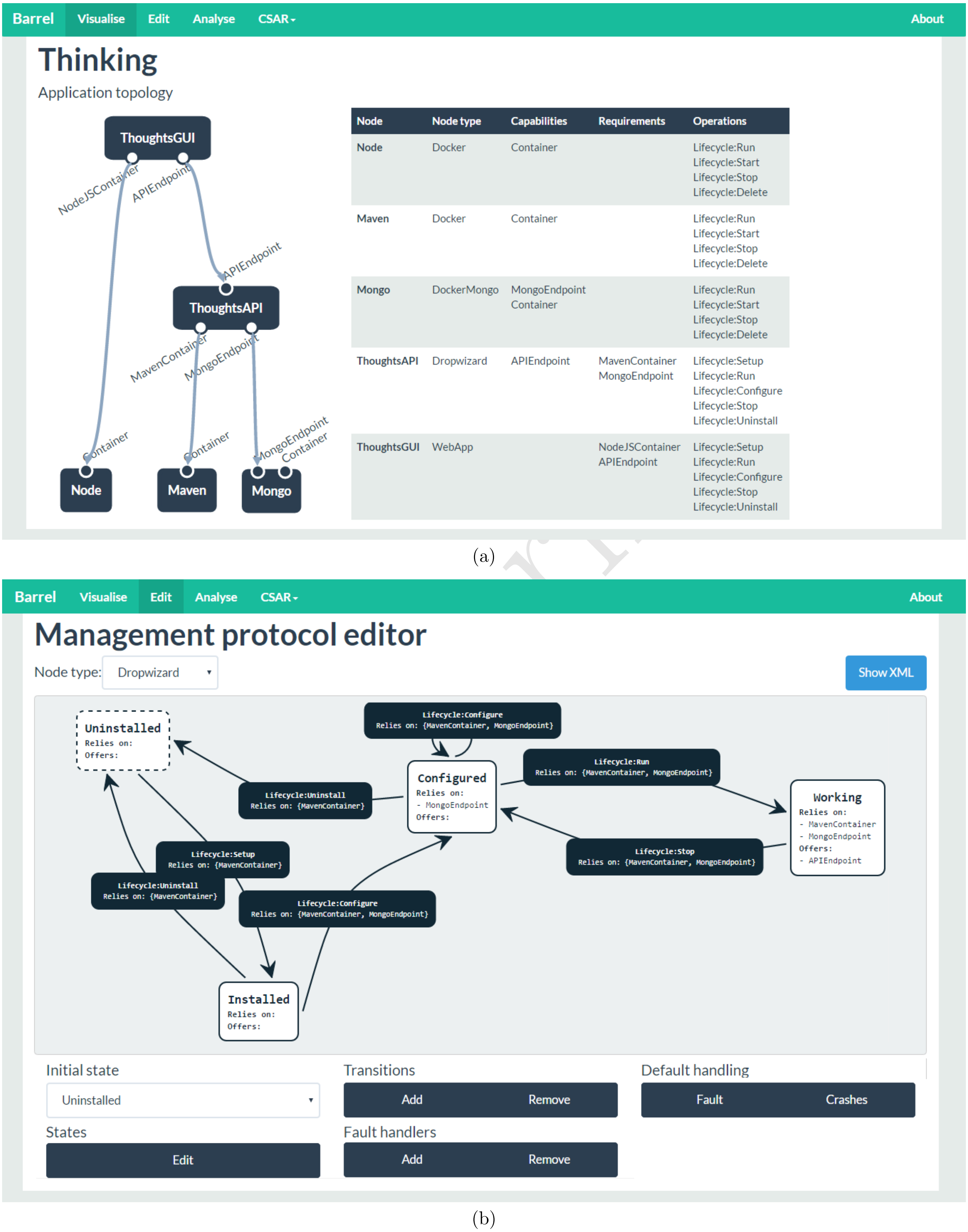
***Editing fault-aware management protocols.*** The *Edit* pane (Fig. 16.(b)) provides all functionalities needed to edit the fault-aware management protocol of each node in the application topology.

The *Management protocol editor* permits selecting the (type of) node to be edited through a dedicated drop-down menu. Once a node type is selected, its fault-aware management protocol is displayed and can be modified by exploiting the toolbars right below it. The initial state can be selected through a dedicated drop-down menu. States can be edited by clicking on the *Edit* button, which opens a popup window that permits editing, for each state $s$, the requirement it assumes and the capabilities it provides (i.e., the values of $\rho(s)$ and $\chi(s)$). Transitions can be added and removed from $\tau$ by clicking on the *Add* and *Remove* buttons, respectively. Similarly, fault-handling transitions can be added and removed from $\varphi$ by clicking on the dedicated buttons. Finally, it is possible to automatically complete the displayed protocol: By clicking on the *Fault* button, all unhandled faults are default handled according to Def. 5. By clicking on *Crashes*, instead, the protocol is updated by adding the crash operation ↯ transitions and all corresponding transitions (to permit analysing the management of the corresponding nodes, also if they are behaving unexpectedly — see Def. 13).

Barrel automatically checks whether the fault-aware management protocols of an application are well-formed, deterministic and race-free. If a protocol does not satisfy either of these properties (e.g., because of an update), an ad-hoc message is displayed right above it when it is selected within the management protocol editor[9].

The updates applied to the fault-aware management protocol of the currently selected node type can also be viewed in the XML source of such node type, by clicking

[9] A complete list of all reasons why one or more fault-aware management protocols in an application are not well-formed, deterministic or race-free is available in the *Analyse* pane.



## Thinking

Application topology

| Node | Node type | Capabilities | Requirements | Operations |
|---|---|---|---|---|
| Node | Docker | Container | | Lifecycle:Run<br>Lifecycle:Start<br>Lifecycle:Stop<br>Lifecycle:Delete |
| Maven | Docker | Container | | Lifecycle:Run<br>Lifecycle:Start<br>Lifecycle:Stop<br>Lifecycle:Delete |
| Mongo | DockerMongo | MongoEndpoint<br>Container | | Lifecycle:Run<br>Lifecycle:Start<br>Lifecycle:Stop<br>Lifecycle:Delete |
| ThoughtsAPI | Dropwizard | APIEndpoint | MavenContainer<br>MongoEndpoint | Lifecycle:Setup<br>Lifecycle:Run<br>Lifecycle:Configure<br>Lifecycle:Stop<br>Lifecycle:Uninstall |
| ThoughtsGUI | WebApp | | NodeJSContainer<br>APIEndpoint | Lifecycle:Setup<br>Lifecycle:Run<br>Lifecycle:Configure<br>Lifecycle:Stop<br>Lifecycle:Uninstall |

(a)

## Management protocol editor

Node type: Dropwizard

Initial state: Uninstalled

(b)

Figure 16: (a) *Visualise* and (b) *Edit* panes of Barrel.

on the *Show XML* button appearing in the top-right corner of the *Edit* pane. Once the fault-aware management protocols of the node types have been edited, the updated CSAR can be downloaded through the *CSAR Export* functionality (in the navigation bar).

***Analysing the management of applications.*** The *Analyse* pane (Fig. 17) permits interactively analysing the fault-aware management behaviour of the imported composite application. The *Analyse* pane contains two separate tools, namely a *Simulator* for interactively simulating the behaviour of the composite application, and a *Planner* for automatically determining (valid) plans reaching desired application configurations. Both simulation and planning can be carried out with hard recovery enabled or disabled (by setting the corresponding toggling option in the *Options* section).

The *Simulator* permits simulating sequences of operations and determining their effects on the whole application. More precisely, the *Simulator*'s table lists all the nodes in the application topology, each associated with its current state, the requirements it currently relies on, the capabilities it offers, and the operations actually available. Each operation is rendered as a green button if all the capabilities connected to requirements needed to execute it are currently available, otherwise it is rendered as a yellow disabled button. By clicking on available operations, users can simulate their execution, and subsequently update the global state displayed by the simulator table.

- The update may violate some requirements (hence generating some faults), which are then displayed as red buttons. If the fault of a requirement can be handled, the corresponding button is clickable, otherwise it is disabled. By clicking on the button corresponding to a requirement, users can simulate the handling of such requirement, hence updating the global state displayed by the *Simulator*.
- An update can also result in permitting to hard recover a node that is stuck in its *Crashed* state. If this is the case, a red button *Hard recover* appears next to the current state of such node, and by clicking on such button the global state of the application is updated by resetting the node to its initial state.

The *Simulator* can be reset at any time, by simply clicking on the button *Reset simulator*.

With the *Simulator*, users can already perform most of the analyses described in Sect. 4. For instance, to check whether an existing plan is valid (see Def. 9), they just need to simulate its sequential traces and check that such traces can be executed in their entirety. To check whether an existing plan is fault-free (see Def. 10), they just need to check that no fault is generated by simulating any of its sequential traces. They can also compute the effects of an existing plan on states, capabilities and requirements by looking at the initial and final configurations displayed by the *Simulator* table.

The *Planner* instead automatically determines new sequential plans that permit reaching the *Target global state* from the *Starting global state*. Both global states can be set by associating each node with one of its states through a dedicated drop-down menu. The management plan reaching the target global state from the starting global state is displayed at the bottom of the planner. Notice that, whenever one of the two global states is changed, the plan is immediately recomputed.

## 6. Barrel at work

### *6.1. Case study*

We hereby illustrate how we fruitfully exploited Barrel to analyse, validate and plan (at design-time) the management of a concrete application[10].

***The* Thinking *application.*** *Thinking* is an open-source[11] web application that allows end-users to share what they are thinking about, so that all other end-users can read it.

*Thinking* is composed by three main components: (i) an instance of MongoDB that is exploited to permanently store the collection of thoughts shared by end-users, (ii) *ThoughtsApi*, which is a Dropwizard-based REST API that permits accessing the collection of shared thoughts, and (iii) *ThoughtsGui*, which is a web-based graphical user interface that interacts with *ThoughtsApi* to permit retrieving and adding thoughts to the shared collection. The MongoDB instance is obtained by instantiating a *Mongo* Docker container, while *ThoughtsApi* and *ThoughtsGui* are made concrete by hosting them on a *Maven* Docker container and on a *Node* Docker container, respectively. The resulting application topology is depicted in Fig. 18.

We hereafter illustrate the fault-aware management protocols of the nodes in the application topology of *Thinking*. All such protocols permit analysing the effects of a misbehaving node on the rest of the application (i.e., they include crash transitions ↯ — see Def. 13).

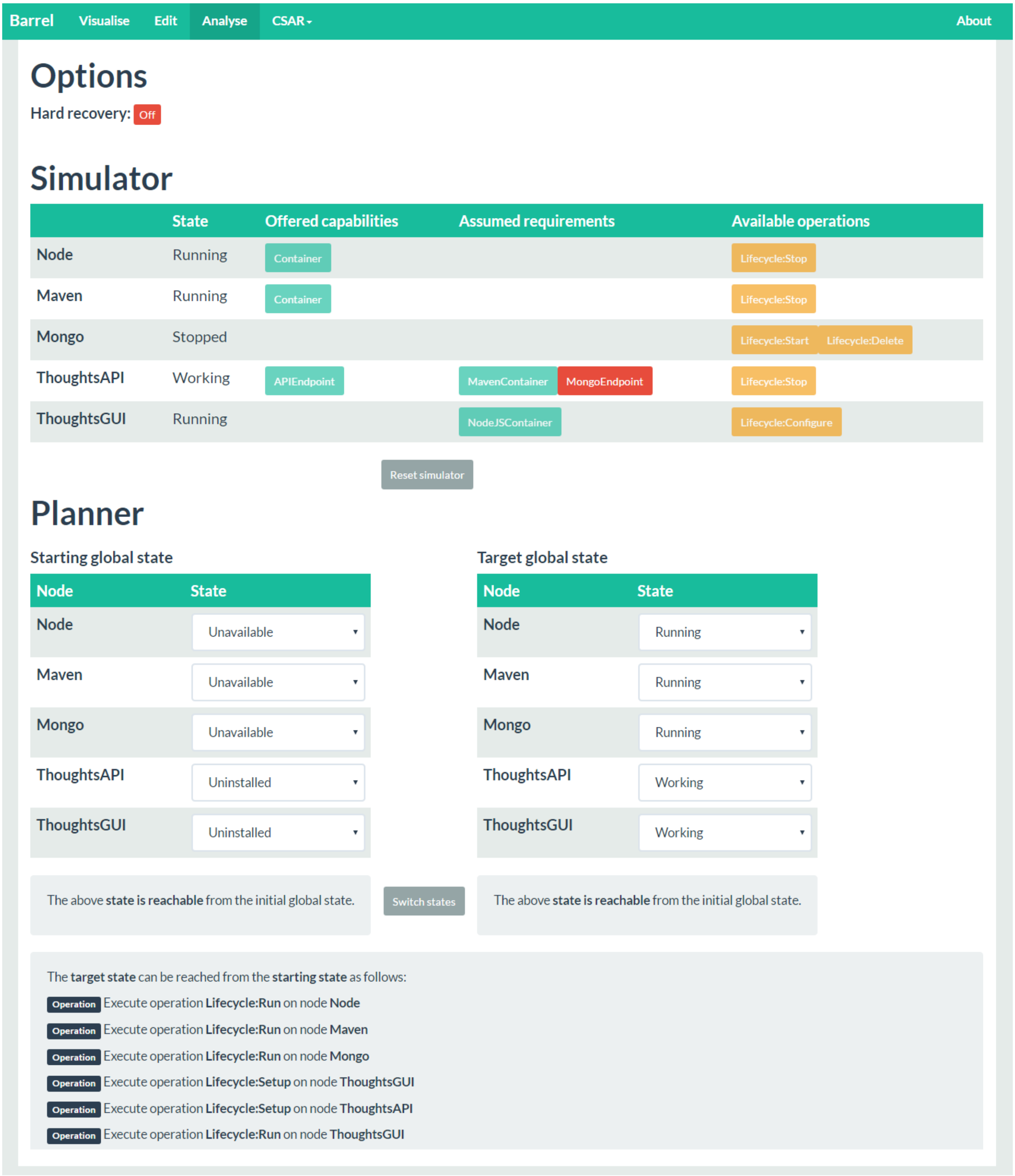
- *Mongo* is a *DockerMongo* node. It offers a capability *MongoEndpoint* (that is used to satisfy the corresponding requirement of *ThoughtsApi*) and the management operations to *Run*, *Start*, *Stop*, and *Delete* the corresponding container[12].

  The fault-aware management protocol of *Mongo* is shown in Fig. 19. Its initial state is *Unavailable*, where *Mongo* is not providing any capability, and where it

[10] The case study has been run on an Ubuntu 16.04 LTS virtual machine, with 32 GB of storage and 8 GB of memory.

[11] The source code of *Thinking* is publicly available in a GitHub repository (`https://github.com/di-unipi-socc/thinking`).

[12] The implementation of *Mongo*'s management operations is as follows. *Run* is implemented by the command line instruction "`docker run -name mongo -p 27017:27017 -d mongo`", *Start* by "`docker start mongo`", *Stop* by "`docker stop mongo`", and *Delete* by "`docker rm mongo`".



## Options

Hard recovery: Off

## Simulator

| | State | Offered capabilities | Assumed requirements | Available operations |
|---|---|---|---|---|
| Node | Running | Container | | Lifecycle:Stop |
| Maven | Running | Container | | Lifecycle:Stop |
| Mongo | Stopped | | | Lifecycle:Start<br>Lifecycle:Delete |
| ThoughtsAPI | Working | APIEndpoint | MavenContainer<br>MongoEndpoint | Lifecycle:Stop |
| ThoughtsGUI | Running | | NodeJSContainer | Lifecycle:Configure |

## Planner

Starting global state

| Node | State |
|---|---|
| Node | Unavailable |
| Maven | Unavailable |
| Mongo | Unavailable |
| ThoughtsAPI | Uninstalled |
| ThoughtsGUI | Uninstalled |

The above **state is reachable** from the initial global state.

Target global state

| Node | State |
|---|---|
| Node | Running |
| Maven | Running |
| Mongo | Running |
| ThoughtsAPI | Working |
| ThoughtsGUI | Working |

The above **state is reachable** from the initial global state.

The **target state** can be reached from the **starting state** as follows:

Operation Execute operation **Lifecycle:Run** on node **Node**

Operation Execute operation **Lifecycle:Run** on node **Maven**

Operation Execute operation **Lifecycle:Run** on node **Mongo**

Operation Execute operation **Lifecycle:Setup** on node **ThoughtsGUI**

Operation Execute operation **Lifecycle:Setup** on node **ThoughtsAPI**

Operation Execute operation **Lifecycle:Run** on node **ThoughtsGUI**

Figure 17: *Analyse* pane of Barrel.

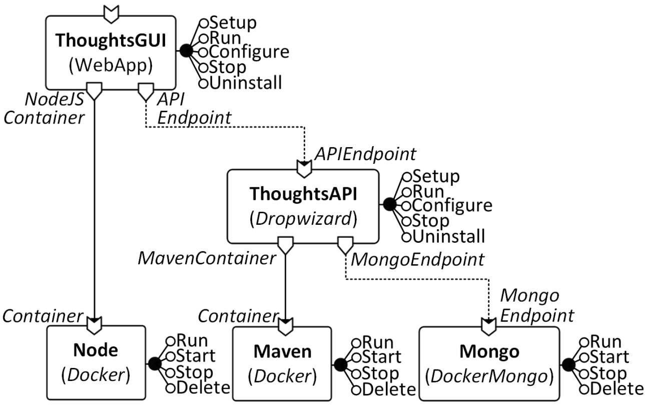


Figure 18: Topology of the *Thinking* application, represented according to the TOSCA graphical notation [9].

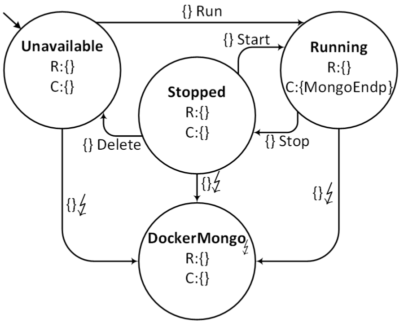


Figure 19: Fault-aware management protocol for nodes of type *DockerMongo* (such as *Mongo* in Fig. 18).

can perform the operation *Run* to become *Running*. In the *Running* state, *Mongo* continues to provide its capability *MongoEndpoint* (hence satisfying the requirements connected to it). It also permits executing the operation *Stop*, which makes *Mongo* transit to the state *Stopped*, where it stops providing the capability *MongoEndpoint*. From the *Stopped* state, *Mongo* can return to be *Running* or *Unavailable*, respectively by executing *Start* or *Delete*.

- *Node* and *Maven* are *Docker* nodes, and their structure is similar to that of *Mongo*. They indeed offer a capability to satisfy other nodes' requirements, and the operations to *Run*, *Start*, *Stop*, and *Delete* them. Their fault-aware management protocol is also very similar to that of *Mongo* (with the only difference that *Node* and *Maven* continue to provide their capability *Container* in their state *Running*, while *Mongo* is offering the capability *MongoEndpoint*).

- *ThoughtsApi* is a *Dropwizard* application implementing a REST API to be exposed on a given port. *ThoughtsApi* hence offers a capability *APIEndpoint* (that is used to satisfy the corresponding requirement of *ThoughtsGui*), and the set of management operations implementing its lifecycle[13]. *ThoughtsApi* also requires a *MavenContainer* to be installed into, and a *MongoEndpoint* to connect to.

  The fault-aware management protocol of *ThoughtsApi* is illustrated in Fig. 21. Its states are *Uninstalled* (initial), *Installed*, *Configured*, *Working*, and *Dropwizard*$_{↯}$. *Uninstalled* is the only state that is not associated with any requirement or capability, while all other states specify the requirements that must continue to be satisfied in order for *ThoughtsApi* to continue to work properly. All states (but *Uninstalled*) also specify the capabilities that they continue to provide. All transitions bind their executability to the availability of the capability satisfying the requirement *MavenContainer*. *Configure*, *Run*, and *Stop* bind their executability also to the availability of the capability satisfying the requirement *MongoEndpoint*. Finally, it is worth noting that all faults are default handled by a transition targeting the "sink" state *Dropwizard*$_{↯}$ (Def. 5).

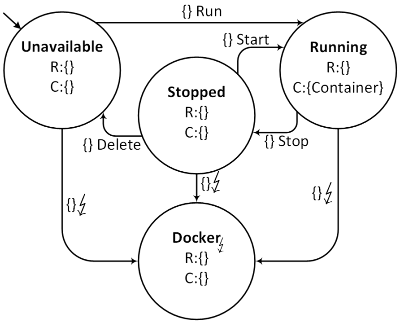


Figure 20: Fault-aware management protocol for nodes of type *Docker* (such as *Node* and *Maven* in Fig. 18).

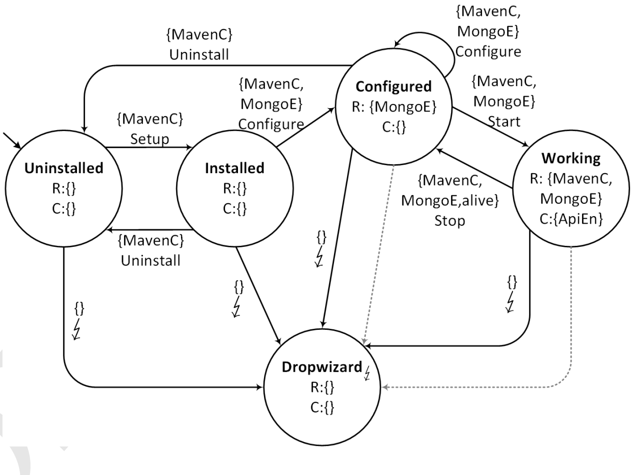


Figure 21: Fault-aware management protocol for nodes of type *Dropwizard* (such as *ThoughtsApi* in Fig. 18).

[13] The bash scripts implementing the operations to *Setup*, *Run*, *Configure*, *Stop*, and *Uninstall* of *ThoughtsApi* are publicly available on GitHub at `https://github.com/di-unipi-socc/thinking/tree/master/api/scripts`.

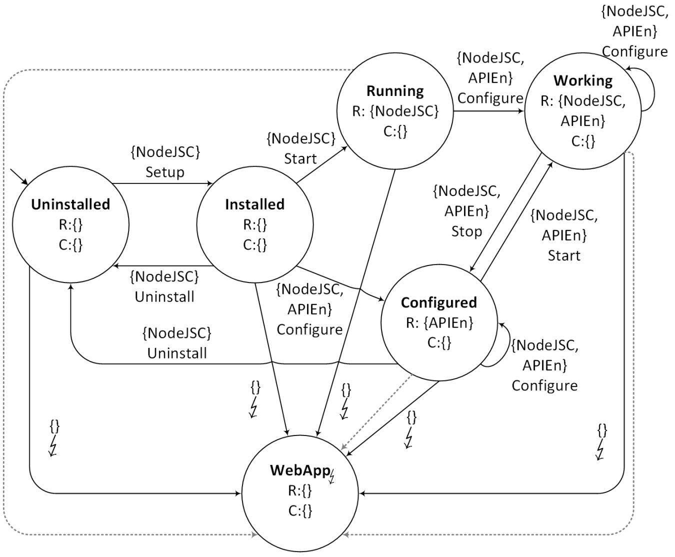


Figure 22: Fault-aware management protocol for nodes of type *WebApp* (such as *ThoughtsGui* in Fig. 18).

- *ThoughtsGui* is a *WebApp* node, which offers the operations to manage its lifecycle[14]. To effectively run, *ThoughtsGui* also require a container to be installed into, and the endpoint of the instance of *ThoughtsApi* to connect to (through its requirements *NodeJSContainer* and *APIEndpoint*, respectively).

  The fault-aware management protocol of *ThoughtsGui* is illustrated in Fig. 22. Its states are *Uninstalled* (initial), *Installed*, *Configured*, *Running*, *Working*, and $WebApp_{↯}$. *Uninstalled* is the only state that is not associated with any requirement or capability, while all other states specify that the capabilities corresponding to the indicated requirements must continue to be provided in order for *ThoughtsGui* to continue to work properly. All transitions bind their executability to the availability of the capability satisfying the requirement *NodeJSContainer*. The transitions targeting the states *Configured* and *Working* also require the requirement *APIEndpoint* to be actually satisfied. Finally, it is worth noting that all faults are default handled by a fault-handling transition targeting the "sink" state $WebApp_{↯}$ (Def. 5).

To permit analysing the management behaviour of *Thinking*, we modelled its application topology in TOSCA with Winery [9]. We then exploited Barrel to edit the fault-aware management protocols of the nodes in the application topology of *Thinking*[15] (by leveraging of the functionalities of Barrel to default handle faults, and to model the unexpected behaviour of components).

***Analysing the deployment of* Thinking.** Our first aim was to develop a management plan for deploying an instance of *Thinking*. By exploiting the documentation of *Thinking* available online as a reference, we came up with the plan depicted in Fig. 23.

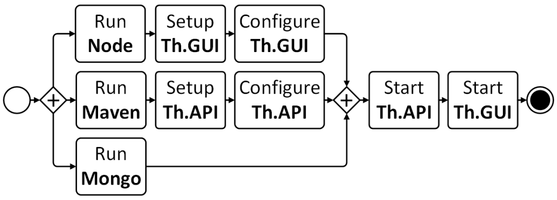


Figure 23: A deployment plan for *Thinking*.

We employed Barrel to check whether the plan we developed was valid, by simulating all its possible sequential traces with the *Simulator* available in Barrel. We actually discovered that none of them is valid, because each sequential trace tries to execute a management operation in a global state where the requirements needed to execute such operation are not satisfied. Each sequential trace indeed either tries to *Configure ThoughtsApi* to connect to the endpoint offered by *Mongo* before *Mongo* is actually offering its endpoint (viz., before *Mongo* is *Running*), or it tries to *Configure ThoughtsGui* to connect to the endpoint of *ThoughtsApi* before such endpoint is actually available (viz., before *ThoughtsApi* is *Working*).

We hence refined our deployment plan to avoid both aforementioned issues. Namely, to ensure that the *Configure* operation of *ThoughtsApi* is invoked when *Mongo* is *Running*, we moved it right after the initial parallel flow. To ensure that the the *Configure* operation of *ThoughtsGui* is invoked when *ThoughtsApi* is *Working*, we moved it right after the *Start* of *ThoughtsApi*. The resulting plan is shown in Fig. 24.

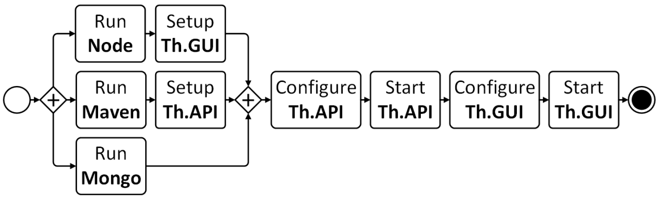


Figure 24: A valid deployment plan for *Thinking*.

We then simulated the sequential traces of the refined plan with Barrel, which all turned out to be valid. This in turn means that the deployment plan in Fig. 24 is valid. The latter effectively models reality, as by manually executing each of its sequential traces (viz., by concretely running the Docker commands and bash scripts implementing the management operations forming such traces), we always ended up with a "fresh" instance of the *Thinking*

[14] The bash scripts implementing the operations to *Setup*, *Run*, *Configure*, *Stop*, and *Uninstall* of *ThoughtsGui* are publicly available on GitHub at `https://github.com/di-unipi-socc/thinking/tree/master/gui/scripts`.

[15] The resulting CSAR is publicly available on GitHub at `https://github.com/di-unipi-socc/barrel/blob/master/examples`. It can be used to repeat the analyses discussed in the following sections.

application[16]

***Planning the undeployment of* Thinking.** We then considered the problem of undeploying a running instance of *Thinking*. Namely, we focused on the development of a management plan that, if executed when all components of *Thinking* are up and running, allows to go back to the initial global state where all components of *Thinking* are not installed.

We employed the *Planner* available in Barrel to automatically determine the desired undeployment plan. We first set the starting global state as that where *Node*, *Maven* and *Mongo* are *Running*, and where *ThoughtsApi* and *ThoughtsGui* are *Working*. As the target global state is by default set to that where all components are not installed, the *Planner* immediately displayed a valid sequence of operations allowing to move from the starting global state to the target global state. Such sequence is displayed in Fig. 25.

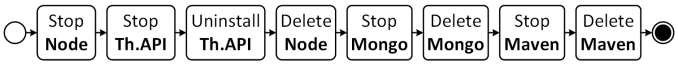


Figure 25: A management plan that permits undeploying a running instance of the *Thinking* application. This plan was automatically determined by the *Planner* of Barrel.

To check whether such plan was effectively undeploying *Thinking*, we launched an instance of *Thinking* by executing the deployment plan in Fig. 24. We then manually executed the sequence of operations displayed in Fig. 25, and this effectively resulted in going back to the initial situation where the containers *Node*, *Maven*, and *Mongo* were not present on the host[17].

***Analysing the effects of a misbehaving component.*** We also analysed the worst-possible effects of an unexpected crash of the *ThoughtsApi* component on a running instance of *Thinking*.

We first set the *Simulator* of Barrel to start from the global state where *Node*, *Maven* and *Mongo* are *Running*, and where *ThoughtsApi* and *ThoughtsGui* are *Working*. We then injected the crash of *ThoughtsApi* by simulating the execution of its *Crash* operation, which resulted in making *ThoughtsApi* become *Crashed*[18]. This also produced a fault in *ThoughtsGui*, as the latter was assuming the *APIEndpoint* offered by *ThoughtsApi* to continue to be provided. We also simulated the handling of such failure, which brought the application to the global state where *Node*, *Maven*, and *Mongo* are *Running*, where *ThoughtsApi* is *Crashed*, and where *ThoughtsGui* is *Running* (but not connected to any instance of *ThoughtsApi*).

Notice that the reached global state effectively models reality. To check this, we indeed started an instance of *Thinking* (by executing the deployment plan in Fig. 24) and we forced *ThoughtsApi* to "crash" (by actually killing the corresponding process in the *Maven* container). After such crash, both *ThoughtsGui* and *Mongo* were still running, but *ThoughtsGui* was not capable to effectively serve its clients because it could not connect to *ThoughtsApi*. Indeed, whenever we tried to connect to *ThoughtsGui*, no thought was displayed in it (since the GET request sent to *ThoughtsApi* did not receive any answer).

***Planning the (hard) recovery of* Thinking.** We finally considered the problem of developing a recovery plan for an instance of *Thinking* whose *ThoughtsApi* component unexpectedly crashed. After enabling hard recovery in Barrel, we exploited its *Planner* to automatically determine such recovery plan. We first set the starting global state as that where *Node*, *Maven* and *Mongo* are *Running*, where *ThoughtsApi* is *Crashed* and where *ThoughtsGui* is *Running*. We then set the target global state as that where *Node*, *Maven* and *Mongo* are *Running*, and where *ThoughtsApi* and *ThoughtsGui* are *Working*. The *Planner* immediately displayed a valid sequence of management operations allowing to reach the target global state from the starting one. Such a sequence of operations is depicted in Fig. 26.

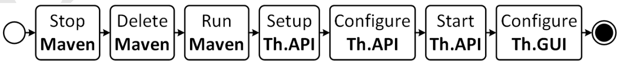


Figure 26: A (hard) recovery plan for a running instance of the *Thinking* application, whose *ThoughtsApi* component unexpectedly crashed. This plan was automatically determined by the *Planner* of Barrel.

We then went back to our crashed instance of *Thinking* (obtained as discussed above), and we manually executed the sequence of management operations in Fig. 26. This resulted in effectively recovering such an instance of *Thinking*, whose *ThoughtsGui* was again connected to an instance of *ThoughtsApi* (hence allowing to visualise stored thoughts and to insert new thoughts).

### *6.2. Controlled experiment*

In this section we report on a small controlled experiment whose aim was to provide a quantitative evalaution of how much our approach can be of help in analysing and planning the management of a composite application.

The experiment was based on four main tasks[19], each requiring to analyse/plan a different phase of the management of the *Thinking* application (Sect. 6.1):

- Task $T_1$ required to check the validity of a set of existing deployment plans. It was composed by four

[16].

[17] This was actually confirmed by their absence from the output of the command line instruction `docker ps -a` (which lists all Docker containers available on a host).

[18] Barrel denotes the ↯ operation and the sink state of a node with the names *Crash* and *Crashed*, respectively.

[19] A copy of the test (and of its solutions) is publicly available at `https://github.com/di-unipi-socc/barrel/blob/master/test`.

sub-tasks ($T_{1.1}$, $T_{1.2}$, $T_{1.3}$ and $T_{1.4}$), each concerning a different sequence of management operations for deploying instances of *Thinking*, and each requiring to be validated.

- Task $T_2$ focused on analysing the effects of a set of existing plans for managing a running instance of *Thinking*. It was composed by three sub-tasks ($T_{2.1}$, $T_{2.2}$ and $T_{2.3}$), each concerning a different management plan.
- Task $T_3$ concerned a running instance of *Thinking* and it was composed by the following two sub-tasks. $T_{3.1}$ required to understand how the state of a running instance of *Thinking* changes if we stop its component *Mongo*. $T_{3.2}$ required to indicate whether/how all components of an instance of *Thinking* can return to be up and running from the state reached in $T_{3.1}$.
- Task $T_4$ required to indicate whether/how the deployment of an instance of *Thinking* can be completed even if the *ThoughtsAPI* unexpectedly fails and becomes unresponsive to any of its management operations.

We submitted the above tasks to 24 volunteers, who were partitioned in two groups. The first group was composed by 2 post-doc researchers, 4 participants holding a MSc in computer science and 6 participants holding BSc in computer science. The second group was composed by 1 post-doc researcher, 4 participants holding a MSc in computer science and 7 participants holding BSc in computer science. Both groups were provided with the documentation and source code of *Thinking* available online. The first group was also provided with a TOSCA representation of *Thinking* (including the fault-aware management protocols of its components), which they were allowed to use in Barrel to analyse and plan the management of the *Thinking* application.

The goal of the experiment was to compare the time needed and the success rate of both groups to solve the above listed management tasks.

- The average time needed by the group exploiting Barrel to complete all tasks was 25 minutes (with a standard deviation of 6 minutes), while for the group not using Barrel it was 32 minutes (with a standard deviation of 8 minutes). We can hence observe that the group using Barrel was faster in completing the analysis and planning of the management of the *Thinking* application (25 minutes vs. 32 minutes), and that the time performance achieved by the single participants in the first group were also more uniform (due to a lower standard deviation).
- The success rate of both groups is displayed in Fig. 27. The average success rate of the group using Barrel was 89.2% (with a standard deviation of 6.7%), against the 30.8% (with a standard deviation of 13.8%) achieved by the other group. It is also worth noting that the highest differences in success rates have been registered in the tasks requiring to understand whether a plan was valid or not (viz., tasks $T_{1.1}$, $T_{1.2}$, $T_{1.3}$, $T_{1.4}$ and $T_{2.2}$), as well as in those requiring to understand the effects of failures and to recover from them (viz., tasks $T_{3.1}$, $T_{3.2}$ and $T_4$).

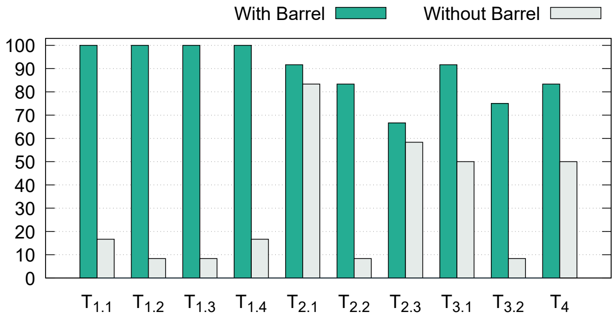


Figure 27: Histogram chart displaying the results of our controlled experiment. For each task (on the $x$-axis), two histograms display the average *success rate* (on the $y$-axis) obtained by both groups involved in our experiment (viz., the group exploiting Barrel as an additional design-time support, and the group not using it).

The small experiment (24 participants at work, on a 5-component application) that we conducted clearly indicates that the group using Barrel completed the tasks in less time and with a considerably higher success rate then the other group. This suggests that our approach can concretely help in designing the management of composite applications, especially if we think about enterprise applications, which typically integrate dozens (or even hundreds) of application components [3].

## 7. Related work

How to automate the management of applications is a well-known problem in computer science. In the cloud era, it has become even more prominent because of the complexity of both applications and platforms [2]. This is witnessed by the proliferation of so-called "configuration management systems", like Chef (`https://www.chef.io`) or Puppet (`https://puppet.com`). These management systems provide domain-specific languages to model the desired configuration for a software solution, and employ a client-server model to ensure that such configuration is met. However, the lack of a machine-readable representation of how to effectively manage cloud application components inhibits the possibility of performing automated analyses on components configurations and dependencies.

A first attempt to model the deployment of cloud-based applications was the Aeolus component model [15]. The Aeolus model shares our baseline idea of describing management behaviour of application components through finite-state machines, whose states are associated with conditions describing what is offered and required in a state.

Such descriptions can then be composed to model the management behaviour of a composite application. Aeolus however differs from our approach since it focuses on automating the deployment and configuration of composite cloud applications. Fault-aware management protocols instead focus on allowing to model and analyse the whole management of an application, including the recovery a desired application configuration after one or more of its components were faulted.

Another approach worth mentioning is Engage [16]. Given a partial installation specification, Engage is capable of automatically determining a full installation plan, which permits coordinating the deployment and configuration of an application across multiple machines. However, similarly to Aeolus [15], Engage differs from our approach since it focuses on the deployment of applications. We instead focus on the whole application management, by also allowing to explicitly model faults, to inject failures in application components, analysing their effects, and reacting to them to restore a desired application state.

The problem of rigorously engineering fault-tolerance in complex systems has been widely studied in computer science [17], and there exist various approaches allowing to design and analyse such a kind of systems. Two concrete examples are the approaches proposed by Qiang et al. [18] and Betin Can et al. [19], which provide facilities for fault-localisation in complex applications, hence allowing to re-design such applications by avoiding the occurrence of the identified faults. Another example is given by Johnsen et al. [20], which propose a solution for designing object-oriented applications by first considering fault-free applications, and by then iteratively refining their design by identifying and handling the faults that may occur. The main difference between the aforementioned approaches and ours is that objective of those approaches is to obtain applications that "never fail", because their potential faults have already been identified and properly handled. Our approach is instead more recovery-oriented [21], as we consider applications where faults can possibly occur, and we permit designing applications that can be recovered.

A similar argument applies to the approaches proposed by Grunske et al. [22], Kaiser et al. [23], and Alhosban et al. [24]. These approaches can however be considered somehow closer to ours. They indeed share with our approach the basic idea of first indicating which faults can affect components and how the latter react to them, and then composing the obtained models according to the dependencies occurring among the components of an application (viz., according to its topology).

Friedrich et al. [25] proposes an approach to handle faults in service-based processes which is very close in the spirit to ours. As we do for composite applications, service-based processes are described with a model-based approach, and the description of a process includes the possible repair actions for each of its activities. This permits checking recoverability of actions at design time, and generating recovery plans whenever a fault is detected to have happened (by an external monitoring tool). Friedrich et al. [25] however differ from our approach mainly because of the application domain, which permits them to exploit different techniques (such as heuristics based on branching probabilities) to carry out their analyses, and since they assume faults to happen one at a time. Friedrich et al. [25] differ from our approach also because they do not cope with services whose actual behaviour is different from that modelled in the process.

Boyer et al. [26] propose an approach to reconfigure component assemblies that is resilient to failures. Their approach is capable of automatically determining a reconfiguration plan that brings a component assembly from its actual configuration to a target configuration, by executing a sequence of reconfiguration operations. Their approach also deals with unexpected failures of components while the reconfiguration plan is being executed, by understanding which components are affected by a failure, and by re-computing a new reconfiguration plan still bringing the system to its target configuration (which, in the worst case, may require to roll-back a component assembly to its initial configuration, and to re-compute a different reconfiguration plan). The approach by Boyer et al. [26] however differs from our approach, as it assumes the set of reconfiguration operations and the behaviour of components to be fixed. We instead permit fully customising the set of operations to manage a component, as well as its (fault-aware) management behaviour.

Durán and Salaün [27] present a decentralised solution for the deployment and reconfiguration of composite cloud applications in presence of failures. In their solution, composite applications are modelled as sets of interconnected virtual machines. Each virtual machine is provided with a *configurator* module, which manages its lifecycle. A manager is then in charge of orchestrating deployment and reconfiguration of composite applications, by interacting with the configurators of the virtual machines of its components. The solution proposed by Durán and Salaün [27] is related to our approach as it aims at managing composite applications by taking into account failures and by specifying the management of each component separately. It however differs from our approach since, despite it allows to model the dependencies occurring among application components, it is not possible to distinguish between "vertical" dependencies (i.e., indicating that a component is hosted on another) and "horizontal" dependencies (i.e., indicating that a component just requires another to be up and running). Additionally, the solution proposed by Durán and Salaün [27] only considers environmental faults, while we also deal with application-specific faults. Similar considerations apply to the approach proposed by Etchevers et al. [28]

Liggesmeyer and Rothfelder [29] propose an approach for identifying faults in composite systems, by relying on a finite state machine-based representation of the behaviour of system components. Despite the analyses carried out by Liggesmeyer and Rothfelder are different from ours, they rely on a modelling quite close to ours. They indeed rely on a sort of requirements and capabilities to model the interaction among components, and they also allow to "implicitly" indicate how such components react to possible faults. Our modelling strictly generalises that proposed by Liggesmeyer and Rothfelder [29] because of the following reasons. Firstly, in our modelling the state of a component can be modified not only because of a requirement that is no more satisfied, but also because of the actual execution of a management operation. Secondly, our modelling allows to "explicitly" indicate how a node reacts to the occurrence of single/multiple faults (as fault handling transitions are distinct from those modelling the normal behaviour of a component). Similar considerations apply to the modelling proposed by Chen et al. [30], which is also based on finite state machines, and which also includes a sort of requirements and capabilities to model the interaction among components.

Hajisheykhi et al. [31] propose UFIT, a tool that permits analysing the fault-tolerance of systems. On the one hand, UFIT is similar to our approach as it allows to specify the behaviour of a system with a (timed) finite-state machine, which includes transitions explicitly modelling how a system reacts to the occurrence of faults. On the other hand, UFIT differs from our approach since it focuses on monolithic systems, hence not providing any natural way to combine the finite state machines modelling the behaviour of multiple interacting systems.

Two other approaches worth mentioning are those proposed by de Lemos and Fiadeiro [32], and by Nagatou and Watanabe [33]. The approach by de Lemos and Fiadeiro [32] indeed inspired the way in which we rely on the interdependencies among components to model fault-awareness, as well as the idea of recovering applications from faults by applying sequences of atomic operations (until we reach a desired application configuration). Nagatou and Watanabe [33] instead inspired our idea of injecting faults in applications to compute the effects of misbehaving components (hence allowing to recover applications from unpredictable faults).

To the best of our knowledge, ours is the first the approach that allows to automate the orchestration of the management of composite cloud applications, which considers that faults can (and probably will) occur during the management of such a kind of applications, hence allowing to explicitly indicate how each component of an application will react to their occurrence. It does so by allowing to customise the management behaviour of each component, and by automatically deriving the behaviour of a composite application by combining the behaviour of its components (according to the application topology).

Finally, it is worth noting that we investigated the possibility of employing composition-oriented automata (like *interface automata* [34]) to model the valid management of a composite application as the language accepted by the automaton obtained by composing the automata modelling the management protocols of its components. The main drawbacks of such an approach are the size of the obtained automaton (which in general grows exponentially with the number of application components), and the need of recomputing the automaton whenever the management protocol of a component is modified or whenever a new component is added to an application.

## 8. Conclusions

Automating the management of composite applications is currently one of the major concerns of enterprise IT [35]. Composite applications typically integrate various heterogeneous components, whose deployment, configuration, enactment, and termination must be suitably coordinated, by also taking into account all dependencies occurring among application components. As the amount of components grows, or the need to reconfigure them becomes more frequent, application management becomes more and more time-consuming and error-prone [1].

In this paper, we have proposed fault-aware management protocols, which permit compositionally modelling the management of composite cloud applications, by also taking into account the possibility of faults suddenly occurring, as well as of misbehaving components. We have shown that, given the specification of the topology of an application and of the fault-aware management protocols of its components, we can automatically derive the management behaviour of the whole application. We have also shown how such behaviour permits automating various useful analyses, like checking whether a (existing) workflow orchestrating the management of an application is valid, which are its effects, whether it generates faults, or determining plans for changing the actual configuration of an application (which is particularly useful to recover an application that is stuck because of a faulted/misbehaving node).

We also illustrated how we fruitfully integrated fault-aware management protocols in the OASIS standard TOSCA [7], by means of the Barrel prototype. Barrel can be exploited to validate and automate the (fault-aware) management of existing TOSCA applications, as we illustrated in Sect. 6. It is worth observing that, even if some of the analyses we presented in Sect. 4 have exponential time complexity in the worst case, they still constitute a significant improvement with respect to the state-of-the-art, as currently the management of the components forming a complex application is coordinated manually (e.g., by developing ad-hoc scripts), and it is hardly reusable since it is tightly coupled to such application.

The presented approach can be exploited for developing engines capable of automatically orchestrating the management of composite applications in a fault-resilient manner. Indeed, given a desired application configuration, an orchestrator can automatically execute the sequence of operations needed to reach such configuration, and it can maintain such configuration even if faults suddenly occur (or if components behave unexpectedly). The development of such orchestrator, as well as its exploitation to further validate our approach, is in the scope of our future work.

More generally, we plan to extend the analyses that can be performed on fault-aware management protocols. For instance, we plan to devise techniques permitting to improve our analyses by determining fragments of the topology of a composite application that can be managed independently from the rest of the topology.

Finally, it is worth noting that fault-aware management protocols currently do not take into account costs nor QoS, which are however important factors for cloud applications [36]. We plan to extend fault-aware management protocols to account also for costs and QoS, and to devise analysis techniques to determine the cost and QoS needed to manage a composite application (e.g., allowing to determine the "best" management/recovery plan that permit changing/restoring the configuration of an application).

## References

[1] T. Binz, U. Breitenbücher, O. Kopp, F. Leymann, TOSCA: Portable automated deployment and management of cloud applications, in: A. Bouguettaya, Z. Q. Sheng, F. Daniel (Eds.), Advanced Web Services, Springer, New York, NY, 2014, pp. 527–549. doi:10.1007/978-1-4614-7535-4_22.

[2] F. Leymann, Cloud computing, it - Information Technology 53 (4) (2011) 163–164. doi:10.1524/itit.2011.9070.

[3] T. Binz, C. Fehling, F. Leymann, A. Nowak, D. Schumm, Formalizing the Cloud through Enterprise Topology Graphs, in: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, IEEE, 2012, pp. 742–749. doi:10.1109/CLOUD.2012.143.

[4] A. Brogi, A. Canciani, J. Soldani, Modelling and analysing cloud application management, in: S. Dustdar, F. Leymann, M. Villari (Eds.), Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Taormina, Italy, September 15-17, 2015, Proceedings, Springer International Publishing, 2015, pp. 19–33. doi:10.1007/978-3-319-24072-5_2.

[5] R. I. Cook, How complex systems fail, Cognitive Technologies Laboratory, University of Chicago, URL: http://web.mit.edu/2.75/resources/random/HowComplex%20Systems%20Fail.pdf (1998).

[6] J. Gray, Why do computers stop and what can be done about it?, in: Proceedings of the 5th Symposium on Reliability in Distributed Software and Database Systems, Los Angeles, CA, USA, 1986, pp. 3–12.

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

[8] A. Brogi, A. Canciani, J. Soldani, Fault-aware application management protocols, in: M. Aiello, B. E. Johnsen, S. Dustdar, I. Georgievski (Eds.), Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings, Springer International Publishing, 2016, pp. 219–234. doi:10.1007/978-3-319-44482-6_14.

[9] O. Kopp, T. Binz, U. Breitenbücher, F. Leymann, Winery – a modeling tool for TOSCA-based cloud applications, in: S. Basu, C. Pautasso, L. Zhang, X. Fu (Eds.), Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings, Springer, Berlin, Heidelberg, 2013, pp. 700–704. doi:10.1007/978-3-642-45005-1_64.

[10] V. Andrikopoulos, A. Reuter, S. Gómez Sáez, F. Leymann, A GENTL approach for cloud application topologies, in: M. Villari, W. Zimmermann, K.-K. Lau (Eds.), Service-Oriented and Cloud Computing: 3rd European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings, Springer Berlin Heidelberg, 2014, pp. 148–159. doi:10.1007/978-3-662-44879-3_11.

[11] A. Brogi, J. Soldani, P. Wang, TOSCA in a nutshell: Promises and perspectives, in: M. Villari, W. Zimmermann, K.-K. Lau (Eds.), Service-Oriented and Cloud Computing: Third European Conference, ESOCC 2014, Manchester, UK, September 2-4, 2014. Proceedings, Springer Berlin Heidelberg, Berlin, Heidelberg, 2014, pp. 171–186. doi:10.1007/978-3-662-44879-3_13.

[12] R. W. Floyd, Algorithm 97: Shortest path, Communications of the ACM 5 (6) (1962) 345.

[13] T. Binz, U. Breitenbücher, F. Haupt, O. Kopp, F. Leymann, A. Nowak, S. Wagner, OpenTOSCA – a runtime for TOSCA-based cloud applications, in: S. Basu, C. Pautasso, L. Zhang, X. Fu (Eds.), Service-Oriented Computing: 11th International Conference, ICSOC 2013, Berlin, Germany, December 2-5, 2013, Proceedings, Springer, Berlin, Heidelberg, 2013, pp. 692–695. doi:10.1007/978-3-642-45005-1_62.

[14] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, Vinothek - A Self-Service Portal for TOSCA, in: Proceedings of the 6th Central European Workshop on Services and their Composition (ZEUS'14), Vol. 1140 of CEUR Workshop Proceedings, CEUR-WS.org, 2014, pp. 69–72.

[15] R. Di Cosmo, J. Mauro, S. Zacchiroli, G. Zavattaro, Aeolus: A component model for the cloud, Information and Computation 239 (0) (2014) 100 – 121. doi:10.1016/j.ic.2014.11.002.

[16] J. Fischer, R. Majumdar, S. Esmaeilsabzali, Engage: A deployment management system, in: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, ACM, 2012, pp. 263–274. doi:10.1145/2254064.2254096.

[17] M. Butler, C. Jones, A. Romanovsky, E. Troubitsyna, Rigorous Development of Complex Fault-Tolerant Systems, Vol. 4157 of Lecture Notes in Computer Science, Springer-Verlag Berlin Heidelberg, 2006. doi:10.1007/11916246.

[18] W. Qiang, L. Yan, S. Bliudze, M. Xiaoguang, Automatic fault localization for BIP, in: X. Li, Z. Liu, W. Yi (Eds.), Dependable Software Engineering: Theories, Tools, and Applications, First International Symposium, Nanjing, China, November 4-6, 2015, Proceedings, SETTA, Springer, 2015, pp. 277–283. doi:10.1007/978-3-319-25942-0_18.

[19] A. Betin Can, T. Bultan, M. Lindvall, B. Lux, S. Topp, Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers, Automated Software Engineering 14 (2) (2007) 129–178. doi:10.1007/s10515-007-0008-2.

[20] E. Johnsen, O. Owe, E. Munthe-Kaas, J. Vain, Incremental fault-tolerant design in an object-oriented setting, in: Proceedings of the Second Asia-Pacific Conference on Quality Software, APAQS, IEEE Computer Society, 2001, pp. 223–.

[21] G. Candea, A. B. Brown, A. Fox, D. Patterson, Recovery-oriented computing: Building multitier dependability, Computer 37 (11) (2004) 60–67. doi:10.1109/MC.2004.219.

[22] L. Grunske, B. Kaiser, Y. Papadopoulos, Model-driven safety evaluation with state-event-based component failure annotations, in: Proceedings of the 8th International Conference on Component-Based Software Engineering, CBSE, Springer-Verlag, 2005, pp. 33–48. doi:10.1007/11424529_3.

[23] B. Kaiser, P. Liggesmeyer, O. Mäckel, A new component concept for fault trees, in: Proceedings of the 8th Australian Workshop on Safety Critical Systems and Software - Volume 33, SCS, Australian Computer Society, Inc., 2003, pp. 37–46.

[24] A. Alhosban, K. Hashmi, Z. Malik, B. Medjahed, S. Benbernou, Bottom-up fault management in service-based systems, ACM Trans. Internet Technol. 15 (2) (2015) 7:1–7:40. doi:10.1145/2739045.

[25] G. Friedrich, M. G. Fugini, E. Mussi, B. Pernici, G. Tagni,

Exception handling for repair in service-based processes, IEEE Transactions on Software Engineering 36 (2) (2010) 198–215. doi:10.1109/TSE.2010.8.

[26] F. Boyer, O. Gruber, D. Pous, Robust reconfigurations of component assemblies, in: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, IEEE Press, Piscataway, NJ, USA, 2013, pp. 13–22.

[27] F. Durán, G. Salaün, Robust and reliable reconfiguration of cloud applications, J. Syst. Softw. 122 (C) (2016) 524–537. doi:10.1016/j.jss.2015.09.020.

[28] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, N. DePalma, Reliable self-deployment of distributed cloud applications, Softw. Pract. Exper. 47 (1) (2017) 3–20. doi:10.1002/spe.2400.

[29] P. Liggesmeyer, M. Rothfelder, Improving system reliability with automatic fault tree generation, in: Proceedings of the 28th International Symposium on Fault-Tolerant Computing, FTCS, IEEE Computer Society, 1998, pp. 90–99. doi:10.1109/FTCS.1998.689458.

[30] L. Chen, J. Jiao, J. Fan, Fault propagation formal modeling based on stateflow, in: Reliability Systems Engineering, First International Conference on, ICRSE, IEEE, 2015, pp. 1–7. doi:10.1109/ICRSE.2015.7366480.

[31] R. Hajisheykhi, A. Ebnenasir, S. S. Kulkarni, UFIT: A tool for modeling faults in UPPAAL timed automata, in: K. Havelund, G. Holzmann, R. Joshi (Eds.), NASA Formal Methods, 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings, Springer, 2015, pp. 429–435. doi:10.1007/978-3-319-17524-9_32.

[32] R. de Lemos, J. L. Fiadeiro, An architectural support for self-adaptive software for treating faults, in: Proceedings of the First Workshop on Self-healing Systems, WOSS, ACM, 2002, pp. 39–42. doi:10.1145/582128.582136.

[33] N. Nagatou, T. Watanabe, A model-checking based approach to robustness analysis of procedures under human-made faults, in: C. Ouyang, J.-Y. Jung (Eds.), Asia Pacific Business Process Management: Second Asia Pacific Conference, AP-BPM 2014, Brisbane, QLD, Australia, July 3-4, 2014. Proceedings, Springer International Publishing, 2014, pp. 117–131. doi:10.1007/978-3-319-08222-6_9.

[34] L. de Alfaro, T. A. Henzinger, Interface automata, in: Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-9, ACM, 2001, pp. 109–120. doi:10.1145/503209.503226.

[35] C. Pahl, A. Brogi, J. Soldani, P. Jamshidi, Cloud Container Technologies: a State-of-the-Art Review, IEEE Transactions on Cloud Computing*[In press]*. doi:10.1109/TCC.2017.2702586.

[36] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, M. Zaharia, A view of cloud computing, Commun. ACM 53 (4) (2010) 50–58. doi:10.1145/1721654.1721672.

## About the authors

**Antonio Brogi** is a full professor at the Department of Computer Science, University of Pisa (Italy) since 2004. He holds a Ph.D. in Computer Science (1993) from the University of Pisa. His research interests include service-oriented and cloud-based computing, coordination and adaptation of software elements, formal methods, and design of programming languages. He has published the results of his research in over 150 papers in international journals and conferences.

**Andrea Canciani** is a post-doc researcher at the University of Pisa (Italy). He holds a PhD in Computer Science (2017, University of Pisa). His research interests span over multiple areas, from programming languages to cloud and fog computing, passing through concurrency theory and event-based software engineering. He has been actively participating in several research projects both at national and international level.

**Jacopo Soldani** is a post-doc researcher at the University of Pisa (Italy). He holds a PhD in Computer Science (2017, University of Pisa). His research interests include, but are not limited to, service-oriented and cloud computing, adaptation, coordination, and integration of software elements, software engineering and formal methods. He has been involved in various research projects on cloud and fog computing both at local and EU level.