Modelling and Analysing Replica- and Fault-aware Management of Horizontally Scalable Applications

JACOPO SOLDANI, Department of Computer Science, University of Pisa, Italy
MARCO CAMERIERO, Department of Computer Science, University of Pisa, Italy
GIULIO PAPARELLI, Department of Computer Science, University of Pisa, Italy
ANTONIO BROGI, Department of Computer Science, University of Pisa, Italy

Modern enterprise applications integrate multiple interdependent software components, whose management must be suitably coordinated. This must be done by taking into account all inter-component dependencies, the faults potentially affecting them, and the fact that each component can be horizontally scaled, i.e., that multiple instances of each component can be spawned or destroyed depending on application needs. In this paper, we introduce a novel solution for suitably modelling and analysing the replica- and fault-aware management of multi-component applications, based on topology graphs and management protocols. More precisely, we first introduce a compositional model of the management behaviour of the (possibly multiple) instances of the components forming an application, faults included. We then show how this model enables automating various useful analyses, from checking the validity of management plans, to automatically determining management plans allowing the instance of an application to reach and maintain a desired target configuration.

CCS Concepts: • **Applied computing** → *Service-oriented architectures*; *Service-oriented architectures*; Enterprise applications; • **Software and its engineering** → *Orchestration languages*; Formal methods; *Orchestration languages*; Software fault tolerance; Formal methods.

1 INTRODUCTION

Automating application management is one of the main challenges in enterprise IT nowadays [16]. The efficient exploitation of cloud computing peculiarities indeed highly depends on the degree of management automation of deployed applications [18]. At the same time, since modern enterprise applications typically integrate multiple heterogeneous components [24], automating their management requires suitably coordinating the concurrent deployment, configuration, enactment, and termination of the (possibly multiple) instances of their components. Even if this may be done by different independent teams (e.g., DevOps squads), it must be done by considering all dependencies occurring among all the instances of the components forming an application. As the number of instantiated components grows, and the need to reconfigure them becomes more frequent, application management becomes more complex, time-consuming, and error-prone [3].

Multi-component applications can be conveniently specified by means of topology graphs [4] and management protocols [6, 7]. Topology graphs [4] enable specifying the topology (i.e., the structure) of an application as a directed graph, whose nodes represent application components and whose oriented arcs model inter-component dependencies. More precisely, each node in a topology graph represents an application component by also

Authors' addresses: Jacopo Soldani, Department of Computer Science, University of Pisa, Pisa, Italy, jacopo.soldani@unipi.it; Marco Cameriero, Department of Computer Science, University of Pisa, Pisa, Italy; Giulio Paparelli, Department of Computer Science, University of Pisa, Pisa, Italy; Antonio Brogi, Department of Computer Science, University of Pisa, Pisa, Italy.

1533-5399/2022/1-ART $15.00
https://doi.org/10.1145/3511302

specifying the operations to manage its lifecycle, the requirements it needs to work, and the capabilities it offers to satisfy the requirements of other components. Each oriented arc instead represents the dependency of a component on another by connecting a requirement of the former to a capability of the latter. On the other hand, management protocols enable modelling the management behaviour of the nodes in an application topology [6, 7]. A management protocol is a finite state machine whose states correspond to the possible states of the node, and whose transitions indicate whether a management operation can be performed in a state and which is the state reached by performing it. Transitions and states are also associated with conditions indicating which requirements must be satisfied for a node to reside in a state or to successfully complete a management operation, as well as which capabilities it actually provides while residing in a state or performing a transition. Management protocols also indicate how a node reacts to faults, which occur whenever a needed requirement stops being satisfied.

Given the possibility of statically specifying multi-component applications as described above, one can exploit it to model and analyse the management of actual instances of such an application, therein included all the node instances used to replicate its components, and their possible faults. This is however not possible with currently existing modelling and analysis approaches based on topology graphs and management protocols [6, 7]. These are inherently designed to deal with the static structure given by a topology graph, which means that they can work with *only one* instance of each node. Such a restriction is quite limiting, as it does not allow dealing with the horizontal scaling of applications, i.e., the spawning of multiple replicated instances of their components.

The horizontal scaling of application components is however crucial in application management nowadays [12, 23], as it allows to spawn/destroy replicas of an application component based on the application's needs [18]. An incorrect management of application replicas would result in not suitably exploiting horizontal scaling, which could severely affect the overall application management. A simple example of why properly dealing with replicas is important is the following. Suppose that two instances f1 and f2 of a frontend component are relying on an instance b1 of another backend component to deliver their functionalities, and suppose that b1 suddenly fails. The fault of b1 does not necessarily lead f1 and f2 to stop delivering their functionalities if they can switch to using another instance of the backend component. On the other hand, if no other instance of the backend component is available, or if f1 and f2 need precisely the availability of instance b1 (e.g., because they set up a persistent, encrypted connection to b1), the fault of b1 will lead f1 and f2 to stop delivering their functionalities (and, in the worst case, even to possibly fail).

In this paper, we overcome the limitations of currently existing approaches based on management protocols by extending them to enable modelling and analysing the management of horizontally scalable applications. The key ingredient for this is releasing the assumption of having a "static" topology describing the application *both* at design-time and at run-time. More precisely, we clearly distinguish the static specification of applications, given in terms of topology graphs and management protocols, from the modelling of application instances. We present a new compositional approach to dynamically and adaptively modelling and analysing runtime application instances, geared towards enabling to horizontally scale their components, while at the same time considering the possibility of component instances to fail. The main contributions of this paper are threefold:

(i) We introduce a compositional modelling that enables deriving the allowed management behaviour for instances of multi-component applications. The modelling distinguishes among the different instances of the nodes in the application specification, which can be spawned or destroyed to increase or decrease the replicas of a given application component, hence setting the foundations for modelling the management of horizontally scalable applications.
(ii) We illustrate how our modelling can be used to automate various useful analyses on the management of horizontally scalable applications, e.g., checking the validity of existing plans or whether they bring an

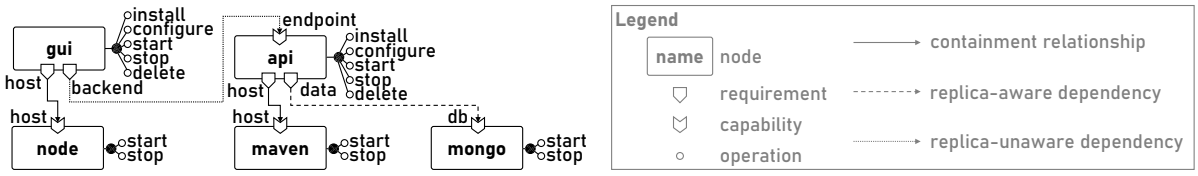


Fig. 1. Topology of the *Thinking* application.

application to a desired configuration (e.g., instantiating a given amount of replicas for each component, and bringing such replicas to given states).

(iii) We show how our modelling also enables automatically planning the fault-aware management of horizontally scalable applications, i.e., automatically deriving management plans allowing an application to reach and/or maintain a target configuration, even in presence of faults. We also formally prove that, even if applications can have infinitely many possible configurations, it is possible to finitely determine whether and how a target configuration can be reached.

The rest of this paper is organised as follows. Section 2 provides an example motivating the need for dealing with replicas when modelling and analysing multi-component application management. Section 3 illustrates how to compositionally specify the structure and management behaviour of multi-component applications, and how to model their runtime instances. Section 4 introduces the analyses that can be performed with the presented modelling, from validating already planned management to automatically planning it. Section 6 and Section 7 discuss related work and draw some concluding remarks, respectively.

This article extends our previous work [7]. The notion of management protocols (Section 3.1) is taken from [7], while the way of specifying multi-component applications (Section 3.2) suitably extends the corresponding specification from [7]. The modelling of the management of horizontally scalable applications (Section 3.3) and the analysis and planning of the management of horizontally scalable applications (Section 4) are instead entirely new and presented here for the first time.

2 MOTIVATING SCENARIO

Consider *Thinking* (https://github.com/di-unipi-socc/thinking), an open-source application specifically designed for showcasing solutions for orchestrating multi-component applications. *Thinking* is a web-based application allowing its users to share thoughts on a web portal, so that other users can read them. It is composed by three main components (Figure 1): (i) a database storing the collection of thoughts shared by users, which is obtained by directly instantiating a mongo Docker container, (ii) a Java-based RESTful api to remotely access the database of shared thoughts, and (iii) a web-based gui visualising all shared thoughts and allowing to insert new thoughts into the database. The api is hosted on a maven Docker container, and it requires to be directly connected to the mongo container (for remotely accessing the database of shared thoughts). The gui is instead hosted on a node container, and it depends on the availability of an instance of the api to properly work (as it sends HTTP requests to the api to retrieve/add shared thoughts).

Figure 1 illustrates the application topology of *Thinking*, depicted according to the OASIS TOSCA graphical notation [17]. Topology nodes represent the components of the *Thinking* application, with each node indicating the requirements of the corresponding component, the capabilities it offers to satisfy the requirements of other components, and the operations allowing to manage its lifecycle. Inter-component relationships are represented as directed arcs connecting a requirement of a node to the capability of another node that is used to satisfy such requirement. Relationships can be of three types, i.e., *containment* relationships, and *replica-aware* and *replica-unaware* dependencies, to distinguish three possible inter-component dependencies. Containment relationships

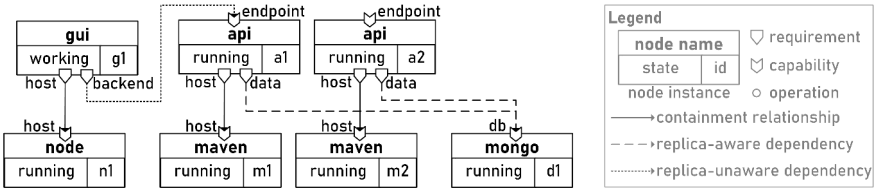


Fig. 2. An example of global state for the *Thinking* application.

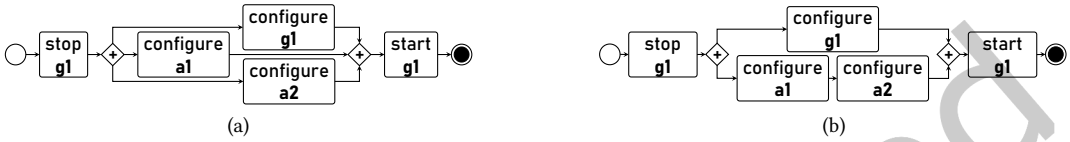


Fig. 3. Management plans for reconfiguring the instances of gui and api.

model the fact that a component is contained in another, as in the case of the gui being hosted on node, for instance. Replica-aware dependencies indicate that each instance of the source node depends on a specific instance of the target node, e.g., each instance of api sets a persistent connection to the instance of mongo it connects to, hence depending on such specific instance. Finally, replica-unaware dependencies indicate that each instance of the source node gets the corresponding requirement satisfied as long as there is an instance of the target node providing the corresponding capability, regardless of which specific replica is used to actually satisfy it, e.g., whenever an instance of gui sends a request to the backend api, any instance of the latter can be used to satisfy such request.

Consider now the situation shown in Figure 2, in which there are instances for all nodes of *Thinking*, each in a given *state* and with an *id* uniquely identifying it. According to the figure, there is one instance of the nodes gui, node, and mongo, and there are two different instances of api (a1 and a2) hosted by two different instances of maven (m1 and m2, respectively). The figure also shows the runtime bindings among node instances, with g1 currently exploiting a1 as backend, and with both instances of api being connected to the instance d1 of mongo. Suppose also that we wish to reconfigure the instances of the nodes gui and api. Designing and developing management plans allowing to do it is however complex, error-prone, and time-consuming, as we must consider all the dependencies occurring among the instances of the application components [19]. All such dependencies should be satisfied while executing the plan, as violating some dependency may result in causing some faults in some components, or even worse in disallowing to execute some management operation. Checking the validity of existing plans is not easy as well [6]: it indeed requires checking all possible evolutions of an application based on all possible parallel executions of the operations, hence resulting in a problem whose complexity grows (in the worst case) exponentially with the numbers of involved component instances and management operations.

Suppose, for example, that we devised two different management plans for reconfiguring the instances of the nodes gui and api, i.e., the plans (a) and (b) in Figure 3. Both management plans seem to validly reconfigure the instances of gui and api, whereas only plan (b) is actually valid. Some concrete executions of plan (a) may indeed fail in reconfiguring the instance g1 of gui. The instance g1 can indeed successfully execute its configure operation only if an instance of api is actually providing its endpoint to satisfy the requirement backend of g1. In other words, it must be that either a1 continues to provide its endpoint capability to satisfy the requirement backend of g1, or that g1 dynamically switches to the endpoint offered by a2 to successfully complete its configure operation. However, both a1 and a2 are not providing any endpoint while they are being reconfigured, i.e., while executing their configure operation. This means that it is not possible to reconfigure all the instances of gui and

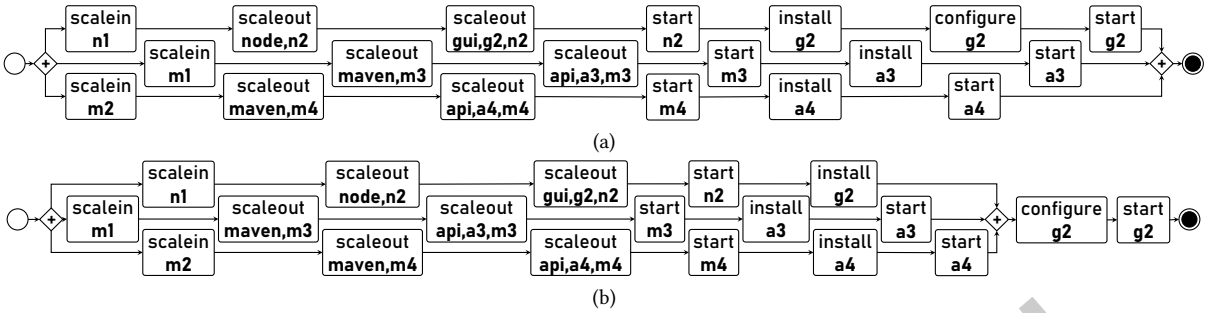


Fig. 4. Management plans for restarting the instances of node and maven.

api in parallel. On the other hand, before and after being reconfigured, both instances of api are actually listening on their endpoint, hence meaning that the parallel execution of operations shown in plan (b) can effectively work, as there is always an instance of api capable offering the endpoint needed by g1.

Another interesting scenario is the need to restart all running instances of node and maven from the overall application state shown in Figure 2, e.g., because of security updates to the corresponding Docker containers. Figure 4 presents two plans that seem to accomplish the desired management task, but only plan (b) validly does so. Plan (a) destroys all the instances of node and maven, it creates and starts three new instances replacing the destroyed ones, on which it creates and starts new instances of gui and api. Given that all three software stacks are created and started in parallel, the operation to start the newly created instance g2 of gui might be executed before any instance of api is up and running, which would result in a fault as the requirement backend of g2 should be satisfied for g2 to succesfully start. This hence means that some concrete executions of plan (a) in Figure 4 may not successfully complete. The same does not hold for plan (b), which validly executes the operation start on g2 only when the newly created instances of api are up and running.

Even if simple, the two above examples clearly show that it is crucial to take into account the concurrent management of multiple replicas while analysing the fault-aware management of multi-component applications. This not only holds for validating plans designed to accomplish given management tasks (such as in the example above), but also to automatically generate such plans, e.g., to automatically plan the deployment or reconfiguration of an application, as well as to automatically generate recovery plans enabling to restore a desired application configuration after some fault occurred. We hereafter present a formal framework that enables both validating the planned fault-aware management of horizontally scalable applications, and automatically determining the sequence of management operations allowing applications to reach and maintain a desired configuration (i.e., how many replicas of each component to instantiate, and which is their target state), even in presence of faults.

3 MODELLING THE MANAGEMENT OF HORIZONTALLY SCALABLE APPLICATIONS

Modelling the fault-aware management of horizontally scalable applications requires to take into account two distinct, but related, aspects, i.e., the *specification* of a multi-component application and its runtime *instances*. In this section, after recapping how management protocols allow modelling the fault-aware management behaviour of application components (Section 3.1), we illustrate how to compositionally *specify* the structure and management behaviour of a multi-component application (Section 3.2). We then exploit the formalised notion of application specification to model the replica- and fault-aware management of application *instances* (Section 3.3).

Whilst the recap of management protocols in Section 3.1 is retaken from our previous work [7], the notion of replica-aware application specification presented in Section 3.2 is obtained by suitably extending the corresponding

notions from our previous work [7]. The modelling of the replica- and fault-aware management of application instances presented in Section 3.3 is instead brand new.

3.1 Management Protocols

Multi-component applications are typically represented by indicating the states, requirements, capabilities and management operations of the nodes composing their topology [1]. Management protocols [7] enable specifying the management behaviour of a node N by indicating (i) whether/how each management operation of N depends on other management operations of N, (ii) whether/how it depends on operations of the nodes that provide capabilities satisfying the requirements of N, and (iii) how N reacts when a fault occurs. The dependencies (i) among the management operations of N are indicated by means of a transition relation τ_N, which relates such operations with its states. The transition relation indicates whether a management operation can be executed in a state of N, and which state is reached if its execution is successfully completed.

The description of (ii) whether and how the management of N depends on that of other nodes is instead given by associating (possibly empty) sets of requirements with both states and transitions. The requirements associated with a state or transition of N must continue to be satisfied in order for N to continue residing in such state or to successfully complete executing such transition. As a requirement is satisfied when the corresponding capability is provided, the requirements associated with states and transitions actually indicate which capabilities must continue to be provided in order for N to continue to work properly. The description of a node N is then completed by associating its states and transitions with (possibly empty) sets of capabilities that indicate the capabilities that are actually provided by N while residing in a state and while executing a transition.

Finally, (iii) faults can occur when N is assuming some requirements to be satisfied and some of the capabilities satisfying such requirements stop being provided by the corresponding nodes. To describe how N reacts to faults, a transition relation φ_N models the fault handling of N by indicating the state it reaches when a fault occurs while it is in a state or executing a transition.

Definition 3.1 (Management Protocols). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be the node modelling an application component, with S_N, R_N, C_N, and O_N being the finite sets of states, requirements, capabilities, and management operations of N, respectively. $\mathcal{M}_N = \langle \bar{s}_N, \tau_N, \rho_N, \chi_N, \varphi_N \rangle$ is a finite state machine defining the* management protocol *of N, where*[1]

- *$\bar{s}_N \in S_N$ is the initial state,*
- *$\tau_N \subseteq S_N \times O_N \times S_N$ models the transition relation,*
- *$\rho_N : (S_N \cup \tau_N) \rightarrow \mathcal{P}(R_N)$ indicates which requirements must hold in each state $s \in S_N$ and during each transition $\langle s, o, s' \rangle \in \tau_N$,*
- *$\chi_N : (S_N \cup \tau_N) \rightarrow \mathcal{P}(C_N)$ indicates which capabilities of N are offered in each state $s \in S_N$ and during each transition $\langle s, o, s' \rangle \in \tau_N$, and*
- *$\varphi_N \subseteq (S_N \cup \tau_N) \times S_N$ models the fault handling for a node.*

Figure 5 illustrates the management protocols of the nodes forming the *Thinking* application in our motivating scenario (Section 2). Consider, for instance, the management protocol (b), which describes the management behaviour of api. Its possible states are unavailable (initial), available, running, and damaged. No requirements and capabilities are associated with states unavailable, available, and damaged, which means that api does not require nor provide anything in such states. The same does not hold for the running state, where api concretely provides its endpoint capability, and where it assumes its requirements host and database to continue to be satisfied. If host or database are faulted while api is running, api goes back to its available state. Finally, the transitions of the management protocol indicate that api needs host to be satisfied while executing its operations,

[1] In the formulas, we follow the convention of using $\mathcal{P}(X)$ to denote the powerset of X.

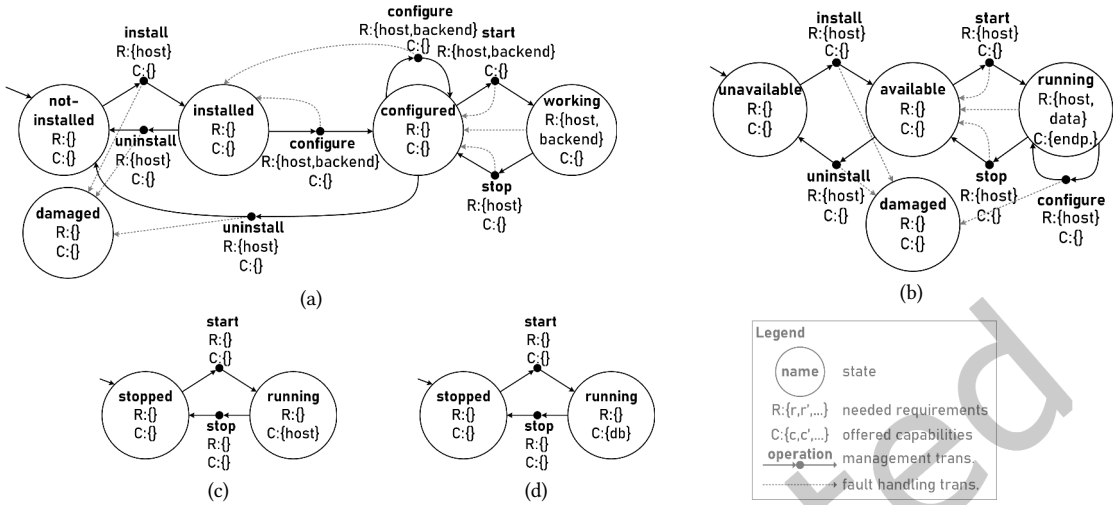


Fig. 5. Management protocols of the nodes (a) gui, (b) api, (c) node and maven, and (d) mongo in *Thinking*.

and that it does not feature any capability during their execution. If host is faulted while executing start or stop, then api gets back to its state available. If host is instead faulted while executing install, uninstall, or config, then api gets damaged.

To summarise, each component in an application can be modelled by a node describing its states, requirements, the capabilities it features (to satisfy the requirements of other components), and the operations for managing it. The management behaviour of the application component can then be modelled by associating the node with a management protocol describing such node [7].

3.2 Multi-Component Application Specification

Multi-component applications can be conveniently represented by means of topology graphs [4]. The nodes in a topology graph represent the components of an application (Section 3.1). Oriented arcs instead represent the dependencies among such components, by associating the requirements of a node with capabilities featured by other nodes. They hence define a *binding* function associating each requirement of each node to the capability satisfying it. For instance, in our motivating scenario (Section 2), the binding function associates the requirements host and database of api with the host capability featured by maven and the db capability featured by database, respectively.

The specification of the nodes forming an application and the binding function modelling their interconnections is enough to specify applications whose components are not going to be replicated, as in our previous work [7]. Our aim here is instead to enable dealing with replicas while modelling and analysing the management of multi-component applications, therein included the fact that —at runtime— we may have multiple replicated instances of each node in an application. Consider the instance of a node, and suppose that it starts assuming a requirement r (e.g., after successfully completing the execution of a management operation). The binding function associates r to a given capability of a given node, which may be in turn available in multiple instances, some of which are in a state where they concretely provide the needed capability. The actual choice of which instance of the target component to exploit to satisfy r strictly depends on the application behaviour, e.g., to the policy it implements to balance the load among the replicas of a node. As the binding function per se is not modelling such application-specific choices, the application specification is completed by a *connection policy function* specifically

doing it. Notably, the possible connection policy functions can be many (from random choices to more complex schemes, e.g., round robin), and different policies can be used for different requirements of different nodes of the same application. The connection policy function hence provides a "hook" that enables tailoring the specification to model the actual behaviour of an application. For instance, in our motivating example (Figure 1), to ensure that the same instance of the gui will display the same set of thoughts, the connection policy function will be such that an instance of gui will connect to one of the replicas of api managing the access to the instances of mongo managing the same data.

We hereafter formally define the notion of *application specification*.[2] We do it right after introducing some shorthand notation for denoting the identifiers that can be possibly assigned to instances of the nodes in an application, which is needed for actually typing the connection policy function.

Notation 3.1 (Identifiers). *We denote with $\mathbb{I}$ the universe of possible* identifiers *for the instances of the nodes forming an application.*

Definition 3.2 (Application Specification). *The* specification *of a multi-component application is a triple $A = \langle T, b, \pi \rangle$, where*

- *T is a finite set of nodes modelling the application components,*
- *$b : \bigcup_{N \in T} R_N \rightarrow \bigcup_{N \in T} C_N$ is the binding function, and*
- *$\pi : \bigcup_{N \in T} R_N \times \mathcal{P}(\mathbb{I}) \rightarrow \mathbb{I}$ is the connection policy function.*

In the specification of each node forming an application, we must distinguish among three different types of requirements, i.e., *containment*, *replica-aware*, and *replica-unaware* requirements. These indeed are sources of different types of relationships modelling different type of dependencies, which result in different types of potential faults.

Containment requirements are used to model so-called "vertical" dependencies (e.g., a node hosted on another node) meaning that the former is actually contained in the latter. Notably, when the instance of a node is destroyed, all the contained instances immediately disappear. An example for this is the abrupt shutdown of the Docker container n1 in Figure 1, which immediately results in abruptly shutting down also the software component running within such container, i.e., g1.

Replica-aware and replica-unaware requirements model "horizontal" dependencies, e.g., indicating that a node connects to another node, or that a node exploits some functionality provided by another node. Replica-aware and replica-unaware requirements differ one another based on whether the actual identity of the target node instance is important or not. When a replica-aware requirement of the instance of a node stops being satisfied, an explicit fault must be fired to let the node instance handle such fault. This is the case of the database requirement of api in our motivating scenario: as each instance of api sets a persistent connection to an instance of the mongo database, if the latter stops providing its endpoint, the connection gets broken. A fault has hence to be notified to the instance of api, to let it create a new connection to another instance of mongo, if available. Instead, with a replica-unaware requirement, the identity of the instance used to satisfy the requirement is not important, and we can avoid firing an explicit fault if another instance of the target node can provide the same capability. This is the case of the backend requirement of gui in our motivating scenario: whenever an instance of gui needs to communicate with the backend api, any instance of the latter can be used to satisfy the corresponding requests.

Notation 3.2 (Requirement Types). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node. We denote with $R_N^C \subseteq R_N$ the containment requirements of N, with $R_N^A \subseteq R_N$ its replica-aware requirements, and with $R_N^U \subseteq R_N$ its replica-unaware requirements.*

[2]For simplicity, and without loss of generality, we assume that the names of states, requirements, capabilities, and operations of the nodes in a topology are all disjoint.

The above given notions may allow introducing some inconsistencies while providing an application specification. To avoid such inconsistencies, we define below the notion of *well-formed application specification*, which prescribes that (i) each node can have at most one containment requirement, that (ii) the requirements of each node are partitioned among containment, replica-aware, and replica-unaware requirements, that (iii) management protocols enjoy some basic properties, that (iv) the application topology is acyclic, and that (v) the connection policy function π selects an identifier within the set of possible identifiers given it as input. Condition (iv) is to align with the existing notion of topology graphs [4], which should be directed and acyclic. In our case, this means that there is no chain of bindings between the requirements and capabilities of components such that the initial and final components in the chain are the same.

Definition 3.3 (Well-Formed Application Specification). *Let $A = \langle T, b, \pi \rangle$ be an application specification, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T. A is* well-formed *iff*[3]

(i) $\forall N \in T \,.\, |R_N^C| \leq 1$,
(ii) $\forall N \in T \,.\, R_N^C \uplus R_N^A \uplus R_N^U = R_N$,
(iii) $\forall N \in T \,.\, \mathcal{M}_N$ *is well-formed, deterministic, race-free, and complete,*[4]
(iv) $\nexists N_1, N_2, \ldots N_m \in T \,.\, (\exists r_1 \in R_{N_1} : b(r_1) \in C_{N_2}) \wedge (\exists r_2 \in R_{N_2} : b(r_2) \in C_{N_3}) \wedge \cdots \wedge (\exists r_{m-1} \in R_{N_{m-1}} : b(r_{m-1}) \in C_{N_m}) \wedge (\exists r_m \in R_{N_m} : b(r_m) \in C_{N_1})$, *and*
(v) $\forall I \subseteq \mathbb{I} \,.\, \pi(\cdot, I) \in I$.

In the following, we assume application specifications to be well-formed.

3.3 Replica-aware Management of Application Instances

The specification of multi-component applications sets the foundations for modelling the replica- and fault-aware management behaviour of application instances, which we present hereafter. We first introduce the notion of *node instance*, which can intuitively be seen as a named replica of a node in the application specification. When a node is replicated in multiple node instances, each of them is named differently, and it may be residing in a state or performing a transition that is potentially different from that of the other instances of the same node. The instance of a node is hence modelled by a pair indicating its unique identifier and the state or transition it is currently in.

Definition 3.4 (Node Instance). *Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node, and let $\mathbb{I}$ be the universe of possible instance identifiers. An* instance *of the node N is modelled by a pair $\langle i, e \rangle_N$, where*

- *$i \in \mathbb{I}$ is the unique identifier of the node instance,*[5] *and*
- *$e \in (S_N \cup \tau_N)$ is the state or transition in which the instance is residing.*

Notably, despite different instances of a node N constitute different entities, they all feature the same management behaviour, defined by the management protocol of N itself. In other words, the management behaviour of the instances replicating a node is fixed, even if they can simultaneously be residing in different states or performing different transitions.

The states/transitions of the replicated instances of the nodes forming an application constitute the *global state* of the instance of a multi-component application. Such global state also indicates the actual relationships established among the various node instances, i.e., which node instance is (providing the capability) actually used to satisfy the requirements of another node instance. The latter are partitioned between two different sets H and

[3]In the formulas, we follow the convention of using $\uplus$ to denote the disjoint union of sets.
[4]The notions of well-formedness, determinism and race-freedom of management protocols, as well as the techniques for automatically completing them, are reported in the supplemental material available online.
[5]The application management behaviour (Definition 3.9) ensures that each newly spawned instance of an application component is assigned with a "fresh" identifier uniquely identifying the instance.

V, which distinguish the "horizontal" relationships defined by replica-aware and replica-unaware requirements from the "vertical" relationships defined by containment requirements.

Definition 3.5 (Global State). *Let $A = \langle T, b, \pi \rangle$ be an application specification, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T. A* global state *for an instance of A is defined by a triple $\langle G, H, V \rangle$ where*

- $G \subseteq \mathbb{I} \times \bigcup_{N \in T} (S_N \cup \tau_N)$ *is the set of currently active node instances,*
- $H \subseteq \mathbb{I} \times \bigcup_{N \in T} (R^A_N \cup R^U_N) \times \mathbb{I}$ *is the set of "horizontal" runtime bindings, and*
- $V \subseteq \mathbb{I} \times \bigcup_{N \in T} R^C_N \times \mathbb{I}$ *is the set of "vertical" runtime bindings.*

The runtime bindings contained in the sets H and V of a global state can be seen as actual "instances" of the relationships defined by the static bindings in the specification of a multi-component application, i.e., by the function b in Definition 3.2. This can be easily observed by looking again at the specification of the application in our motivating scenario (Figure 1), and by comparing it with the example of global state provided by Figure 2. The triple modelling the global state in Figure 2 is also reported hereafter:

$$\langle \{\langle \mathsf{a1}, \mathsf{running}\rangle, \langle \mathsf{a2}, \mathsf{running}\rangle, \langle \mathsf{g1}, \mathsf{working}\rangle, \langle \mathsf{d1}, \mathsf{running}\rangle, \langle \mathsf{m1}, \mathsf{running}\rangle, \langle \mathsf{m2}, \mathsf{running}\rangle, \langle \mathsf{n1}, \mathsf{running}\rangle\},$$
$$\{\langle \mathsf{g1}, \mathsf{backend}, \mathsf{a1}\rangle, \langle \mathsf{a1}, \mathsf{data}, \mathsf{d1}\rangle, \langle \mathsf{a2}, \mathsf{data}, \mathsf{d1}\rangle\}, \{\langle \mathsf{g1}, \mathsf{host}, \mathsf{n1}\rangle, \langle \mathsf{a1}, \mathsf{host}, \mathsf{m1}\rangle, \langle \mathsf{a2}, \mathsf{host}, \mathsf{m2}\rangle\}\rangle$$

The specification indicates that the backend requirement of gui is to be satisfied by the endpoint capability of api. At the same time, the considered global state contains two different instances of api, i.e., a1 and a2, and it shows that the backend requirement assumed by the instance g1 of gui is actually satisfied by the endpoint capability provided by a1, even if a2 is also providing such capability (as both a1 and a2 are in state running, where —as shown by the management protocol in Figure 5(b)— api provides the endpoint capability).

The rationale for splitting runtime bindings in two sets H and V deserves a short digression: The vertical runtime bindings in V define stronger relationship instances if compared with the horizontal runtime bindings in H. Each vertical runtime binding is established when its source node instance is created to track which is its container, and it lasts for the whole lifetime of the source node instance. The same does not hold for horizontal runtime bindings, which can be created and destroyed multiple times to satisfy the requirements of a node instance during its lifetime. Consider again the instance g1 of gui in Figure 2: The vertical runtime binding from g1 to n1 is established when g1 is created, and it lasts until g1 gets destroyed. Conversely, the horizontal runtime binding from g1 to a1 may change over time, e.g., if a1 stops providing the capability for satisfying the requirement backend of g1, and a2 becomes the node instance used to satisfy such requirement.

In other words, horizontal runtime bindings are temporary and dynamic, as they allow switching the node instances used to satisfy their source requirements. Vertical runtime bindings are instead fixed for the entire lifetime of a node instance. If vertical runtime bindings would instead be dynamically set as for horizontal runtime bindings, this would allow a sort of "container jumping" behaviour. Suppose, for example, that an instance i of a node temporarily stops needing its container to run, e.g., since i is temporarily stopped. Once re-started, it again needs the container to run. If a new "vertical" runtime binding would be dynamically created, it may be that the containment requirement of i is bound to another node instance, which would mean that i gets running in a different container with respect to the former, i.e., that it has "jumped" from a container to another. Even if the new binding may not violate any topological constraint, the "container jumping" performed by i would not align with reality. Each contained instance is bound to its container for its entire lifetime, and the only way to migrate a node from a container c to another container c' is by destroying the instance running in c and creating a brand new instance on c'.

In summary, a global state provides the "structure" to represent the current state of the runtime instances of the nodes forming a multi-component application and of the actual bindings set among such instances at runtime.

When starting to concretely manage a multi-component application, no node instance is currently active, nor runtime bindings have been established yet.

Definition 3.6 (Initial Global State). *Let $A = \langle T, b, \pi \rangle$ be an application specification. The* initial global state *for A is $\langle \emptyset, \emptyset, \emptyset \rangle$.*

While managing the application, there will be runtime bindings only for requirements of existing node instances.[6] Such bindings in non-initial global states may however model the fact that some node instance is broken or failed. The case of *broken instances* occurs when currently active node instances continue to exist, even if they were contained in instances that were previously destroyed. When a software component is destroyed, all the components it contains get destroyed as well, e.g., destroying a Docker container results in deleting also all the software running within such container. To reflect this phenomenon in our modelling, we consider the node instances vertically bound to previously destroyed node instances as broken, viz., to be destroyed to align the global state with the real state of the application.

Definition 3.7 (Broken Instances). *Let $A = \langle T, b, \pi \rangle$ be an application specification, let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T, and let $\langle G, H, V \rangle$ be a global state for A. The set of* broken instances *in $\langle G, H, V \rangle$ is defined as:*

$$\mathsf{broken}(G, V) = \{\langle i, \cdot \rangle_N \in G \mid \langle i, \cdot, i' \rangle \in V \wedge \nexists \langle i', \cdot \rangle_{N'} \in G\}.$$

Example. Consider the case of scaling in node from the global state in Figure 2, which results in destroying the instance n1. We obtain the global state

$$\langle \{\langle a1, running\rangle, \langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m1, running\rangle, \langle m2, running\rangle\},$$
$$\{\langle g1, backend, a1\rangle, \langle a1, data, d1\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a1, host, m1\rangle, \langle a2, host, m2\rangle\}\rangle,$$

in which g1 results to be a broken instance. Indeed, g1 is still among the currently active node instances, despite the vertical binding $\langle g1, host, n1\rangle$ indicates that it is contained in n1, which has however been destroyed. We hence have that

$$\mathsf{broken}(\{\langle a1, running\rangle, \langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m1, running\rangle, \langle m2, running\rangle\},$$
$$\{\langle g1, host, n1\rangle, \langle a1, host, m1\rangle, \langle a2, host, m2\rangle\}) = \{\langle g1, working\rangle\}.$$

The above explicitly denotes the fact that, in reality, by destroying the instance n1, we also destroyed the instance g1 that was contained therein. Hence, for the model to align with reality, the instance g1 is to be destroyed as well.

The case of *faults* affecting instances instead occurs when one or more of the currently active node instances need some requirements to keep staying in their actual state or to successfully complete the execution of the transition they are currently performing, even if there is no capability provided for satisfying such requirements. Requirements that are assumed to be satisfied while not bound (at runtime) to capabilities that can satisfy them are considered as *pending faults*. Pending faults need to be suitably handled. Notably, for replica-unaware requirements, such handling may be automatically *resolved* by exploiting suitable capabilities offered by other node instances, if any.

Notation 3.3 (Satisfying Requirements). *Let $A = \langle T, b, \pi \rangle$ be an application specification, let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T (with $\mathcal{M}_N = \langle \bar{s}_N, \tau_N, \rho_N, \chi_N, \varphi_N \rangle$), and let $\langle G, H, V \rangle$ be a global state for A. Let also r be a requirement of a node instance in G. We denote the set of identifiers of node instances in G offering a capability that can satisfy r with*

$$\mathsf{satisfy}(r, G) = \{i \mid \langle i, e \rangle_N \in G \wedge b(r) \in \chi_N(e)\}.$$

[6] The application management behaviour (Definition 3.9) indeed ensures that any global state $\langle G, H, V \rangle$ reached from the initial global state is, by construction, such that $\forall \langle i, \cdot, \cdot \rangle \in H \cup V \,.\, \exists \langle i, \cdot \rangle_N \in G$.

Definition 3.8 (Faults). *Let $A = \langle T, b, \pi \rangle$ be an application specification, let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T (with $\mathcal{M}_N = \langle \bar{s}_N, \tau_N, \rho_N, \chi_N, \varphi_N \rangle$), and let $\langle G, H, V \rangle$ be a global state for A. The set of* pending faults *in $\langle G, H, V \rangle$ is defined as*

$$\text{pending}(G,H,V) = \{\langle i, r\rangle \mid \langle i, e\rangle_N \in G \wedge r \in \rho_N(e) \wedge \\ (\nexists i', e' \,.\, \langle i, r, i'\rangle \in (H \cup V) \wedge \langle i', e'\rangle_{N'} \in G \wedge b(r) \in \chi_{N'}(e'))\},$$

whose subset of resolvable faults *is defined as*

$$\text{resolvable}(G,H,V) = \{\langle i, r\rangle \in \text{pending}(G,H,V) \mid r \in R_N^U \wedge \text{satisfy}(r, G) \neq \emptyset\}.$$

The pending faults in $\text{pending}(G, H, V)$ are hence defined as the set of pairs $\langle i, r \rangle$ such that a requirement r is assumed by an instance i (i.e., $\langle i, e \rangle_N \in G \wedge r \in \rho_N(e)$), even if there is no instance i' that provides a capability satisfying the requirement r (i.e., $\nexists i', e' \,.\, \langle i, r, i' \rangle \in (H \cup V) \wedge \langle i', e' \rangle_{N'} \in G \wedge b(r) \in \chi_{N'}(e')$). The resolvable faults $\text{resolvable}(G, H, V)$ are instead the subset of pending faults, in which each pair $\langle i, r \rangle$ concerns a replica-unaware requirement r (i.e., $r \in R_N^U$), and another node instance can provide the capability needed to satisfy r (i.e., $\text{satisfy}(r, G) \neq \emptyset$).

Example. Consider the following global state for the application in our motivating scenario (Section 2), which involves the same node instances as in Figure 2, but in different states:

$$\langle \{\langle a1, available\rangle, \langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m1, running\rangle, \langle m2, running\rangle, \langle n1, stopped\rangle\}, \\ \{\langle g1, backend, a1\rangle, \langle a1, data, d1\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a1, host, m1\rangle, \langle a2, host, m2\rangle\}\rangle$$

In particular, we have that g1 is working, hence needing both its requirements host and backend to be satisfied (Figure 5(a)). However, none of the two is actually satisfied in the considered global state: The vertical binding $\langle g1, host, n1\rangle$ indicates that the requirement host is to be satisfied by n1, which is however not providing the needed capability in its current state stopped (Figure 5(c)). Similarly, the horizontal binding $\langle g1, backend, a1\rangle$ instead indicates that the requirement backend is to be satisfied by a1, which is however not providing the needed capability in its current state available (Figure 5(b)). We hence have that the requirements host and backend of g1 are faulted, and that the corresponding faults are pending on g1:

$$\text{pending}(\{\langle a1, available\rangle, \langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m1, running\rangle, \langle m2, running\rangle, \langle n1, stopped\rangle\}, \\ \{\langle g1, backend, a1\rangle, \langle a1, data, d1\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a1, host, m1\rangle, \langle a2, host, m2\rangle\}) \\ = \{\langle g1, host\rangle, \langle g1, backend\rangle\}$$

It is also worth noting that the pending fault $\langle g1, backend\rangle$ can be resolved. The requirement backend is indeed a replica-unaware requirement, and the instance a2 is providing the capability needed to satisfy such requirement in its current state running (Figure 5(b)):

$$\text{resolvable}(\{\langle a1, available\rangle, \langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m1, running\rangle, \langle m2, running\rangle, \langle n1, stopped\rangle\}, \\ \{\langle g1, backend, a1\rangle, \langle a1, data, d1\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a1, host, m1\rangle, \langle a2, host, m2\rangle\}) \\ = \{\langle g1, backend\rangle\}$$

The management behaviour for an instance of the multi-component application specified by $A = \langle T, b, \pi \rangle$ is then defined (in Definition 3.9) by a labelled transition system, whose configurations are the possible global states for A, and whose transition relation $\Rightarrow$ enables modelling horizontal scaling of nodes, operation execution on a node instance, and fault propagation and handling.

- The horizontal scaling of node instances is modelled by the transition rules *scaleout*, *scaleout-c* and *scalein*. *scaleout* and *scaleout-c* increase the amount of replicas of a node by adding a new instance, and they distinguish whether the newly created instance is standalone or to be contained in another instance. *scalein* instead reduces the replicas of a node by selectively destroying one of its instances.
- The execution of management operations on node instances is modelled by the transition rules *op-start* and *op-end*. The rule *op-start* models the starting of execution of a management operation on a node instance, while *op-end* enables to observe its successful completion.

- Fault handling and propagation are modelled by the transition rules *resolve-fault*, *handle-fault*, and *destroy*. The rule *resolve-fault* automatically resolves a resolvable fault, by replacing the node instance used to satisfy the corresponding requirement with a node instance providing the needed capability. The rule *handle-fault* models the execution of fault handling transitions to explicitly handle pending faults, while *destroy* destroys broken instances.

The transition relation $\Rightarrow$ is defined by exploiting a support relation $\rightarrow$, so as to allow the rule *destroy* to take precedence over all other rules. While $\rightarrow$ describes the application management behaviour under the assumption that no broken instance is to be destroyed, $\Rightarrow$ prescribes that any other management transition defined by $\rightarrow$ can be executed only if there are *no-broken-instances*. Instead, if there are broken instances, $\Rightarrow$ forces the execution of *destroy* to resolve them, before any further management action can be executed. This aligns the model with the actual behaviour of node instances contained in a destroyed node instance (e.g., by a *scalein*), which are immediately destroyed too.

Definition 3.9 (Application Management Behaviour). *Let $A = \langle T, b, \pi \rangle$ be an application specification and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T (with $\mathcal{M}_N = \langle \bar{s}_N, \tau_N, \rho_N, \chi_N, \varphi_N \rangle$). The* management behaviour *of the application specified by A is modelled by a labelled transition system whose configurations are the global states $\langle G, H, V \rangle$ of A, and whose transition relation $\Rightarrow$ is defined by the following inference rules*

$$\frac{R^C_N = \emptyset \qquad \nexists \langle i, \cdot \rangle_{N'} \in G \qquad G' = G \cup \{\langle i, \bar{s}_N \rangle_N\} \qquad H' = H \cup \{\langle i, r, i' \rangle \mid r \in \rho_N(\bar{s}_N) \wedge i' = \pi(r, \mathsf{satisfy}(r, G))\}}{\langle G, H, V \rangle \xrightarrow{scaleout(N,i)} \langle G', H', V \rangle} \textit{(scaleout)}$$

$$\frac{R^C_N = \{r_c\} \qquad \langle i_c, \cdot \rangle_{N_c} \in G \qquad b(r_c) \in C_{N_c} \qquad \nexists \langle i, \cdot \rangle_{N'} \in G \qquad G' = G \cup \{\langle i, \bar{s}_N \rangle_N\} \qquad H' = H \cup \{\langle i, r, i' \rangle \mid r \in \rho_N(\bar{s}_N) \wedge r \neq r_c \wedge i' = \pi(r, \mathsf{satisfy}(r, G))\} \qquad V' = V \cup \{\langle i, r_c, i_c \rangle\}}{\langle G, H, V \rangle \xrightarrow{scaleout(N,i,i_c)} \langle G', H', V' \rangle} \textit{(scaleout-c)}$$

$$\frac{\langle i, e \rangle_N \in G \qquad G' = G \setminus \{\langle i, e \rangle_N\} \qquad H' = H \setminus \{\langle i, \cdot, \cdot \rangle, \langle \cdot, \cdot, i \rangle \in H\} \qquad V' = V \setminus \{\langle i, \cdot, \cdot \rangle \in V\}}{\langle G, H, V \rangle \xrightarrow{scalein(i)} \langle G', H', V' \rangle} \textit{(scalein)}$$

$$\frac{\langle i, s \rangle_N \in G \qquad \langle s, o, s' \rangle \in \tau_N \qquad G' = (G \setminus \{\langle i, s \rangle_N\}) \cup \{\langle i, \langle s, o, s' \rangle \rangle_N\} \qquad H' = \psi(H, N, i, s, \langle s, o, s' \rangle)}{\langle G, H, V \rangle \xrightarrow{op_{start}(i,o)} \langle G', H', V \rangle} \textit{(op-start)}$$

$$\frac{\langle i, \langle s, o, s' \rangle \rangle_N \in G \qquad \langle i, \cdot \rangle \notin \mathsf{pending}(G, H, V) \qquad G' = (G \setminus \{\langle i, \langle s, o, s' \rangle \rangle_N\}) \cup \{\langle i, s' \rangle_N\} \qquad H' = \psi(H, N, i, \langle s, o, s' \rangle, s')}{\langle G, H, V \rangle \xrightarrow{op_{end}(i,o)} \langle G', H', V \rangle} \textit{(op-end)}$$

$$\frac{\langle i, r \rangle \in \mathsf{resolvable}(G, H, V) \qquad H' = (H \setminus \{\langle i, r, \cdot \rangle \in H\}) \cup \{\langle i, r, \pi(r, \mathsf{satisfy}(r, G)) \rangle\}}{\langle G, H, V \rangle \xrightarrow{resolve(i,r)} \langle G, H', V \rangle} \textit{(resolve-fault)}$$

$$\frac{\langle i, e \rangle_N \in G \qquad \langle i, r \rangle \in (\mathsf{pending}(G, H, V) \setminus \mathsf{resolvable}(G, H, V)) \qquad \langle e, s' \rangle \in \varphi_N \qquad r \notin \rho_N(s') \qquad \nexists \langle e, s'' \rangle \in \varphi_N \,.\, r \notin \rho_N(s'') \wedge \rho_N(s') \subsetneq \rho_N(s'') \qquad G' = (G \setminus \{\langle i, e \rangle_N\}) \cup \{\langle i, s' \rangle_N\} \qquad H' = \psi(H, N, i, e, s')}{\langle G, H, V \rangle \xrightarrow{handle(i,r)} \langle G', H', V \rangle} \textit{(handle-fault)}$$

$$\frac{\langle i, \cdot \rangle_N \in \mathsf{broken}(G, V) \qquad \langle G, H, V \rangle \xrightarrow{scalein(i)} \langle G', H', V' \rangle}{\langle G, H, V \rangle \xRightarrow{destroy(i)} \langle G', H', V \rangle} \quad \textit{(destroy)}$$

$$\frac{\mathsf{broken}(G, V) = \emptyset \qquad \langle G, H, V \rangle \xrightarrow{\alpha} \langle G', H', V' \rangle}{\langle G, H, V \rangle \xRightarrow{\alpha} \langle G', H', V' \rangle} \quad \textit{(no-broken-instances)}$$

where $\psi(H, N, i, e, e')$ denotes how the set of runtime bindings in H are updated when the node instance $\langle i, e \rangle_N$ changes its state to e', i.e.,

$$\begin{aligned}\psi(H, N, i, e, e') = (H \setminus \{\langle i, r, \cdot \rangle \in H \mid r \in \rho_N(e) \setminus \rho_N(e')\}) \\ \cup \{\langle i, r, i' \rangle \mid r \in \rho_N(e') \setminus \rho_N(e) \wedge r \notin R_N^C \wedge i' = \pi(r, \mathsf{satisfy}(r, G))\}\end{aligned}$$

We hereby discuss more in detail the inference rules of the transition system defining the management behaviour of a multi-component application.

- *scaleout* models the creation of a new instance of a node N that does not expose any containment requirement (i.e., $R_N^C = \emptyset$). The identifier i to be assigned to the newly created instance must be unused (i.e., $\nexists \langle i, \cdot \rangle_{N'} \in G$). If this is the case, a new instance of N with identifier i is added to the global state and set to reside in the initial state of N (i.e., $G' = G \cup \{\langle i, \overline{s}_N \rangle_N\}$). The runtime bindings in H are also updated to satisfy the requirements needed by the newly added instance, if possible (i.e., $H' = H \cup \{\langle i, r, i' \rangle \mid r \in \rho_N(\overline{s}_N) \wedge i' = \pi(r, \mathsf{satisfy}(r, G))\}$).
- *scaleout-c* models the creation of a new instance of a node N exposing a containment requirement r_c (i.e., $R_N^C = \{r_c\}$). Its behaviour is analogous to that of *scaleout*, with the only difference of explicitly indicating the node instance i_c used to contain the newly created node instance. If such instance can provide the capability needed to satisfy r_c (i.e., $\langle i_c, \cdot \rangle_{N_c} \in G$ and $b(r_c) \in C_{N_c}$), the new instance $\langle i, \overline{s}_N \rangle_N$ is added to the global state (as for *scaleout*), and the vertical runtime bindings are updated to track that i is contained in i_c (i.e., $V' = V \cup \{\langle i, r_c, i_c \rangle\}$).
- *scalein* models the removal of the active node instance identified by i (i.e., $\langle i, e \rangle_N \in G$). The rule indeed removes such instance from G (i.e., $G' = G \setminus \{\langle i, e \rangle_N\}$), and it destroys the corresponding runtime bindings: all horizontal runtime bindings involving i are deleted (i.e., $H' = H \setminus \{\langle i, \cdot, \cdot \rangle, \langle \cdot, \cdot, i \rangle \in H\}$) to explicitly fault the requirements of the node instances depending on i, if any. Only the vertical runtime binding outgoing from i is instead removed from V (i.e., $V' = V \setminus \{\langle i, \cdot, \cdot \rangle \in V\}$), as the vertical bindings for which i is the target must be kept to enable explicitly marking the source node instances as "broken" (Definition 3.7).
- *op-start* indicates how the global state $\langle G, H, V \rangle$ evolves when a management operation o is started on a node instance $\langle i, s \rangle_N \in G$. For the operation to be started, a corresponding operation transition must be defined in the management protocol of N (i.e., $\langle s, o, s' \rangle \in \tau_N$). As a result, the global state is updated by changing the actual state of the instance (i.e., $G' = (G \setminus \{\langle i, s \rangle_N\}) \cup \{\langle i, \langle s, o, s' \rangle \rangle_N\}$), and the runtime bindings in H are updated accordingly. Such bindings are updated by $\psi(H, N, i, s, \langle s, o, s' \rangle)$, which first removes the bindings of requirements that are no more needed (i.e., the $\langle i, r, \cdot \rangle \in H$ such that $r \in \rho_N(e) \setminus \rho_N(e')$), and then adds new runtime bindings in H for the newly assumed requirements (i.e., $\rho_N(e') \setminus \rho_N(e)$) analogously to *scaleout* and *scaleout-c*.
- *op-end* models the successful completion of an operation transition $\langle s, o, s' \rangle$ on a node instance, provided that no fault is pending on such instance (i.e., $\langle i, \cdot \rangle \notin \mathsf{pending}(G, H, V)$). As a result, the global state of the application is updated analogously to the case of *op-start*.

- *resolve-fault* indicates how to solve a resolvable fault $\langle i, r\rangle \in \mathsf{resolvable}(G, H, V)$. This is done by replacing the runtime binding for r in H, which gets bound to an instance providing the capability needed for satisfying r (i.e., $H' = (H \setminus \{\langle i, r, \cdot\rangle \in H\}) \cup \{\langle i, r, \pi(r, \mathsf{satisfy}(r, G))\rangle\}$).
- *handle-fault* describes how to update the global state to handle the fault $\langle i, r\rangle$ pending on the instance $\langle i, e\rangle_N$. This is done by executing the fault handling transition $\langle e, s'\rangle \in \varphi_N$, which can be executed only if the fault is not resolvable otherwise (i.e., $\langle i, r\rangle \in \mathsf{pending}(G, H, V) \setminus \mathsf{resolvable}(G, H, V)$), and if the transition $\langle e, s'\rangle$ actually handles the faulted requirement (i.e., $r \notin \rho_N(s')$). In addition, among all transitions that can handle the fault, $\langle e, s'\rangle$ is that bringing the instance to the state s' preserving the largest set of assumed requirements (i.e., $\nexists \langle e, s''\rangle \in \varphi_N \,.\, r \notin \rho_N(s'') \wedge \rho_N(s') \subsetneq \rho_N(s'')$). The latter condition ensures that fault handling is deterministic [6].
- *destroy* models the deletion of a broken instance $\langle i, \cdot\rangle_N \in \mathsf{broken}(G, V)$, which is destroyed by automatically invoking the *scalein* rule. The rationale of *destroy* is to align the modelled behaviour with reality, as broken instances are those instances whose container has been destroyed, and which are then to be destroyed as well (Definition 3.7).
- *no-broken-instances* allows *destroy* to take precedence over all other aforementioned rules. While *destroy* can be executed when there is a broken instance, *no-broken-instances* prescribes that all other rules can be executed only if there are no broken instances (i.e., $\mathsf{broken}(G, V) = \emptyset$). The rationale of *no-broken-instances* is to make this explicit, without cluttering the definition of all other rules with the constraint $\mathsf{broken}(G, V) = \emptyset$.

Example. Consider again the application in our motivating scenario (Section 2), and suppose that it is in the global state shown in Figure 2, whose corresponding triple is reported hereafter:

$$\langle \{\langle a1, \mathit{running}\rangle, \langle a2, \mathit{running}\rangle, \langle g1, \mathit{working}\rangle, \langle d1, \mathit{running}\rangle, \langle m1, \mathit{running}\rangle, \langle m2, \mathit{running}\rangle, \langle n1, \mathit{running}\rangle\},$$
$$\{\langle g1, \mathit{backend}, a1\rangle, \langle a1, \mathit{data}, d1\rangle, \langle a2, \mathit{data}, d1\rangle\}, \{\langle g1, \mathit{host}, n1\rangle, \langle a1, \mathit{host}, m1\rangle, \langle a2, \mathit{host}, m2\rangle\}\rangle$$

Suppose now that we scale in the node maven by selectively removing its instance m1. The corresponding evolution of the global state is the following:

$$\langle \{\langle a1, \mathit{running}\rangle, \langle a2, \mathit{running}\rangle, \langle g1, \mathit{working}\rangle, \langle d1, \mathit{running}\rangle, \langle m1, \mathit{running}\rangle, \langle m2, \mathit{running}\rangle, \langle n1, \mathit{running}\rangle\},$$
$$\{\langle g1, \mathit{backend}, a1\rangle, \langle a1, \mathit{data}, d1\rangle, \langle a2, \mathit{data}, d1\rangle\}, \{\langle g1, \mathit{host}, n1\rangle, \langle a1, \mathit{host}, m1\rangle, \langle a2, \mathit{host}, m2\rangle\}\rangle$$
$$\Downarrow\ \mathit{scalein}(m1)$$
$$\langle \{\langle a1, \mathit{running}\rangle, \langle a2, \mathit{running}\rangle, \langle g1, \mathit{working}\rangle, \langle d1, \mathit{running}\rangle, \langle m2, \mathit{running}\rangle, \langle n1, \mathit{running}\rangle\},$$
$$\{\langle g1, \mathit{backend}, a1\rangle, \langle a1, \mathit{data}, d1\rangle, \langle a2, \mathit{data}, d1\rangle\}, \{\langle g1, \mathit{host}, n1\rangle, \langle a1, \mathit{host}, m1\rangle, \langle a2, \mathit{host}, m2\rangle\}\rangle$$

In the reached global state, a1 turns out to be a broken instance, as it is contained in an instance that has been previously destroyed (as indicated by the vertical binding $\langle a1, \mathit{host}, m1\rangle$). We hence have to apply the destroy rule for deleting such broken instance and aligning the situation modelled by the global state with reality (where a1 got destroyed together with the deletion of m1):

$$\langle \{\langle a1, \mathit{running}\rangle, \langle a2, \mathit{running}\rangle, \langle g1, \mathit{working}\rangle, \langle d1, \mathit{running}\rangle, \langle m2, \mathit{running}\rangle, \langle n1, \mathit{running}\rangle\},$$
$$\{\langle g1, \mathit{backend}, a1\rangle, \langle a1, \mathit{data}, d1\rangle, \langle a2, \mathit{data}, d1\rangle\}, \{\langle g1, \mathit{host}, n1\rangle, \langle a1, \mathit{host}, m1\rangle, \langle a2, \mathit{host}, m2\rangle\}\rangle$$
$$\Downarrow\ \mathit{destroy}(a1)$$
$$\langle \{\langle a2, \mathit{running}\rangle, \langle g1, \mathit{working}\rangle, \langle d1, \mathit{running}\rangle, \langle m2, \mathit{running}\rangle, \langle n1, \mathit{running}\rangle\},$$
$$\{\langle a2, \mathit{data}, d1\rangle\}, \{\langle g1, \mathit{host}, n1\rangle, \langle a2, \mathit{host}, m2\rangle\}\rangle$$

No broken instance is contained in the reached global state, but we have that there is a pending fault: The instance g1 is in state working, where it needs its requirement backend to be satisfied (Figure 5(a)). As per the topology of our motivating example (Figure 1), such requirement should be satisfied by binding it to the capability endpoint offered by a currently active instance of api. However, none of the horizontal bindings in the reached global state are binding the requirement backend of g1 to one such capability. At the same time, there exists an instance a2 of api that is currently active and offering the capability endpoint needed to satisfy the requirement backend of g1. Given that such requirement is replica-unaware, we can resolve its corresponding fault

by dinamically adapting the horizontal bindings:

$$\begin{aligned}
&\langle \{\langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle \\
&\ \Downarrow\ resolve(g1, backend) \\
&\langle \{\langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle g1, backend, a2\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle
\end{aligned}$$

Suppose now that, in the reached global state, we invoke the operation for stopping the instance a2 of api, and that we observe its completion. The corresponding evolution of the global state is shown hereafter:

$$\begin{aligned}
&\langle \{\langle a2, running\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle g1, backend, a2\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle \\
&\ \Downarrow\ op_{start}(a2, stop) \\
&\langle \{\langle a2, \langle running, stop, available\rangle\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle g1, backend, a2\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle \\
&\ \Downarrow\ op_{end}(a2, stop) \\
&\langle \{\langle a2, available\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle g1, backend, a2\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle
\end{aligned}$$

In the reached global state, the requirement backend of g1 is again faulted. Such requirement is bound to a2, which is however not providing the capability needed to satisfy g1's backend in its current state available (Figure 5(b)). No other instance of api is available, hence not allowing to dynamically resolve the fault pending on g1 and corresponding to its requirement backend. Such fault is hence to be handled by executing the corresponding fault handler on g1, which leads g1 back to its state configured (Figure 5(a)):

$$\begin{aligned}
&\langle \{\langle a2, available\rangle, \langle g1, working\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle g1, backend, a2\rangle, \langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle \\
&\ \Downarrow\ handle(g1, backend) \\
&\langle \{\langle a2, available\rangle, \langle g1, configured\rangle, \langle d1, running\rangle, \langle m2, running\rangle, \langle n1, running\rangle\}, \\
&\ \{\langle a2, data, d1\rangle\}, \{\langle g1, host, n1\rangle, \langle a2, host, m2\rangle\}\rangle
\end{aligned}$$

4 ANALYSING THE MANAGEMENT OF HORIZONTALLY SCALABLE APPLICATIONS

The modelled management behaviour of a multi-component application enables automating various analyses on the management of horizontally scalable applications, which are all brand new and first presented in this article. Such analyses include the validation of planned management (Section 4.2), and automatically determining management plans allowing to reach and maintain a desired global state (Section 4.3). To simplify the presentation of such analyses, we first introduce an abstraction over the rules defining the management behaviour of an application (Section 4.1), which enables focusing only on the actions that can be actually taken by a human or software agent orchestrating the management of the node instances forming a multi-component application.

4.1 Focusing on Observable Actions

The transition rules defining the management behaviour of a multi-component application (Definition 3.9) include actions that can be executed by an agent orchestrating the application, i.e., *scaleout*, *scaleout-c*, and *scalein* for horizontally scaling its nodes, and *op-start* and *op-end* to invoking management operations and observe their completion. At the same time, the rules in Definition 3.9 also model the internal evolution of the application due to fault handling and propagation, i.e., *resolve-fault* and *handle-fault* to handle faults, and *destroy* to automatically propagate the deletion of a node instance to the node instances it contains.

We hereby introduce an abstraction over the management behaviour of multi-component applications, which enables focusing only on the actions that can be executed by an agent to orchestrate their management, while at

the same time abstracting from the internal evolution of the application. When an agent executes an action to manage an application (e.g., it starts a management operation on a node instance), it can observe the effects of the propagation and handling of potential faults due to such action. At the same time, it is not the agent that invokes fault handlers or explicitly propagates the faults. Fault handling and propagation are indeed internal to the application itself: they are never directly executed by the agent, even if they can be caused by the actions it takes.

The proposed abstraction, hereafter called "*observable behaviour*", is a labelled transition system that hides the execution of fault handling and propagation actions (i.e., *resolve-fault*, *handle-fault*, and *destroy* in Definition 3.9). The observable behaviour enables reasoning only in terms of the actions for horizontally scaling nodes, for starting management operations on node instances, and for observing their termination. Notably, given that an operation may terminate also due to fault handling, the observable behaviour enables observing the termination of an operation in both cases. All other internal actions due to fault handling and propagation are absorbed by observable actions.

Definition 4.1 (Observable Behaviour). *Let $A = \langle T, b, \pi \rangle$ be an application specification and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a generic node in T (with $\mathcal{M}_N = \langle \bar{s}_N, \tau_N, \rho_N, \chi_N, \varphi_N \rangle$). The* observable behaviour *of the application specified by A is modelled by a labelled transition system whose configurations are the global states $\langle G, H, V \rangle$ of A, and whose transition relation $\mapsto$ is defined by the following inference rules*

$$\frac{\langle G,H,V\rangle \overset{\alpha}{\Longrightarrow} \langle G',H',V'\rangle \qquad \alpha \in \{scaleout(\cdot,\cdot), scaleout(\cdot,\cdot,\cdot), scalein(\cdot), op_{start}(\cdot,\cdot), op_{end}(\cdot,\cdot)\}}{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G',H',V'\rangle} \textit{ (exec)}$$

$$\frac{\langle G,H,V\rangle \overset{handle(i,\cdot)}{\Longrightarrow} \langle G',H',V'\rangle \qquad \langle i,\langle\cdot,o,\cdot\rangle\rangle_N \in G}{\langle G,H,V\rangle \overset{op_{end}(i,o)}{\longmapsto} \langle G',H',V'\rangle} \textit{ (op-fault)}$$

$$\frac{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G',H',V'\rangle \qquad \langle G',H',V'\rangle \overset{resolve(i,\cdot)}{\Longrightarrow} \langle G'',H'',V''\rangle}{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G'',H'',V''\rangle} \textit{ (absorbe-r)}$$

$$\frac{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G',H',V'\rangle \qquad \langle i,s\rangle_N \in G' \qquad s \in S_N \qquad \langle G',H',V'\rangle \overset{handle(i,\cdot)}{\Longrightarrow} \langle G'',H'',V''\rangle}{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G'',H'',V''\rangle} \textit{ (absorbe-h)}$$

$$\frac{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G',H',V'\rangle \qquad \langle G',H',V'\rangle \overset{destroy(\cdot)}{\Longrightarrow} \langle G'',H'',V''\rangle}{\langle G,H,V\rangle \overset{\alpha}{\mapsto} \langle G'',H'',V''\rangle} \textit{ (absorbe-d)}$$

We hereby discuss more in detail the inference rules of the transition system defining the observable behaviour of a multi-component application.

- *exec* enables executing the actions for horizontally scaling components (i.e., *scaleout* and *scalein*), and for starting management operations on node instances and observing their termination (i.e., op_{start} and op_{end}, respectively). Such operations can all be executed by the agent orchestrating the management of an application, and they must hence be observable from the transition system $\mapsto$ defining the observable behaviour.

- *op-fault* enables observing the termination of a management operation also due to a fault affecting the corresponding node instance. The rule also unifies this event with that of the successful completion of the execution of a management operation, whose labels are equal in the transition system $\mapsto$ defining the observable behaviour.
- *absorbe-r*, *absorbe-h* and *absorbe-d* hide the internal evolution due to the handling and propagation of faults, whose outcomes are absorbed by the executable action α that caused the corresponding handling and propagation of faults. Note that *op-fault* and *absorbe-h* are mutually exclusive, with the latter only absorbing the handling of faults affecting a node instance that resides in one of its possible states (i.e., $\langle i, s \rangle_N \in G'$ and $s \in S_N$).

In summary, the observable behaviour of a multi-component application enables reasoning on how to orchestrate its management by focusing only on executable actions, i.e., *scaleout* and *scalein* for increasing or decreasing the replicas of a node, op_{start} for starting the execution of a management operation on a node instance, and op_{end} for observing the termination of the execution of a management operation on a node instance (independently of whether such operation completed successfully or was terminated by a fault). By looking at the global state reached after executing one of the above actions, one can then understand its outcomes (therein included the effects of handling and propagating the faults potentially caused —and absorbed— by such action).

4.2 Analysing Planned Management

The observable behaviour of a multi-component application enables validating its already planned management. A sequence of actions to be executed in a global state can be considered *valid* if (and only if) the actions it contains can *always* be executed in the given order.

Definition 4.2 (Valid Management Sequence). *Let $A = \langle T, b, \pi \rangle$ be an application specification and let $\langle G, H, V \rangle$ be a global state for A. A sequence of management actions $\alpha_1 \alpha_2 ... \alpha_n$ is* valid *in $\langle G, H, V \rangle$ iff the corresponding predicate holds:*

$$\mathsf{valid}(\alpha_1 \alpha_2 ... \alpha_n, \langle G, H, V \rangle) \Leftrightarrow$$
$$(n = 0) \ \vee \ (\forall \langle G', H', V' \rangle : \langle G, H, V \rangle \overset{\alpha_1}{\longmapsto} \langle G', H', V' \rangle \,.\, \mathsf{valid}(\alpha_2 ... \alpha_n, \langle G', H', V' \rangle))$$

The notion of validity for management sequences sets also the foundations for validating plans orchestrating the management of a multi-component application A (i.e., acyclic workflows orchestrating the management operations of the instances of the components in A), such as those in Figure 3 and Figure 4. A management plan P_A defines a partial ordering on the actions for orchestrating the management of A, viz., it indicates which actions must be completed before executing other actions. In addition, each management operation invoked by P_A corresponds to two subsequent actions on the corresponding node instance, viz., starting the execution of the management operation and observing its completion (in this order). The resulting partial ordering can be visualised as a DAG, whose vertices model actions, and where each arc indicates that the action corresponding to its source must be executed before that corresponding to its target.

Example. Consider the management plan in Figure 3(a). Such plan indicates that the execution of the operation stop on the node instance g1 must be completed before starting the operations config on the node instances g1, a1, and a2. Such instances can execute their operations config in parallel, and —only when all of them are completed— the plan restarts g1 by executing its operation start. The partial ordering among the corresponding actions is displayed as a DAG in Figure 6.

Notably, the topological sorts of the DAG corresponding to a management plan P_A define its "sequential traces", i.e., all the possible sequences of management actions executing all actions in P_A, while at the same time respecting the ordering constraints defined by P_A itself [7]. The validity of management plan P_A can hence be defined in terms of that of its sequential traces: a management plan P_A is *valid* if all its sequential traces are valid.

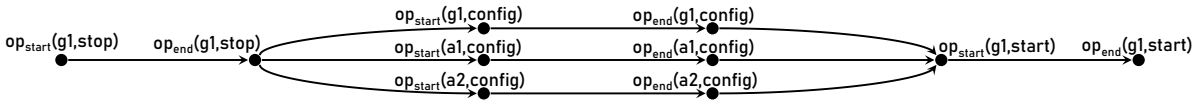


Fig. 6. DAG defined by the management plan in Figure 3(a).

A *weaker* notion of validity is also defined, which requires that *some* sequential traces (but not necessarily all traces) are valid. A plan is instead considered *not valid* if none of the above applies.

Notation 4.1 (Sequential Traces). *Let P_A be a plan for orchestrating the management of a multi-component application A. We denote with* $\mathsf{traces}(P_A)$ *the set of all* sequential traces *of P_A.*

Definition 4.3 (Valid Management Plan). *Let $A = \langle T, b, \pi \rangle$ be an application specification and let $\langle G, H, V \rangle$ be a global state for A. A plan P_A orchestrating the management of A from $\langle G, H, V \rangle$ is* valid *or* weakly valid *in $\langle G, H, V \rangle$ iff the corresponding predicate holds:*

- $\mathsf{valid}(P_A, \langle G, H, V \rangle) \Leftrightarrow \forall \alpha_1 ... \alpha_n \in \mathsf{traces}(P_A) \,.\, \mathsf{valid}(\alpha_1 ... \alpha_n, \langle G, H, V \rangle)$,
- $\mathsf{weaklyValid}(P_A, \langle G, H, V \rangle) \Leftrightarrow \exists \alpha_1 ... \alpha_n \in \mathsf{traces}(P_A) \,.\, \mathsf{valid}(\alpha_1 ... \alpha_n, \langle G, H, V \rangle)$.

The notion of valid plan enables checking whether the plans for reconfiguring the application in our motivating scenario (Figure 3) validly reconfigure such application from the global state in Figure 2. One can readily check that all sequential traces of the plan in Figure 3(b) are valid, hence meaning that such plan is valid. This means that whichever is the parallel execution of the actions in the plan, we can always execute all of them.

Weak validity is instead useful for refining plans, especially in the case of weakly valid plans that are not fully valid: even if such plans include non-valid sequential traces that will never complete, the information in their valid sequential traces can be used to refine weakly valid plans and obtain fully valid plans that will always succeed being executed. A concrete example of this is given by the plan in Figure 3(a). Some of its sequential traces are not valid, e.g.,

$$op_{start}(\mathrm{g1}, \mathrm{stop})\ op_{end}(\mathrm{g1}, \mathrm{stop})\ op_{start}(\mathrm{g1}, \mathrm{config})\ op_{start}(\mathrm{a1}, \mathrm{config})\ op_{start}(\mathrm{a2}, \mathrm{config})$$
$$\underline{op_{end}(\mathrm{g1}, \mathrm{config})}\ op_{end}(\mathrm{a1}, \mathrm{config})\ op_{end}(\mathrm{a2}, \mathrm{config})\ op_{start}(\mathrm{g1}, \mathrm{start})\ op_{end}(\mathrm{g1}, \mathrm{start})$$

Suppose that we execute the latter sequence in the global state in Figure 2. When observing that the operation config on g1 has completed (through the underlined action $op_{end}(\mathrm{g1}, \mathrm{config})$), we would realise that such operation faulted. This is because, to successfully complete, the operation config on an instance of gui requires the requirement backend to continue to be satisfied, but this is not the case when both instances of the api are executing their operation config (as they are not providing the capability endpoint needed for satisfying such requirement —Figure 5(b)). This hence results in a fault for g1, which the latter handles by getting back to its state installed (Figure 5(a)). The node instance g1 then remains in such state until the trace reaches the action $op_{start}(\mathrm{g1}, \mathrm{start})$. However, there is no transition allowing to execute the operation start on an instance of gui residing in state installed. This means that the action $op_{start}(\mathrm{g1}, \mathrm{start})$ cannot be executed, which in turn means that the considered sequential trace is not valid.

At the same time, there exist sequential traces of the plan in Figure 3(a) that are valid, e.g.,

$$op_{start}(\mathrm{g1}, \mathrm{stop})\ op_{end}(\mathrm{g1}, \mathrm{stop})\ op_{start}(\mathrm{g1}, \mathrm{config})\ op_{start}(\mathrm{a1}, \mathrm{config})\ op_{end}(\mathrm{g1}, \mathrm{config})$$
$$op_{end}(\mathrm{a1}, \mathrm{config})\ op_{start}(\mathrm{a2}, \mathrm{config})\, op_{end}(\mathrm{a2}, \mathrm{config})\ op_{start}(\mathrm{g1}, \mathrm{start})\ op_{end}(\mathrm{g1}, \mathrm{start}).$$

In this case, when we execute the action $op_{end}(\mathrm{g1}, \mathrm{config})$ for observing the termination of config on g1, there is one instance of api providing the capability needed to successfully complete the operation, i.e., a2, to which g1 has been reconnected after the (resolvable) fault caused by the fact that a1 started executing its operation *config*. The operation config on g1 hence successfully completes and brings g1 to its state configured, from which it can later execute its operation start.

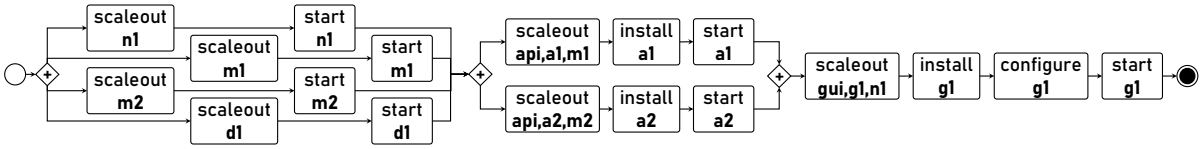


Fig. 7. Management plan for deploying the application in our motivating scenario.

The above means that the plan in Figure 3(a) is weakly valid, hence meaning that some of the parallel executions of the actions forming the plan successfully terminate. Notably, all such traces are those in which the operation config on *g1* is not executed in parallel with that of config on both a1 or a2. This information can hence be exploited to refine the plan to avoid this to happen, e.g., by postponing the execution of the operation config on *g1* in the weakly valid plan in Figure 3(a), which would result in obtaining a valid plan analogous to that in Figure 3(b).

Similar considerations can be applied to the much more complex case of the management plans for restarting the instances of node and maven in our motivating scenario (Figure 4). While the management plan in Figure 4(b) is valid, management plan in Figure 4(a) is only weakly valid. Again, the information given by the valid sequential traces of the weakly valid plan in Figure 4(a) could be used to refine it and obtain a management plan analogous to that in Figure 4(b).

The management behaviour of a multi-component application and its observable behaviour can be exploited not only for checking the validity of management plans, or refining them, but also for various other purposes. For instance, it is also interesting to understand what happens to an application A when executing a valid plan P_A. One may indeed wish to determine how many instances of the nodes in A are there after executing P_A, which are their actual states, which capabilities they provide, which requirements they assume to be satisfied, and whether/how requirements and capabilities have been bound. Such information can be excerpted directly from the global state(s) reached by the sequential traces of P_A.

In addition, 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. Intuitively, a plan is deterministic if, independently of how parallel actions are executed, it always ends up in instantiating the same amount of replicas for each node, and it always brings each replica to the same state. In other words, a plan is deterministic if, given any two of its sequential traces, such traces bring to global states including the same sets of active node instances.

Definition 4.4 (Deterministic Plan). *Let $A = \langle T, b, \pi \rangle$ be an application specification, let $\langle G, H, V \rangle$ be a global state for A, and let P_A be a valid management plan for A in $\langle G, H, V \rangle$. The plan P_A is also deterministic iff*

$$\forall \alpha_1 ... \alpha_n, \alpha'_1 ... \alpha'_m \in \mathsf{traces}(P_A)\ .$$
$$\langle G, H, V \rangle \overset{\alpha_1}{\longmapsto} ... \overset{\alpha_n}{\longmapsto} \langle G', \cdot, \cdot \rangle \ \wedge\ \langle G, H, V \rangle \overset{\alpha'_1}{\longmapsto} ... \overset{\alpha'_m}{\longmapsto} \langle G'', \cdot, \cdot \rangle \ \wedge\ G' = G''.$$

Example. Consider again the application in our motivating scenario (Section 2), and suppose that we developed the deployment plan in Figure 7. The plan validly deploys the application, by first creating and starting an instance of node (viz., n1), two instances of maven (viz., m1 e m2), and an instance of mongo (viz., d1). It then creates, installs, and starts two instances of the api (viz., a1 and a2). When started, both a1 and a2 connect to d1, the only running instance of mongo. Afterwards, it creates, installs, configures, and starts an instance of gui. The instance of gui can be connected to any of the instances of the api, as both are already running and offering their endpoint capability when the latter is needed to satisfy the requirements the instance of gui.

As a result, the application can reach the global state in Figure 2, as well as an analogous global state, where the only difference is that g1 connects to a2 (instead of a1), e.g., if the connection policy function is such that the instance for satisfying the backend requirements of gui instances is randomly chosen. Independently from which horizontal bindings are set, however, the deployment plan Figure 7 can be considered deterministic, as it always creates the same node instances and brings them to the same state.

4.3 Automatically Planning Application Management, Validly

The observable behaviour of a multi-component application also enables automatically planning the orchestration of its management, i.e., finding whether there is a sequence of management operations allowing an application to go from a global state $\langle G, H, V \rangle$ to a target configuration G' (indicating how many replicas of each node must be there, and which is the desired state for each of such replicas). Such a planning problem can obviously be solved with a breadth-first search (BFS) over the graph defined by the labelled transition system of the observable behaviour of the application, and starting from $\langle G, H, V \rangle$. Unfortunately, such a graph is not finite, as we can scale out as many replicas as we wish, hence meaning that we potentially have infinite replicas of each node forming the application. This in turn means that a BFS over such graph may not terminate: even if there is no sequence of management operations for changing the global state of the application from $\langle G, H, V \rangle$ to some global state $\langle G', \cdot, \cdot \rangle$, the BFS could continue traversing arcs corresponding to the action of scaling out one of the nodes in the application, hence never ending.

A solution to this is to define a finite search space ensuring that, if no sequence is found in such a search space, then there is no sequence at all. We hereafter provide the bricks for building such a search space by first defining a set of *support node instances*. Each support node instance is included "ad-hoc" to enable satisfying a given requirement of a given node instance that is in the starting or target global state (and only that requirement of that instance). The set of support node instances contains also instances for satisfying support node instances themselves. This is because, for example, for a support node instance to get in a state where it can satisfy the requirement it is associated with, such a support node instance may need its requirements to get satisfied as well. Hence, having a support node instance for each requirement of each support node instance enables to spawn support node instances whenever they are needed, i.e., whenever the corresponding requirement of a node instance in the starting or target global state needs to be satisfied.

Definition 4.5 (Support Node Instances). *Let $A = \langle T, b, \pi \rangle$ be an application specification and let $\langle G, \cdot, \cdot \rangle$ and $\langle G', \cdot, \cdot \rangle$ be two global states for A. The set* $\mathsf{supp}(G, G')$ *of* support node instances *for the node instances in G and G' is one of the smallest sets satisfying the following conditions:*

(i) $\forall \langle i, \cdot \rangle_N \in G \cup G' \cup \mathsf{supp}(G, G') \,.\, \forall r \in R_N : b(r) \in C_{N'} \,.\, \exists \langle j, \cdot \rangle_{N'} \in \mathsf{supp}(G, G')$

(ii) $\forall \langle i, \cdot \rangle_N \in G \cup G' \cup \mathsf{supp}(G, G') \,.$
$\forall r, r' \in R_N : (r \neq r' \wedge b(r) \in C_{N'} \wedge b(r') \in C_{N''}) \,.\, \exists \langle j, \cdot \rangle_{N'}, \langle j', \cdot \rangle_{N''} \in \mathsf{supp}(G, G') : j \neq j'$

(iii) $\forall \langle i, \cdot \rangle_N, \langle i', \cdot \rangle_{N'} \in G \cup G' \cup \mathsf{supp}(G, G') \,.$
$\forall r \in R_N, r' \in R_{N'} : b(r) \in C_{N''} \wedge b(r') \in C_{N''} \,.\, \exists \langle j, \cdot \rangle_{N''}, \langle j', \cdot \rangle_{N''} \in \mathsf{supp}(G, G') : j \neq j'$

Intuitively, the conditions in Definition 4.5 indicate that (i) each requirement of each node instance in $G \cup G' \cup \mathsf{supp}(G, G')$ has a support node instance, (ii) different requirements of the same node instance have different support node instances, and (iii) different node instances have different support node instances. The set of support node instance is one of the smallest sets satisfying such conditions, and it is also *finite*.

Proposition 4.1 (Finiteness of Support Node Instances). *Let $A = \langle T, b, \pi \rangle$ be an application specification and let $\langle G, \cdot, \cdot \rangle$ and $\langle G', \cdot, \cdot \rangle$ be two global states for A. Any set* $\mathsf{supp}(G, G')$ *of support node instances is* finite.

Proof. The thesis can be proved constructively. Given that G and G' are finite (by Definition 3.5), we have a finite number of node instances, each with a finite number of requirements needing a support node instance. Being R the set containing all such requirements, we can create a different support node instance for each requirement in R. Notably, given that the function b is total, we can always identify the node satisfying a requirement, which is instantiated in a support node instance.

The newly created set of support node instances may contain node instances exposing themselves some requirements. Being R' the set containing all such requirements, we can again create a different support node instance for each requirement in R'. We can then continue with the above approach until there are no more

support node instances lacking of a support node instance, i.e., exposing some requirements for which separate support node instance have not been created yet. This is guaranteed to happen, as A is assumed to be well-formed, which means that its topology is acyclic. Indeed, the lack of cycles of dependencies (together with the finiteness of application topologies) ensures that, for each requirement to be satisfied, there is a finite chain of nodes to be instantied to satisfy such requirement, whose last node is without any requirement. We hence eventually reach an iteration in which the newly added support node instances do not expose any requirement, hence completing the construction process.

The above presented approach terminates, as it enables building the set of support node instances after a finite number of iterations. Given that each iteration consists in adding a finite set of support node instances, it directly follows that the set of support node instances is finite as well. □

The support node instances enable defining a finite search space for determining whether there is a sequence of management actions allowing an application to move from global state $\langle G, H, V\rangle$ to a target configuration G'. Such a search space, hereafter called "*planning graph*", is given by a labelled graph whose vertices are global states for the application, and whose labelled arcs correspond to actions leading from a global state to another (according to the observable behaviour of the application — Definition 4.1). In addition, the graph (i) contains the starting global state $\langle G, H, V\rangle$, (ii) it is obtained by executing all possible sequences of management actions, and it is such that the (iii) considered global states involve —at most— the node instances in the starting global state, in the target configuration, and in a corresponding set of support node instances.

Definition 4.6 (Planning Graph). *Let $A = \langle T, b, \pi\rangle$ be an application specification, let $\langle G, H, V\rangle$ and $\langle G', \cdot, \cdot\rangle$ be two global states for A, and let* $\mathsf{supp}(G, G')$ *be a set of support node instances for G and G'. Let also $\mathbb{L}$ denotes the universe of all possible actions in the observable behaviour of A. The* planning graph *from the global state $\langle G, H, V\rangle$ to the configuration G' is a labelled graph* $\mathsf{pGraph}(G, G') = \langle W, E\rangle$ *where:*

- $W \subseteq \mathcal{P}(\mathbb{I} \times \bigcup_{N \in T} S_N \cup \tau_N) \times \mathcal{P}(\mathbb{I} \times (\bigcup_{N \in T} R^A_N \cup R^U_N) \times \mathbb{I}) \times \mathcal{P}(\mathbb{I} \times \bigcup_{N \in T} R^C_N \times \mathbb{I})$ *is the set of vertices, each corresponding to a global state for A,*
- $E \subseteq W \times \mathbb{L} \times W$ *is the set of arcs, each labelled with a management action,*

and they both satisfy the following conditions:

(i) $\langle G, H, V\rangle \in W$,
(ii) $\forall w, w' \in W\,.\,(\exists \langle w, \alpha, w'\rangle \in E \Leftrightarrow w \overset{\alpha}{\mapsto} w')$, *and*
(iii) $\forall \langle G'', \cdot, \cdot\rangle \in W\,.\,\forall \langle i, \cdot\rangle_N \in G''\,.\,\exists \langle i, \cdot\rangle_N \in G \cup G' \cup \mathsf{supp}(G, G')$

The conditions (i-iii) in Definition 4.6 enable constructively building the planning graph: We can indeed build $\mathsf{pGraph}(G, G')$ by starting with W only containing (i) the vertex corresponding to the starting global state, and then iteratively expand W and E by (ii) adding vertices and arcs for each action that can be performed in a global state in W for which a corresponding arc has not been added to E yet. More precisely, each iteration consists in adding, for each global state in W, the arcs corresponding to actions that can be executed in such global state and that are not yet in E, along with the global states that can be reached with such actions (if not already in W). Notably, condition (iii) ensure that the planning graph is finite, hence meaning that the above construction process eventually terminates.

Proposition 4.2 (Finiteness of Planning Graph). *Let $A = \langle T, b, \pi\rangle$ be an application specification, and let $\langle G, \cdot, \cdot\rangle$ and $\langle G', \cdot, \cdot\rangle$ be two global states for A. The planning graph* $\mathsf{pGraph}(G, G') = \langle W, E\rangle$ *for G and G' is* finite.

Proof. The thesis directly follows from condition (iii) in Definition 4.6, which prescribes that the node instances of any global state in W to be (at most) those contained in the starting global state $\langle G, H, V\rangle$, in the target configuration G', and in the considered set of support node instances $\mathsf{supp}(G, G')$. This means that the set

W is (at most) as big as all possible combinations of all possible instances for each node in A in all their possible states, with all possible runtime bindings. The possible instances in A are finitely limited by condition (iii), while their possible states are finitely limited by their management protocols. In addition, the possible bindings are limited as well, as connecting finite sets of requirements to finite sets of capabilities in all possible (but finite) ways. It hence follows that the set W is finite, as all possible combinations of all possible instances for each node in A in all their possible states (with all possible runtime bindings) are finite as well. □

The planning graph defines a finite search space where to look for management sequences allowing an application to reach the target configuration G' (i.e., to reach some global state $\langle G', \cdot, \cdot \rangle$) by starting from $\langle G, H, V \rangle$. We indeed now prove that there exists one such management sequence if and only if there is a path in the planning graph leading from $\langle G, H, V \rangle$ to some $\langle G', \cdot, \cdot \rangle$.

Theorem 4.1 (Soundness and Completeness of Planning). *Let $A = \langle T, b, \pi \rangle$ be an application specification, let $\langle G, H, V \rangle$ and $\langle G', \cdot, \cdot \rangle$ be two global states for A, and let* $\mathsf{pGraph}(G, G') = \langle W, E \rangle$ *be the planning graph for G and G'.*

$$\exists \alpha_1 ... \alpha_n \,.\, \mathsf{valid}(\alpha_1 ... \alpha_n, \langle G, H, V \rangle) \wedge \langle G, H, V \rangle \overset{\alpha_1}{\longmapsto} ... \overset{\alpha_n}{\longmapsto} \langle G', \cdot, \cdot \rangle$$
$$\Updownarrow$$
$$\exists \textit{ a path } \alpha'_1 ... \alpha'_m \textit{ in } \mathsf{pGraph}(G, G') \textit{ from } \langle G, H, V \rangle \textit{ to some } \langle G', \cdot, \cdot \rangle.^{7}$$

Proof. The case ($\Uparrow$) directly follows from the definition of planning graph (Definition 4.6), for which $\mathsf{valid}(\alpha'_1 ... \alpha'_m, \langle G, H, V \rangle)$ holds.

The case ($\Downarrow$) can instead be proved by contradiction. Assume there is no path in $\mathsf{pGraph}(G, G')$ leading from $\langle G, H, V \rangle$ to some $\langle G', \cdot, \cdot \rangle$, i.e.,

$$\nexists \textit{ a path in } \mathsf{pGraph}(G, G') \textit{ from } \langle G, H, V \rangle \textit{ to any } \langle G', \cdot, \cdot \rangle, \tag{1}$$

and suppose that there exists a valid management sequence $\alpha_1 \alpha_2 ... \alpha_n$ leading from $\langle G, H, V \rangle$ to $\langle G', H_n, V_n \rangle$ not corresponding to a path in $\mathsf{pGraph}(G, G')$. We have that $\langle G, H, V \rangle \in \mathsf{pGraph}(G, G')$ and that $\langle G', H_n, V_n \rangle$ is obtained by applying a sequence of management actions. By definition of planning graph (Definition 4.6), it must necessarily be that condition (1) is because of the valid sequence traversing an intermediate global state $\langle G_i, H_i, V_i \rangle$ that is not in $\mathsf{pGraph}(G, G')$, i.e.,

$$(\langle G, H, V \rangle \overset{\alpha_1}{\longmapsto} \langle G_1, H_1, V_1 \rangle \overset{\alpha_2}{\longmapsto} ... \overset{\alpha_n}{\longmapsto} \langle G', H_n, V_n \rangle) \wedge (\langle G_i, H_i, V_i \rangle \notin \mathsf{pGraph}(G, G')), \tag{2}$$

where we slightly abuse the notation to say that the global state $\langle G_i, H_i, V_i \rangle$ is not contained in the set of vertices of $\mathsf{pGraph}(G, G')$.

The planning graph $\mathsf{pGraph}(G, G')$ however includes all the global states that can be reached from $\langle G, H, V \rangle$ by applying executable management actions, and by only considering the node instances that are in G, G', or in a corresponding set $\mathsf{supp}(G, G')$ of support node instances. For condition (2) to hold, it must hence be that

$$\exists \langle \hat{\imath}, \cdot \rangle_N \in G_i \,.\, \langle \hat{\imath}, \cdot \rangle_N \notin (G \cup G' \cup \mathsf{supp}(G, G')). \tag{3}$$

The same does not hold for all previously traversed global states, which (by Definition 4.6) are all part of $\mathsf{pGraph}(G, G')$, i.e.,

$$\forall j \in [1, i) \,.\, \langle G_j, H_j, V_j \rangle \in \mathsf{pGraph}(G, G'). \tag{4}$$

Conditions (3) and (4) mean that the node instance $\langle \hat{\imath}, \cdot \rangle_N$ has been created by the action α_i, i.e.,

$$\alpha_i = \mathit{scaleout}(\hat{\imath}, N) \ \vee \ \alpha_i = \mathit{scaleout}(\hat{\imath}, \cdot, N). \tag{5}$$

[7] Formally, $\exists \{ \langle \langle G, H, V \rangle, \alpha'_1, \langle G_1, H_1, V_1 \rangle \rangle, \langle \langle G_1, H_1, V_1 \rangle, \alpha'_2, \langle G_2, H_2, V_2 \rangle \rangle ... \langle \langle G_{m-1}, H_{m-1}, V_{m-1} \rangle, \alpha'_m, \langle G', \cdot, \cdot \rangle \rangle \} \subseteq E$.

At the same time, since $\langle \hat{\imath}, \cdot \rangle_N \notin G'$, and since (1) is assumed to hold, there must be an action occurring after α_i that destroys the instance identified by $\hat{\imath}$, i.e.,

$$\exists j \in (i, n] \,.\, \alpha_j = \mathit{scalein}(\hat{\imath}). \tag{6}$$

This is coherent with our initial assumptions (1) and (2), as it would mean that part of the management sequence $\alpha_1 ... \alpha_n$ necessarily occurred outside of the planning graph, i.e.,

$$\langle G, H, V \rangle \overset{\alpha_1}{\longmapsto} ... \overset{\alpha_i}{\longmapsto} \langle G_i, H_i, V_i \rangle \overset{\alpha_{i+1}}{\longmapsto} ..[\,\star\,].. \overset{\alpha_j}{\longmapsto} \langle G_j, H_j, V_j \rangle \overset{\alpha_{j+1}}{\longmapsto} ... \overset{\alpha_n}{\longmapsto} \langle G_n, H_n, V_n \rangle \tag{7}$$

with the evolution identified by the $[\,\star\,]$ area is surely occurring outside of $\mathsf{pGraph}(G, G')$. None of its vertices indeed correspond to a global state containing the node instance identified by $\hat{\imath}$, while all the global states between $\langle G_i, H_i, V_i \rangle$ and $\langle G_j, H_j, V_j \rangle$ contain such node instance.

In addition, for condition (1) to hold, we must also have that

$$\langle G_j, H_j, V_j \rangle \notin \mathsf{pGraph}(G, G'). \tag{8}$$

Otherwise, such global state would be reachable from the starting global state $\langle G, H, V \rangle$, and by Definition 4.6 also $\langle G', H_n, V_n \rangle$ would be in $\mathsf{pGraph}(G, G')$. This would then mean that there would be a path from $\langle G, H, V \rangle$ to $\langle G', H_n, V_n \rangle$ in $\mathsf{pGraph}(G, G')$, which would contradict condition (1).

For simplicity, assume now that $\langle \hat{\imath}, \cdot \rangle_N$ is the only additional node instance involved in the $[\,\star\,]$ area in (7). This is not a restriction, as the following reasoning can be readily be generalised and repeated for any number of additional instances.[8]

The global state $\langle G_j, H_j, V_j \rangle$ has been obtained from $\langle G_{i-1}, H_{i-1}, V_{i-1} \rangle$ by applying a sequence of management actions $\alpha_i ... \alpha_j$ allowed by the observable behaviour of A. By Definition 4.1, we hence have that all the runtime bindings in H_j and V_j are generated analogously to all other global state that can be reached from $\langle G_{i-1}, H_{i-1}, V_{i-1} \rangle$, therein included those in $\mathsf{pGraph}(G, G')$. In addition, condition (6) ensures that the additional instance $\langle \hat{\imath}, \cdot \rangle_N$ is not in G_j, which hence only contains node instances that are also in G, G', or $\mathsf{supp}(G, G')$, i.e.,

$$\forall \langle i, \cdot \rangle_N \in G_j \,.\, \exists \langle i, \cdot \rangle_N \in (G \cup G' \cup \mathsf{supp}(G, G')) \tag{9}$$

For condition (8) to hold, it must hence be that there is at least one management action α_k in the $[\,\star\,]$ area in (7) that was enabled, while it cannot be fired in $\mathsf{pGraph}(G, G')$. In particular, since *scalein* can always be fired, and since no other *scaleout* action has been issued (as we assumed $\langle \hat{\imath}, \cdot \rangle_N$ to be the only additional node instance), we must have that α_k is some operation-related transition that can be fired outside of the $\mathsf{pGraph}(G, G')$, while it cannot be fired within $\mathsf{pGraph}(G, G')$. For this to hold, and given the rules for enacting op_{start} and op_{end} while managing A (Definition 3.9), it must be that there is a requirement that can be satisfied by the node instances in the $[\,\star\,]$ area in (7), while it is not possible to satisfy it in the planning graph $\mathsf{pGraph}(G, G')$.

The above leads however to a *contradiction*: The planning graph $\mathsf{pGraph}(G, G')$ is intentionally built by including support node instances so that whenever a requirement is needed we can spawn a support instance for satisfying it (Definition 4.5 and Definition 4.6). We hence obtain a contradiction, from which we can deduce that $\langle G_j, H_j, V_j \rangle \in \mathsf{pGraph}(G, G')$, which in turn means that there is a path in $\mathsf{pGraph}(G, G')$ from $\langle G, H, V \rangle$ to $\langle G', H_n, V_n \rangle$. This contradicts our initial hypothesis in condition (1), hence meaning that also the case ($\Downarrow$) of our theorem holds. □

Obvious consequences of Theorem 4.1 are that a BFS over a planning graph enables checking whether there is a management sequence allowing an application to move from a global state to a target configuration, and that such a check is guaranteed to terminate. In addition, if the BFS finds a path, the latter actually corresponds to the

[8] As none of them can be in G', for condition (1) to hold, this means that we eventually reach a global state where they have all been destroyed. We can then repeat an analogous reasoning from such global state.

sequence of management actions allowing the application to move from the starting global state to the target configuration.

Performing a BFS over a (potentially huge) planning graph is expensive, especially if the outcome of such search is that there is no management sequence for going from a global state to another. However, there are cases in which we can be sure that there is a management sequence allowing to an application to reach the target configuration, which we characterise hereafter. First, we observe that we can always bring an application back to its initial global state from any actual global state it actually reached. This can be done by scaling in all the node instances in the reached global state, as this results in letting the application to step back to a global state with no node instances nor runtime bindings, i.e., its initial global state.

Proposition 4.3 (Resettability). *Let $A = \langle T, b, \pi \rangle$ be an application specification. It is always possible to reach the initial global state $\langle \emptyset, \emptyset, \emptyset \rangle$ from each global state $\langle G, H, V \rangle$ of A.*

Proof. The thesis trivially follows from observing that the initial global state is equal to the state reached after destroying all the node instances in G by performing *scalein* actions. □

Proposition 4.3 enables characterising those situations in which we are sure that there is a plan allowing an application to go from its current global state to another. Given a target configuration G' for an application, if a corresponding global state $\langle G', \cdot, \cdot \rangle$ can be reached from the initial global state $\langle \emptyset, \emptyset, \emptyset \rangle$, then there is always a plan allowing to reach such target configuration from each possible global state of the application.

Theorem 4.2 (Reachability). *Let $A = \langle T, b, \pi \rangle$ be an application specification, and let $\langle G', \cdot, \cdot \rangle$ be a global state for A.*

$$\exists \alpha_1 ... \alpha_n \,.\, \mathsf{valid}(\alpha_1 ... \alpha_n, \langle \emptyset, \emptyset, \emptyset \rangle) \wedge \langle \emptyset, \emptyset, \emptyset \rangle \overset{\alpha_1}{\longmapsto} ... \overset{\alpha_n}{\longmapsto} \langle G', \cdot, \cdot \rangle \Rightarrow$$

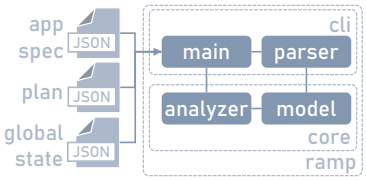
$$\forall \langle G, H, V \rangle \text{ for } A, \exists \alpha'_1 ... \alpha'_m \,.\, \mathsf{valid}(\alpha'_1 ... \alpha'_m, \langle \emptyset, \emptyset, \emptyset \rangle) \wedge \langle G, H, V \rangle \overset{\alpha'_1}{\longmapsto} ... \overset{\alpha'_m}{\longmapsto} \langle G', \cdot, \cdot \rangle.$$

Proof. The thesis obviously follows from Proposition 4.3, which states that that we can always reach the initial global state $\langle \emptyset, \emptyset, \emptyset \rangle$ from every global state of an application. □

The above presented reachability result (Theorem 4.2) ensures that a multi-component application can always get recovered from any possible fault affecting its node instances. Consider a global state $\langle G, H, V \rangle$ reached by executing a management sequence/plan in the initial global state, and suppose that some fault occurs and causes changes of states to some of its node instances, which result in the application getting into the global state $\langle G', H', V' \rangle$. In the worst case, it may be that some of the node instances in G' have become unable to perform any other management operation, i.e., they entered in a sort of "sink" state, where there is no outgoing transition allowing to execute some management operation. Theorem 4.2 however ensures that —even in the worst case— there exists a management sequence allowing the application to get back to the $\langle G, H, V \rangle$, which can be computed with a BFS over a planning graph.[9] In other words, whichever is the global state $\langle G, H, V \rangle$ reached by performing management actions from the initial global state, and whichever are the faults affecting the application in such global state, there always exists a management sequence allowing to recover from such faults, hence enabling to maintain the global state $\langle G, H, V \rangle$.

5 PROOF-OF-CONCEPT IMPLEMENTATION

[9]This provides a much stronger result if compared with the "hard recovery" in [6]. While the latter was assuming to have *only* one instance of each node at a time, and it was able to recover *only* those nodes that were hosted on other nodes, Theorem 4.2 ensures that we can recover *all* node instances that changed their state because of some fault.



(a)

```
FAILED SEQUENCE:
 opStart g1 stop
 opEnd g1 stop
 opStart g1 config
 opStart a1 config
 opStart a2 config
 opEnd g1 config
 opEnd a1 config
 opEnd a2 config
 opStart g1 start
 opEnd g1 start
```

```
ACTIVE INSTANCES
 api      a1   running
 api      a2   running
 node     n1   running
 maven    m1   running
 maven    m2   running
 gui      g1   installed
 mongo    d1   running

FAILED ACTION:
 opStart g1 start
```

(b)

Fig. 8. The (a) architecture of RAMP and (b) a snippet of its output.

To illustrate the feasibility of our solution, we developed RAMP (Replica-Aware Management Protocols), a proof-of-concept implementation of replica-aware management protocols and of their plan validation support. We also used RAMP to analyse the planned management for the multi-component application in our motivating scenario (Section 2).

RAMP is implemented in Java, open-source, and publicly available on GitHub.[10] It provides a CLI (Command-Line Interface) for checking the validity/weak validity of a management plan in a global state of a given application. As shown in Figure 8(a), RAMP takes as input an application specification, the plan to validate, and the global state where the plan should be executed, all given as JSON files. The JSON files are passed to the main module implementing the CLI, which interacts with the parser to generate an internal representation of the application, plan, and global state in the RAMP model (viz., a Java object model enabling to represent the specification of multi-component applications, their management behaviour, and management plans). Such representation is then passed to the analyzer to check the validity/weak validity of the given plan in the given global state. If the check fails, RAMP prints information on why the check failed (Figure 8(b)).

The output information given by RAMP can be useful to the application operator to determine and resolve the issues affecting the given plan. We experimented this by intentionally developing four non-valid plans, one for deploying the application in our motivating scenario from scratch and three for managing it from the global state in Figure 2, with the latters including those in Figure 3(a) and Figure 4(a). We also specified in JSON the application in Figure 1 and the global state in Figure 2. Then, we ran RAMP to check the validity of the plans, which —as expected— were classified by RAMP as not valid. The output provided by RAMP enabled us to observe which trace of each plan cannot successfully complete, the action in such trace that cannot be executed, and the global state reached right before executing such action, e.g., as in Figure 8(b) for the plan in Figure 3(a). We exploited such information to refactor the plans so as to ensure that the shown issue could not happen again, e.g., that g1 cannot be started because it resides in state installed when the operation start is executed, which can only be due to the fact that the configure operation formerly invoked on g1 faulted because no instance of api was capable of satisfying g1's needed requirements. We repeated the above process of checking the validity of refactored plans, always exploiting the outcomes provided by RAMP to further refactor them if they were not valid, until we obtained fully valid plans, such as those in Figure 3(b) and Figure 4(b).[11]

6 RELATED WORK

Automating the management of multi-component applications became a hot topics in IT research after the cloud revolution [18]. The proliferation of so-called "configuration management systems" also witnesses the need for

[10] https://github.com/di-unipi-socc/ramp.

[11] The JSON files and guidelines for repeating our experiments are publicly available on GitHub at https://github.com/di-unipi-socc/ramp/tree/master/data/thinking-app.

management automation by the IT industry, with *Chef* and *Puppet* being two of the most prominent examples.[12] Configuration management systems provide domain-specific languages to describe the desired configuration of an application, which is enforced though the usage of daemons ensuring that such configuration is eventually met. At the same time, the lack of a machine-readable representation of how to effectively manage cloud application components inhibits the possibility of performing automated analyses on their configurations and dependencies.

A first formal modelling of the allowed management of multi-component applications was given by the Aeolus component model [9]. We share the idea of Aeolus [9] of describing the management behaviour of the application components through finite-state machines (enriched with conditions describing what is offered and required in a state) and of deriving the management behaviour of an application by composing those of its components. Aeolus however differs from our approach as it focuses on automating the deployment and configuration of an application, i.e., instantiating a given number of replicas of each application component and bringing them running. We instead focus on the whole management lifecycle of the application, from its first deployment to managing all instances of its components, therein included recovering such instances from possible faults.

Another noteworthy solution is Engage [13]. Given a partial installation specification, Engage [13] can determine a full installation plan, which coordinates the deployment and configuration of the instances of the components forming a multi-component application across multiple machines. Engage differs from our approach since it focuses on determining a deployment of applications whose components are not to be replicated. We instead focus on the management of applications, by also enabling to replicate components and to explicitly model faults, analysing the effects of faults on component instances, and reacting to faults to restore a desired application configuration.

Similar considerations apply to the solution by Breitenbücher et al. [5], which allows to automatically generate executable deployment plans from an application specification given in the OASIS standard TOSCA [17]. Given such specification, their solution automatically determines a sequence of management operations allowing to reach a target configuration. Breitenbücher et al. [5] however focuses on application deployment, by allowing to instantiate each application component only once. Our approach instead supports the whole management of the lifecycle of applications, therein included instantiating multiple replicas of each application component and allowing to restore the desired configuration for an application after some faults occurred.

Etchevers et al. [11] propose a decentralized protocol for self-deployment of cloud applications. Etchevers et al. [11] model cloud applications as a set of virtual machines (VMs), whose interconnections are modelled by connecting the requirements of a VM (called *imports*) to the capabilities of another (called *exports*). Such a structuring is then exploited to coordinate the deployment of a given number of replicas of the VMs forming an application, by also fixing the allowed management behaviour for such VMs. In addition, the internals of each VM are hidden, by assuming that the installation, configuration, and enactment of the software components in a VM have already been planned. Our approach instead enables separately modelling the software components forming an application from the computing nodes used to run them, and to fully customise their management behaviour. Another difference between our approach and that by Etchevers et al. [11] is that they allow handling only environmental faults (due to networing issues), while our approach can also deal with application-specific faults.

Similar considerations apply to the fault-resilient deployment solution proposed by Duran and Salaün [10]. They also model applications as a set of interconnected VMs. In addition, they provide each VM with a module managing its lifecycle, called "configurator". An orchestrator then coordinates the deployment and reconfiguration of an application by interacting with the configurators of the VMs forming the application. The solution by Duran and Salaün [10] is hence related to our approach as it focuses managing multi-component applications by taking into account faults and by specifying the management of each component separately. Their solution however

[12] *Chef*: https://www.chef.io. *Puppet*: https://puppet.com.

differs from our approach since it only considers environmental faults, while we also deal with application-specific faults. In addition, whilst our approach enables instantiating multiple replicas of each component forming an application, the approach by Duran and Salaün [10] fixes the topology once for all and considers one instance of each VM forming the application at a time.

Friedrich et al. [14] instead propose an approach to handle faults is very close in the spirit to ours, even if for a different application domain. As we do for multi-component applications, Friedrich et al. [14] exploit a model-based approach to specify the workflows orchestrating multiple Web services in a business processes, 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). The approach by Friedrich et al. [14] however differs from our approach mainly because of the different application domain, which allows them to exploit different techniques (such as heuristics based on branching probabilities) to carry out their analyses, to reason with only one replica of each service at a time, and to assume faults to happen on one service at a time.

Other approaches worth mentioning relate to the model-driven engineering of horizontally scalable multi-component applications. Vaquero et al. [21] discuss various scaling strategies, both for software components and for the computing nodes used to host them. The authors propose a rule-based engine, whose aim is to dynamically scale application components according to user-defined rules. Despite Vaquero et al. [21] deal with both horizontal and vertical scaling, their approach relies on users specifying how to actually manage the scaling of components. Our appraoch instead provides a compositional modelling for the management of horizontally scalable application, which enables automatically planning the management of such applications.

Toffetti et al. [20] propose a solution for developing self-managing applications. More precisely, they propose a distributed algorithm for component instances to synchronize between themselves and reach a desired configuration. The target configuration is expressed by means of a typed graph, whose cardinalities on the edges represent the desired number of instances of each node. The approach by Toffetti et al. [20] shares our idea of distinguishing the modelling of the topology of an application from that of its runtime instances. They also share our idea of obtaining recoverable applications, whose components are monitored by means of health monitors and restarted if some health probes report some faults. At the same time, their approach is intended to be actually integrated in the development of application, hence requiring to update the application sources to include an implementation of the distributed algorithm proposed by Toffetti et al. [20]. Our approach instead provides a design-time solution for modelling and analysing the management of horizontally scalable applications, without requiring to update their sources.

Finally, it is worth relating our approach to those dealing with the rigorous engineering of fault-tolerant systems. Betin-Can et al. [2] and Wang et al. [22] provide tools for fault-localisation in complex applications, which allows to re-design such applications by avoiding the occurrence of identified faults. Johnsen et al. [15] illustrates a methodology for designing fault-resilient applications, which starts from a fault-free application, and which allows to iteratively refine the application by identifying and handling the faults that may occur. All such approaches however differ from our approach as they aim to obtain applications that "never fail", because potential faults are already identified and handled. Our approach is instead more recovery-oriented [8], as we embrace the idea that applications can (and probably will) fail, in both expected and unexpected ways, and we aim at supporting the design of applications that can be recovered when faults occur.

In summary, to the best of our knowledge, ours is the first approach enabling to automatically analyse and plan the replica-aware management of a multi-component application, while at the same time considering the concurrent management of the instances of its components, and the fact that expected and unexpected faults might happen while such concurrent management is being executed. We do so by building upon the results of our previous work [7], which employs finite state machines to describe the behaviour of the application components

(including how they react to both expected and unexpected faults), and by extending such model to keep track of the runtime evolution of the system, including how instances are created, destroyed, and interconnected.

7 CONCLUSIONS

Automatically coordinating the management of multi-component applications is crucial nowadays [16, 24]. It requires to suitably coordinate the concurrent deployment, configuration, enactment, and termination of possibly multiple instances of the components forming an application, while at the same time taking into account all inter-component dependencies, as well as that different replicas of the same component may reside in different states and get affected by different faults.

Starting from an already existing approach for statically specifying the structure and management behaviour of multi-component applications, we proposed a novel approach enabling to model and analyse the actual management of instances of specified applications. More precisely, after formally defining how to specify multi-component applications by means of topology graphs [4] and management protocols [6, 7], we introduced a novel compositional modelling for the actual management of applications, which distinguishes among the possibly multiple replicas of each node in the application, and enables suitably coordinating their management, therein included how they react to possible faults. We then exploited such modelling to formalise various useful analyses, from validating existing management plans and checking their effects, to automatically determining management plans allowing an application to reach a target configuration. In the latter case, we also observed that the search space where to look for such management plans is inherently infinite, as the same component may be possibly infinitely replicated, hence generating infinitely many possible application configurations. We anyhow formally proved that checking whether the target configuration can be reached and the management plan for actually reaching it can be both finitely computed, and discussed a solution for doing so.

The presented solution for planning is however quite expensive, as it requires to constructively build a possibly huge search space where to look for a plan. As part of our future work, we plan to devise more efficient algorithms enabling to find a plan for reaching a target configuration, e.g., based on the contextual information given by the starting global state and the target configuration (i.e., involved replicas, target state of each replica), or based on suitable heuristics. We also plan to extend RAMP to feature a graphical environment to specify and analyse the management of multi-component applications, therein included planning their management, and to use the extended RAMP to validate our approach against real-world case studies. Finally, we plan to extend the current approach to also consider QoS and costs while analysing the management of applications, e.g., for driving the horizontal scaling of application components to improve the overall QoS of the application, or to derive the cheapest plan allowing to reach a target application configuration.

REFERENCES

[1] Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, and Frank Leymann. 2018. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* 51, 1, Article 22 (2018), 38 pages. https://doi.org/10.1145/3150227

[2] Aysu Betin-Can, Tevfik Bultan, Mikael Lindvall, Benjamin Lux, and Stefan Topp. 2007. Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. *Autom. Softw. Eng.* 14 (07 2007), 129–178. https://doi.org/10.1007/s10515-007-0008-2

[3] Tobias Binz, Uwe Breitenbücher, Oliver Kopp, and Frank Leymann. 2014. TOSCA: Portable Automated Deployment and Management of Cloud Applications. In *Advanced Web Services*. Springer, New York, NY, USA, 527–549. https://doi.org/10.1007/978-1-4614-7535-4_22

[4] Tobias Binz, Christoph Fehling, Frank Leymann, Alexander Nowak, and David Schumm. 2012. Formalizing the Cloud through Enterprise Topology Graphs. In *CLOUD 2012*. IEEE, USA, 742–749. https://doi.org/10.1109/CLOUD.2012.143

[5] Uwe Breitenbücher, Tobias Binz, Kálmán Képes, Oliver Kopp, Frank Leymann, and Johannes Wettinger. 2014. Combining Declarative and Imperative Cloud Application Provisioning based on TOSCA. In *IC2E 2014*. IEEE, USA, 87–96. https://doi.org/10.1109/IC2E.2014.56

[6] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. 2018. Fault-aware management protocols for multi-component applications. *J. Syst. Softw.* 139 (2018), 189–210. https://doi.org/10.1016/j.jss.2018.02.005

[7] Antonio Brogi, Andrea Canciani, and Jacopo Soldani. 2018. True Concurrent Management of Multi-component Applications. In *ESOCC 2018*. Springer, Cham, 17–32. https://doi.org/10.1007/978-3-319-99819-0_2
[8] George Candea, Aaron Brown, Armando Fox, and David Patterson. 2004. Recovery-oriented computing: Building multitier dependability. *Computer* 37 (12 2004), 60–67. https://doi.org/10.1109/MC.2004.219
[9] Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. 2014. Aeolus: A component model for the cloud. *Information and Computation* 239 (2014), 100–121. https://doi.org/10.1016/j.ic.2014.11.002
[10] Francisco Durán and Gwen Salaün. 2016. Robust and reliable reconfiguration of cloud applications. *J. Syst. Softw.* 122 (2016), 524–537. https://doi.org/10.1016/j.jss.2015.09.020
[11] Xavier Etchevers, Gwen Salaün, Fabienne Boyer, Thierry Coupaye, and Noël De Palma. 2017. Reliable Self-deployment of Distributed Cloud Applications. *Software: Practice and Experience* 47, 1 (2017), 3–20. https://doi.org/10.1002/spe.2400
[12] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. 2014. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer-Verlag, Wien, Austria.
[13] Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. 2012. Engage: A Deployment Management System. *SIGPLAN Not.* 47, 6 (2012), 263–274. https://doi.org/10.1145/2345156.2254096
[14] G. Friedrich, M. G. Fugini, E. Mussi, B. Pernici, and G. Tagni. 2010. Exception Handling for Repair in Service-Based Processes. *IEEE Transactions on Software Engineering* 36, 2 (March 2010), 198–215. https://doi.org/10.1109/TSE.2010.8
[15] Einar Broch Johnsen, Olaf Owe, Ellen Munthe-Kaas, and Juri Vain. 2001. Incremental Fault-Tolerant Design in an Object-Oriented Setting. In *APAQS 2001*. IEEE, USA, 223–230. https://doi.org/10.1109/APAQS.2001.990023
[16] Nane Kratzke and Peter-Christian Quint. 2017. Understanding cloud-native applications after 10 years of cloud computing: A systematic mapping study. *J. Syst. Softw.* 126 (2017), 1–16. https://doi.org/10.1016/j.jss.2017.01.001
[17] OASIS. 2013. Topology and Orchestration Specification for Cloud Applications, Version 1.0.
[18] Claus Pahl, Pooyan Jamshidi, and Olaf Zimmermann. 2018. Architectural Principles for Cloud Software. *ACM Trans. Internet Technol.* 18, 2, Article 17 (2018), 23 pages. https://doi.org/10.1145/3104028
[19] Jacopo Soldani, Damian Andrew Tamburri, and Willem-Jan Van Den Heuvel. 2018. The pains and gains of microservices: A Systematic grey literature review. *J. Syst. Softw.* 146 (2018), 215–232. https://doi.org/10.1016/j.jss.2018.09.082
[20] Giovanni Toffetti, Sandro Brunner, Martin Blöchlinger, Florian Dudouet, and Andrew Edmonds. 2015. An Architecture for Self-Managing Microservices. In *AIMC 2015*. ACM, USA, 19–24. https://doi.org/10.1145/2747470.2747474
[21] Luis M. Vaquero, Luis Rodero-Merino, and Rajkumar Buyya. 2011. Dynamically Scaling Applications in the Cloud. *SIGCOMM Comput. Commun. Rev.* 41, 1 (Jan. 2011), 45–52. https://doi.org/10.1145/1925861.1925869
[22] Qiang Wang, Lei Yan, Simon Bliudze, and Xiaoguang Mao. 2015. Automatic Fault Localization for BIP. In *SETTA 2015 (LNCS, Vol. 9409)*. Springer, Cham, 277–283. https://doi.org/10.1007/978-3-319-25942-0_18
[23] Bill Wilder. 2012. *Cloud Architecture Patterns*. O'Reilly Media, USA.
[24] Michael Wurster, Uwe Breitenbücher, Antonio Brogi, Frank Leymann, and Jacopo Soldani. 2020. Cloud-native Deploy-ability: An Analysis of Required Features of Deployment Technologies to Deploy Arbitrary Cloud-native Applications. In *CLOSER 2020*. SciTePress, Setúbal, Portugal, 171–180. https://doi.org/10.5220/0009571001710180