# Modelling the Dynamic Reconfiguration of Application Topologies, Faults Included

Antonio Brogi, Andrea Canciani, Jacopo Soldani

# Modelling the Dynamic Reconfiguration of Application Topologies, Faults Included

Antonio Brogi, Andrea Canciani, and Jacopo Soldani

Department of Computer Science, University of Pisa, Italy

**Abstract.** Fault-aware management protocols permit modelling the management of application components (including potential faults) and analysing the management behaviour of a multi-component application. The analysis is driven by the application topology, and it assumes many-to-1 dependencies among application components, i.e. each requirement of a component can be satisfied by exactly one other component.

In this paper we extend fault-aware management protocols to account for many-to-many dependencies among components, i.e. different application components can be used to satisfy a requirement of another component. The extensions also accounts for dynamic changes in the topology, hence enabling the analysis of the management behaviour of dynamically reconfigurable multi-component applications.

## 1 Introduction

How to automatically manage composite applications is currently one of the major concerns in enterprise IT [4,20]. Composite applications typically integrate various heterogeneous components, like in microservice-based applications [15], and the deployment, configuration, enactment, and termination of the components forming a composite application must be suitably coordinated, by taking into account all dependencies occurring among application components.

It is worth noting that, while coordinating the management of a composite application, we must "enable failure" [19]. Namely, we must be aware that the components forming a composite application may fail, and we should be able to react to such failures by restoring the desired application configuration.

A convenient way to represent the structure of a composite application is a topology graph [5], whose nodes represent the application components, and whose arcs represent the dependencies occurring among such components. Each topology node can be associated with the requirements of a component, the capabilities it features, and the operations to manage it. Inter-node dependencies associate the requirements of a node with capabilities featured by other nodes.

In [7] we showed how the management behaviour of topology nodes can be modelled by *fault-aware management protocols*, specified as finite state machines whose states and transitions are associated with conditions defining the consistency of a node's states and constraining the executability of its management operations. Such conditions are defined on the requirements of a node, and each

requirement of a node has to be fulfilled by a capability of another node. Fault-aware management protocols also permit modelling how a node behaves when a fault occurs (viz., when a node is assuming a requirement to be satisfied by a capability, and such capability stops being provided by the corresponding node). The management behaviour of a composite application can then be easily derived by composing the management protocols of its nodes according to the dependencies defined in its topology.

Fault-aware management protocols (as per their definition in [7]) assume many-to-1 dependencies among components. Namely, while a capability can be used to satisfy multiple requirements, a requirement can be satisfied only by one capability. This is a quite strict assumption, which impedes modelling and analysing applications where a requirement of a node can be satisfied by different capabilities offered by different nodes (e.g., a microservice-based application where a microservice requires a certain API, and alternative implementations of such an API are offered by different microservices).

In this paper we present a proper extension of fault-aware management protocols that relaxes the above mentioned assumption, allowing application topologies to have many-to-many dependencies among components. More precisely, we not only allow a capability to be connected to multiple requirements (viz., the set of requirements it can satisfy), but also a requirement to be connected to multiple capabilities (viz., the set of capabilities that can be used to satisfy such requirement). Whenever a node needs one of its requirements, any capability connected to such requirement can be used to satisfy it, and faults are raised whenever a node stops providing a capability actually satisfying a requirement. The extension also permits indicating whether the actual binding of a requirement can be dynamically changed while executing a management operation or while handling a pending fault, hence allowing to dynamically reconfigure the topology of an application.

We then illustrate how the management behaviour of a composite application can be derived by composing the management protocols of its nodes according to the static dependencies defined in its topology. We also show how this permits automating various useful analyses, like determining whether a plan orchestrating the management of a composite application is "valid", which are its effects (e.g., which capabilities are available after executing it, or whether it can dynamically reconfigure an application), and finding management plans achieving specific goals (e.g., reaching a desired application configuration, or dynamically reconfiguring an application to recover it from a fault).

The rest of the paper is organised as follows. Sect. 2 illustrates a scenario motivating the need for extending fault-aware management protocols to account for many-to-many dependencies in application topologies. Such extension is then formalised in Sect. 3. Sect. 4 shows how to automatically compose management protocols to analyse and automate the management of a composite application in presence of faults. Finally, Sects. 5 and 6 discuss related work and draw some concluding remarks, respectively.

## 2   Motivating scenario

Consider the (toy) microservice-based application in Fig. 1, which is composed by
a JavaScript-based microservice acting as web-based frontend of the application,
two alternative and interchangeable implementations of a backend microservice,
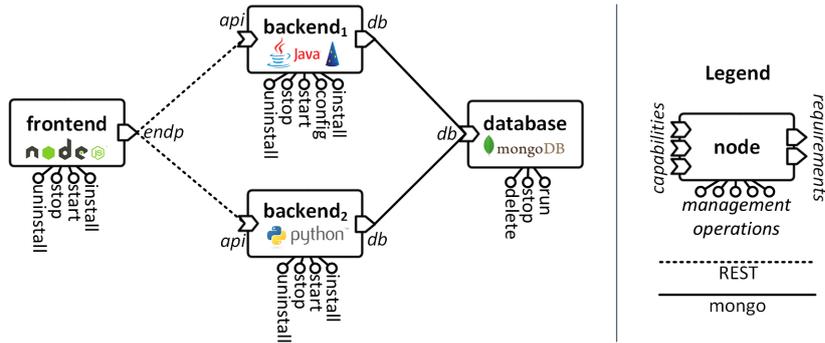and a microservice hosting a Mongo database. The frontend microservice exposes



**Fig. 1.** Motivating example.

a requirement endp, as it needs to know which is the endpoint where to invoke
the RESTful api offered by the backend microservices. These in turn expose a
requirement db, as they need to set up a persistent connection to the microservice
hosting a Mongo database.

Suppose that we wish to orchestrate the deployment of the whole application
with a dedicated management plan. Since the represented application topology
does not include any management protocol, one may produce invalid manage-
ment plans. For instance, while Fig. 2 illustrates three seemingly valid plans, only
(a) is a valid plan. Plan (b) is not valid since the operation config of $backend_1$
must be executed before actually starting the node (to provide $backend_1$ with
the necessary information to set up the connection with the Mongo database).
Plan (c) is not valid either, since database must be running before $backend_1$ and
$backend_2$ can start (as this also results in connecting them to the database).

Suppose now that all microservices have been deployed, started, and properly
connected (e.g., by executing plan (a) in Fig. 2), with frontend being configured
to invoke the api offered by $backend_2$. What happens if the operation stop of
$backend_2$ is executed? The microservice $backend_2$ is stopped, and this may gen-
erate a fault in frontend, as it becomes unable to serve its clients simply because
the api that it remotely invokes is not offered anymore by $backend_2$. A simple yet
effective solution would be to dynamically reconfigure the microservice frontend
to invoke the api offered by $backend_1$. Even worse is the case when database is
stopped, as this causes a fault in $backend_1$ and $backend_2$, which become unable
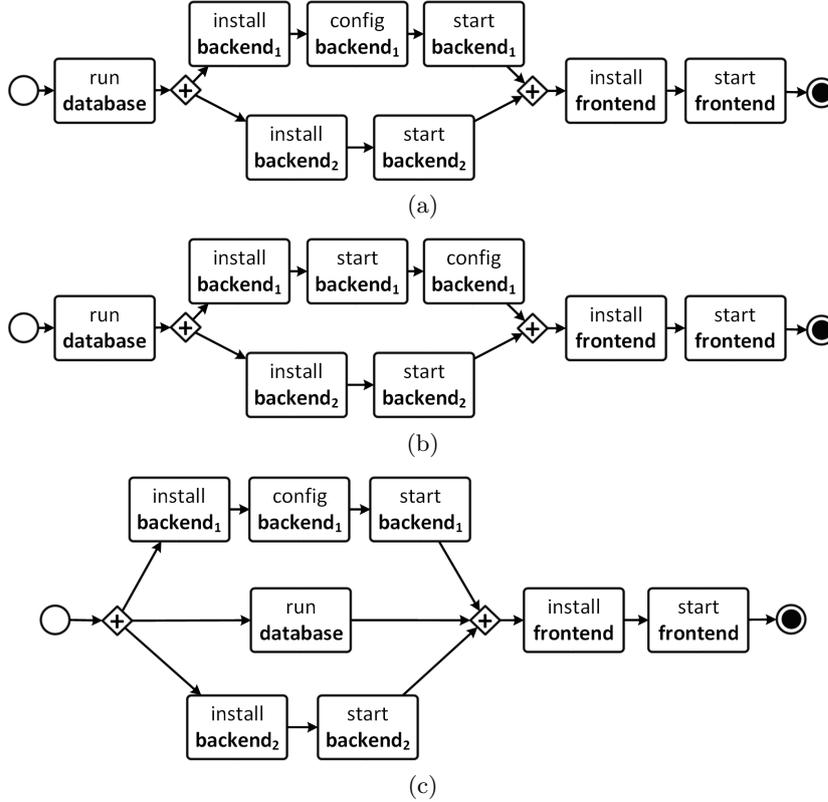to serve their clients, and this in turn causes a fault also in frontend.

**Fig. 2.** Example of deployment plans. Empty circles denote the start of plans, boxes represent the execution of management operations, "plus" operators are used to start and terminate parallel flows, and filled circles denote the end of plans.

Both the above mentioned cases fail because a microservice stops providing its capabilities while other microservices are relying on them to continue to work. In both cases we would like to recover our microservice-based application, by automatically determining a valid reconfiguration plan capable of restoring the desired application configuration.

In summary, while the validity of management plans can be manually verified, this is a time-consuming and error-prone process, especially in microservice-based applications [19]. In order to enable the automated verification of the validity of plans, as well as the automated generation of valid plans reaching desired application configurations, we need an explicit representation of the management protocols of the nodes appearing in the topology of a composite application. Such management protocols have to take into account the possibility of faults to occur, and should permit reacting to them to recover the desired configuration of an application, e.g., by dynamically reconfiguring the inter-connections among the components of an application.

# 3   Fault-aware management protocols

Most of the available languages for modelling composite applications permit describing the states, requirements, capabilities, and management operations of the nodes building the topology of a composite application (e.g., enterprise topology graphs [5], TOSCA [24]). Fault-aware management protocols [7] permit specifying the management behaviour of the nodes composing an application, i.e. the behaviour of a node's management operations, their relations with states, requirements, and capabilities, and how a node reacts to the occurrence of a fault.

However, fault-aware management protocols assume many-to-1 dependencies among the nodes forming a topology (i.e., each requirement of a node is connected to exactly one capability of another node, while each capability of a node can be connected to multiple requirements of other nodes). We hereby present an extension of fault-aware management protocols that relaxes such assumption, by also allowing to specify whether the capability used to satisfy a requirement may be dynamically changed (by choosing another capability connected to it in the application topology).

Let $N$ be a node modelling an application component, whose finite sets of its states, requirements, capabilities, and management operations are denoted by $S_N$, $R_N$, $C_N$, and $O_N$, respectively. Fault-aware management protocols permit modelling the management behaviour of $N$ by describing (i) the order in which the management operations of $N$ can be executed, (ii-iii) whether and how $N$ depends on other nodes providing the capabilities that satisfy its requirements, and (iv) how $N$ reacts when a fault occurs. More precisely:

(i) The order in which the operations of $N$ can be executed is described by a transition relation $\tau_N$ specifying whether an operation $o \in O_N$ can be executed in a state $s \in S_N$, and which state is reached by executing $o$ in $s$.

(ii) The states and transitions of $N$ can be associated with (possibly empty) sets of requirements to indicate that the capabilities satisfying them are assumed to be provided by the corresponding nodes.
   – The requirements associated with a state $s \in S_N$ specify that the capabilities satisfying them must (continue to) be offered by the corresponding nodes in order for $N$ to (continue to) work properly.
   – The requirements associated with a transition $t \in \tau_N$ specify that the capabilities satisfying them must be provided by the corresponding nodes to enable the execution of $t$.

(iii) Each state $s \in S_N$ is also associated with the (possibly empty) set of capabilities provided by $N$ in $s$.

(iv) $N$ is affected by a fault whenever it is in a state assuming some requirement(s) to be satisfied, and some other node stops providing the capabilities satisfying such requirement(s). The explicit fault handling of $N$ is described by a transition relation $\varphi_N$ specifying how $N$ changes its state from $s$ to $s'$ (with $s, s' \in S_N$) when some of the requirements it assumes in $s$ stop being satisfied.

**Definition 1 (Fault-aware management protocols).** *Let* $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ *be a node, where* $S_N$, $R_N$, $C_N$, *and* $O_N$ *are the finite sets of its states, requirements, capabilities, and management operations.* $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$ *is a finite state machine defining the* fault-aware management protocol *of* $N$, *where:*

- $\overline{s}_N \in S_N$ *is the initial state,*
- $\rho_N : S_N \rightarrow 2^{R_N}$ *is a function indicating which requirements must hold in each state* $s \in S_N$,
- $\chi_N : S_N \rightarrow 2^{C_N}$ *is a function indicating which capabilities of* $N$ *are offered in each state* $s \in S_N$,
- $\tau_N \subseteq S_N \times 2^{R_N} \times 2^{R_N} \times O_N \times S_N$ *is a set of quintuples modelling the transition relation, i.e.* $\langle s, P, \Delta, o, s' \rangle \in \tau_N$ *denotes that in state* $s$, *and if the requirements in* $P$ *are satisfied (with those in* $\Delta \subseteq P$ *potentially being satisfied by a different capability after the transition),* $o$ *is executable and leads to state* $s'$, *and*
- $\varphi_N \subseteq S_N \times 2^{R_N} \times S_N$ *is a set of triples modelling the explicit fault handling for a node, i.e.* $\langle s, \Delta, s' \rangle \in \varphi_N$ *denotes that the node will change its state from* $s$ *to* $s'$ *if a subset of the requirements in* $(\rho(s) - \rho(s')) \cup \Delta$ *stops being satisfied (with the requirements in* $\Delta$ *potentially being satisfied by a different capability after the transition)*[1].

*Remark.* Note that we permit specifying which requirements may be satisfied by different capabilities after a transition to enable dynamic reconfiguration (as we will see in Sect. 4). Intuitively, for each requirement $r$ that is assumed to hold in a given application configuration, only one among the capabilities connected to $r$ is actually used to satisfy $r$ (provided that the corresponding node is actually offering such capability). A transition $\langle s, P, \Delta, o, s' \rangle \in \tau_N$ ensures that the capabilities used to satisfy the requirements in $P - \Delta$ continue to be the same after the transition, while the capabilities used to satisfy the requirements in $\Delta$ may be dynamically changed. Similarly, a transition $\langle s, \Delta, s' \rangle \in \varphi_N$ permits dynamically changing the capabilities used to satisfy the requirements in $\Delta$. □

*Example 1.* Fig. 3 shows the management protocols of the nodes composing our motivating scenario (thick arrows represent $\tau$, dashed arrows represent $\varphi$).

Consider, for instance, the management protocol $\mathcal{M}_{\mathsf{frontend}}$, which describes the management behaviour of the microservice frontend. In states uninstalled (initial) and installed it does not require nor provide anything. The same does not hold in the running state, where frontend assumes the requirement endp to (continue to) be satisfied. If the requirement endp is faulted, and there exists a different capability (with respect to that currently bound to endp) that can satisfy it, it is possible to remain in state running by executing the fault-handling transition dynamically changing the actual binding of endp. If instead there is

---

[1] A transition $\langle s, \Delta, s' \rangle \in \varphi_N$ permits handling the fault of a requirement $r$ either by leading to a state $s'$ not assuming $r$ any more (when $r \in \rho(s) - \rho(s')$), or by changing its actual binding (when $r \in \Delta$).
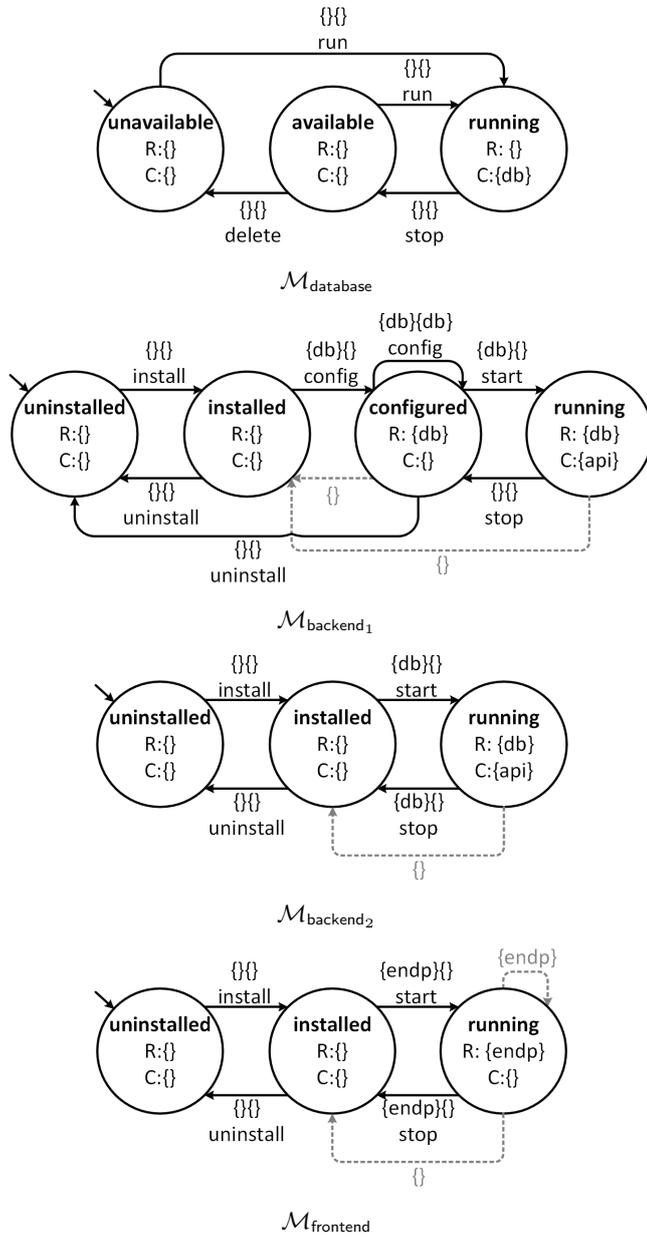
**Fig. 3.** Example of fault-aware management protocols.

no capability capable of satisfying endp, then the microservice frontend goes back to its state installed. Finally, the protocol specifies that operations install, and uninstall do not need any requirement to be performed, while start and stop require endp to be satisfied.

$\mathcal{M}_{\mathsf{backend}_1}$ is another example of management protocol worth looking at. It indeed includes a state offering a capability (viz., $\mathsf{backend}_1$ is offering its capability api only in the state running), and a transition to dynamically reconfigure the actual binding of a requirement (viz., $\mathsf{backend}_1$ can change the actual binding of the requirement db by executing the operation config in state configured). □

In the following, we assume fault-aware management protocols to be well-formed, deterministic, and race-free [6,7]. Also, as the management protocol of a node may leave unspecified how it will behave in case some requirements stop being fulfilled in some states, we assume that management protocols have been automatically completed by adding transitions for all unhandled faults[2].

## 4    Analysing the management of composite applications

In this section we illustrate how to analyse and automate the management of composite applications in a fault-resilient manner, by also taking that the application topology may be dynamically reconfigured while the application is managed. More precisely, we show how to automatically determine the management behaviour of an application by composing the protocols of its nodes according to the application topology (Sect. 4.1). We then describe how this permits automating various useful analyses, like checking the validity of a management plan, which are its effects, or how to automatically determine a valid management plan reaching a desired application configuration (Sect. 4.2).

### 4.1    Management behaviour of an application

We hereby show how to determine the fault-aware management behaviour of an application by composing the fault-aware management protocols of its components. In doing so, we exploit some shorthand notations to denote generic composite applications, the nodes in their topology, and the connections among the requirements and capabilities of such nodes.

**Definition 2 (Composite application).** *We denote with $A = \langle T, B \rangle$ a generic composite application, where $T$ is the finite set of nodes in the application topol-*

---

[2] The procedure to automatically complete management protocols is discussed in [7]. Essentially, they are completed by adding transitions for all unhandled faults, all leading to a "sink" state that requires and provides nothing.

*ogy³, and where inter-node connections are described by a binding relation*

$$B \subseteq R_T \times C_T, \text{ with } R_T = \bigcup_{N \in T} R_N \text{ and } C_T = \bigcup_{N \in T} C_N,$$

*which associates each requirement of each node with the capabilities that can be used to satisfy it.*

Formally, the semantics of the management protocols of a composite application $A = \langle T, B \rangle$ can be defined by a labelled transition system over configurations that denote the states of the nodes in $T$ and the association between each requirement assumed by a node in $T$ and the capability of another node in $T$ that is actually used to satisfy such requirement. Intuitively, $\langle G, b \rangle \xrightarrow{o}_A \langle G', b' \rangle$ is a transition denoting that operation $o$ can be executed (on a node) in $A$ when the "global state" of $A$ is $G$ and the "actual binding" among requirements and capabilities is $b$, making $A$ evolve into a new global state $G'$ with a new actual binding $b'$. We next formally define the notions of *global state* and *actual binding* for a composite application.

The global state $G$ of an application $A$ stores the current state of each of its nodes, while the actual binding $b$ of an application $A$ stores the current association between the requirements assumed by the nodes in $A$ and the capabilities used to satisfy them.

**Definition 3 (Global state).** *Let $A = \langle T, B \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$. A global state $G$ of $A$ is a set of states such that:*

$$G \subseteq \bigcup_{N \in T} S_N \ \wedge \ \forall N \in T. \exists! s \in G : s \in S_N.$$

**Notation 1.** *Let $G$ be a global state of a composite application $A = \langle T, B \rangle$. We denote with $\rho(G)$ the set of requirements that are assumed to hold by the nodes in $T$ when $A$ is in $G$, and with $\chi(G)$ the set of capabilities that are provided by such nodes in $G$. Formally:*

$$\rho(G) = \bigcup_{N \in T} \{ \rho_N(s) \mid s \in G \wedge s \in S_N \}, \text{ and}$$

$$\chi(G) = \bigcup_{N \in T} \{ \chi_N(s) \mid s \in G \wedge s \in S_N \}.$$

An actual binding is a partial function from the requirements $R_T$ to the capabilities $C_T$ of an application $A = \langle T, B \rangle$ which, informally speaking, defines a "subset" of the binding relation $B$.

---

³ For simplicity, and without loss of generality, we assume that the names of states, requirements, capabilities, and operations of a node are all disjoint. We also assume that, given two different nodes in a topology, the names of their states, requirements, capabilities, and operations are disjoint.

**Definition 4 (Actual binding).** *Let $A = \langle T, B \rangle$ be a composite application. An* actual binding $b$ *of $A$ is a partial function from the requirements in $T$ to the capabilities in $T$, which is defined as follows:*

$$b : R_T \to C_T \ \wedge \ \forall r \in R_T : (b(r) = \bot \ \vee \ \langle r, b(r) \rangle \in B).$$

We now define a function $f$ to denote the set of *pending faults* in an application $A$ when its global state is $G$ and when its actual binding is $b$. Intuitively, the faults that are pending in $A$ are the requirements assumed in $G$ that (according to $b$) are bound to capabilities that are not provided in $G$ or not bound at all.

**Definition 5 (Pending faults).** *Let $A = \langle T, B \rangle$ be a composite application. The function $f$ denotes the set of* pending faults *in $A$:*

$$f(G, b) = \{r \in \rho(G) \mid b(r) \notin \chi(G) \ \vee \ b(r) = \bot\}$$

*where $G$ is a global state of $A$, and $b$ is an actual binding of $A$.*

We also define a function $h_N$ to denote the set of *settling handlers* for handling all the faulted requirements of a node $N$ that are pending in $f(G, b)$. Such handlers are all fault handling transitions $\langle s, \Delta, s' \rangle \in \varphi_N$ such that:

− $s \in G$, i.e. $\langle s, \Delta, s' \rangle$ is executable from the current state $s$ of $N$ in $G$,
− $\rho_N(s') - \Delta \subseteq \rho_N(s) - f(G, b)$, i.e. $\langle s, \Delta, s' \rangle$ handles all faulted requirements of $N$ (as its target state $s'$ is not assuming any of the requirements faulted in $f(G, b)$ but those that can be dynamically rebound), and
− $\forall r \in \Delta.\exists c \in \chi(G) : \langle r, c \rangle \in B$, i.e. for each requirement $r$ that can be dynamically rebound by $\langle s, \Delta, s' \rangle$, there exists a capability $c$ that is actually offered in $G$ and that is capable of satisfying $r$.

**Definition 6 (Settling handlers).** *Let $A = \langle T, B \rangle$ be a composite application, and let $G$ and $b$ be a global state and an actual binding of $A$, respectively. Let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ be a node in $T$, with $\mathcal{M}_N = \langle \bar{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$. The function $h_N$ denotes the set of* settling handlers *for the faults pending in $f(G, b)$ and affecting $N$:*

$$\begin{aligned} h_N(G, b) = \{\langle s, \Delta, s' \rangle \in \varphi_N \mid \ & s \in G \ \wedge \\ & \rho_N(s') - \Delta \subseteq \rho_N(s) - f(G, b) \ \wedge \\ & \forall r \in \Delta.\exists c \in \chi(G) : \langle r, c \rangle \in B \ \} \end{aligned}$$

The management behaviour of a composite application $A = \langle T, B \rangle$ is defined by a labelled transition system over pairs denoting the global state and actual binding of $A$. The transition system is characterised by two inference rules, *(op)* for operation execution and *(fault)* for fault propagation. The former permits executing a management operation $o$ on a node $N \in T$ only if there are no pending faults and all the requirements needed by $N$ to perform $o$ are satisfied (by the capabilities provided by other nodes in $T$). The latter defines how to execute settling handlers when there are pending faults.

*Example 2.* Consider again the application in our motivating scenario (Sect. 2), and suppose that it is in the following *global state $G$* (Def. 3):

$$\text{frontend.running} \qquad \text{backend}_1.\text{configured}$$
$$\text{backend}_2.\text{running} \qquad \text{database.running}$$

Suppose also that the *actual binding $b$* of the application (Def. 4) is the following:

$$b(\text{frontend.endp}) = \text{backend}_1.\text{api}$$
$$b(\text{backend}_1.\text{db}) = \text{database.db}$$
$$b(\text{backend}_2.\text{db}) = \text{database.db}$$

In the above situation, the set of *pending faults $f(G, b)$* (Def. 5) is the following:

$$\text{frontend.endp}$$

The set of *settling handlers $h_{\mathsf{frontend}}(G, b)$* (Def. 6) for the above pending fault is the following:

$$\langle \text{frontend.running}, \{\text{frontend.endp}\}, \text{frontend.running} \rangle$$
$$\langle \text{frontend.running}, \{\}, \text{frontend.installed} \rangle$$

$\square$

**Definition 7 (Management behaviour of a composite application).** *Let $A = \langle T, B \rangle$ be a composite application, and let $N = \langle S_N, R_N, C_N, O_N, \mathcal{M}_N \rangle$ with $\mathcal{M}_N = \langle \overline{s}_N, \rho_N, \chi_N, \tau_N, \varphi_N \rangle$. The management behaviour of $A$ is modelled by a labelled transition system whose configurations are pairs $\langle G, b \rangle$, where $G$ is a global state of $A$ and $b$ is an actual binding for $A$, and whose transition relation is defined by the following inference rules:*

$$\frac{
\begin{array}{c}
f(G, b) = \varnothing \quad s \in G \quad \langle s, P, \Delta, o, s' \rangle \in \tau_N \quad G' = G - \{s\} \cup \{s'\} \\
\forall r \in \rho(G) : r \notin \Delta \Rightarrow b'(r) = b(r) \qquad \forall r \in \rho(G') : \langle r, b'(r) \rangle \in B \\
\forall r \in P : b'(r) \in \chi(G) \wedge \langle r, b'(r) \rangle \in B
\end{array}
}{\langle G, b \rangle \xrightarrow{o} \langle G', b' \rangle} \ (op)$$

$$\frac{
\begin{array}{c}
f(G, b) \neq \varnothing \quad s \in G \quad \langle s, \Delta, s' \rangle \in h_N(G, b) \quad G' = G - \{s\} \cup \{s'\} \\
\forall r \in \rho(G) : r \notin \Delta \Rightarrow b'(r) = b(r) \qquad \forall r \in \rho(G') : \langle r, b'(r) \rangle \in B \\
\forall r \in \Delta : b'(r) \in \chi(G) \wedge \langle r, b'(r) \rangle \in B \\
\forall \langle s, \_, s'' \rangle \in h_N(G, b) : \rho(s'') \subseteq \rho(s')
\end{array}
}{\langle G, b \rangle \xrightarrow{\perp} \langle G', b' \rangle} \ (fault)$$

The *(op)* rule defines how the global state of a composite application is updated when a node $N$ performs a transition $\langle s, P, \Delta, o, s' \rangle \in \tau_N$. Such transition can be performed if there are no pending faults (viz., $f(G, b) = \varnothing$), and if there exists a new actual binding $b'$ such that:

- $\forall r \in \rho(G) : r \notin \Delta \Rightarrow b'(r) = b(r)$, i.e. $b'$ preserves the actual binding of the requirements that are assumed in $G$ and that cannot be dynamically changed by the transition,
- $\forall r \in \rho(G') : \langle r, b'(r) \rangle \in B$, i.e. all requirements assumed after the transition are bound to capabilities that can satisfy them (according to $B$), and
- $\forall r \in P : b'(r) \in \chi(G) \land \langle r, b'(r) \rangle \in B$, i.e. all requirements needed to perform the transition are bound to capabilities actually provided in the global state $G$ and that can satisfy them (according to $B$).

As a result, the application configuration $\langle G, b \rangle$ is changed to $\langle G', b' \rangle$, where $G'$ is obtained from $G$ by updating the state of $N$ (viz., $G' = G - \{s\} \cup \{s'\}$), and where $b'$ is a new actual binding satisfying the above explained conditions.

The *(fault)* rule instead models fault propagation. It defines how the global state $G$ of a composite application $A$ is updated when executing a settling handler $\langle s, \Delta, s' \rangle$ for a node $N$. A settling handler $\langle s, \Delta, s' \rangle \in h_N$ can be executed if there are pending faults (viz., $f(G, b) \neq \varnothing$), and if there exists a new actual binding $b'$ satisfying conditions analogous to those imposed by the *(op)* rule. Also, among all settling handlers, $\langle s, \Delta, s' \rangle$ is the handler whose target state $s'$ assumes the biggest set of requirements[4] (viz., $\forall \langle s, \_, s'' \rangle \in h_N(G, b). \rho(s'') \subseteq \rho(s')$). As a result, the application configuration $\langle G, b \rangle$ is changed to $\langle G', b' \rangle$, where $G'$ is obtained from $G$ by updating the state of $N$ (viz., $G' = G - \{s\} \cup \{s'\}$), and where $b'$ is a new actual binding satisfying the above explained conditions.

## 4.2 Analysing the management behaviour of applications

The management behaviour defined in Def. 7 permits analysing and automating the management of a composite application. For instance, we can easily determine which sequences (or, more in general, which workflows) of management operations can be considered valid in a given application configuration. To simplify definitions, we introduce some shorthand notations to observe only the (transitions corresponding to) management operations executed while managing a composite application.

**Notation 2.** *Let $A = \langle T, B \rangle$ be a composite application. The observable behaviour of $A$ is denoted by a labelled transition system whose configurations are pairs $\langle G, b \rangle$ (where $G$ is a global state of $A$ and $b$ is an actual binding for $A$), and whose transition relation is defined by the following inference rules:*

$$\frac{\langle G, b \rangle \xrightarrow{o} \langle G', b' \rangle}{\langle G, b \rangle \xmapsto{o} \langle G', b' \rangle} \qquad \frac{\langle G, b \rangle \xrightarrow{\perp} \langle G', b' \rangle \quad \langle G', b' \rangle \xmapsto{o} \langle G'', b'' \rangle}{\langle G, b \rangle \xmapsto{o} \langle G'', b'' \rangle}$$

Intuitively, a sequence of operations $o_1 o_2 \ldots o_n$ is *valid* in a given application configuration $\langle G, b \rangle$ if (i) $o_1$ is executable in $\langle G, b \rangle$, and (ii) whatever is the application configuration $\langle G', b' \rangle$ reached by executing $o_1$ in $\langle G, b \rangle$, the sequence

---

[4] In this way, fault-handling transitions are guaranteed (to handle all the faults on a node and) to minimise the amount of requirements that stop being assumed.

$o_2 \ldots o_n$ can always be performed in $\langle G', b' \rangle$. Validity of plans follows from that of their sequential traces.

**Definition 8 (Valid plan).** *Let $A = \langle T, B \rangle$ be a composite application. Let also $G$ and $b$ be a global state and an actual binding of $A$, respectively. A sequence of management operations $o_1 o_2 \ldots o_n$ is* valid *in $\langle G, b \rangle$ iff it is empty or*

*(i) $\exists \langle G', b' \rangle : \langle G, b \rangle \overset{o_1}{\longmapsto} \langle G', b' \rangle$, and*
*(ii) $\forall \langle G', b' \rangle : \langle G, b \rangle \overset{o_1}{\longmapsto} \langle G', b' \rangle \Rightarrow o_2 \ldots o_n$ is valid in $\langle G', b' \rangle$.*

*A workflow $W$ orchestrating the management operations in $A$ is a* valid plan *in $\langle G, b \rangle$ iff all the sequential traces of $W$ are valid in $\langle G, b \rangle$.*

The introduced modelling can be exploited for various other purposes besides checking plan validity. For instance, since different sequential traces of a valid plan may lead to different global states, it is interesting to characterise *deterministic* plans.

**Definition 9 (Deterministic plan).** *Let $A = \langle T, B \rangle$ be a composite application. Let also $G$ and $b$ be a global state and an actual binding of $A$, respectively. A sequence of management operations $o_1 \ldots o_n$ in $\langle G, b \rangle$ is* deterministic *iff*

$$ o_1 \ldots o_n \text{ is valid in } \langle G, b \rangle \ \wedge \ \exists! G'.\langle G, b \rangle \overset{o_1}{\longmapsto} \ldots \overset{o_n}{\longmapsto} \langle G', \_ \rangle. $$

*A workflow $W$ orchestrating the management operations in $A$ is a* deterministic plan *in $\langle G, b \rangle$ iff all its sequential traces are deterministic in $\langle G, b \rangle$ and reach the same global state $G'$.*

*Example 3.* One can readily check that plan (a) in Fig. 2 is valid and deterministic. Indeed, all its sequential traces are valid in the initial application configuration (i.e., when database is unavailable, $backend_1$, $backend_2$ and frontend are uninstalled, and the actual binding is empty), and they all bring the application to the global state where all microservices are running. Note that the above holds independently from the fact that the requirement endp of frontend will be bound to the capability api offered by $backend_1$ or to that offered by $backend_2$.

Plan (b) is instead not valid in the initial application configuration. This is because all sequential traces starting with database.run · $backend_1$.install · $backend_1$.start are not valid (because the management protocol of $backend_1$ does not allow to execute start before config — see $\mathcal{M}_{backend_2}$ in Fig. 3).

Plan (c) is not valid either in the initial application configuration, because all the sequential traces starting with $backend_1$.install · $backend_1$.config are not valid. This is because the management protocol of $backend_1$ constrains the executability of config to the satisfaction the requirement endp of $backend_2$ (see $\mathcal{M}_{backend_2}$ in Fig. 3). The latter can only be satisfied by the capability db of database, which is however not provided by database in its starting state unavailable (see $\mathcal{M}_{database}$ in Fig. 3). □

Checking whether a given plan is valid or deterministic corresponds to visiting the graph generated by the transition system of an application's management

behaviour (Def. 7), which also models the non-determinism due to the choice of the new actual binding $b'$. It is worth noting that, thanks to the constraints on management protocols and to the way they are combined, both the possible global states and the possible actual bindings are *finite*, hence guaranteeing that the above mentioned visit of the graph eventually terminates.

It is also worth noting that the effects of a deterministic workflow $W$ on the states of an application's components, as well as on the requirements and capabilities that are available, can be directly determined from the global state reached by performing any of the sequential traces of $W$.

Moreover, the problem of *finding* whether there is a deterministic plan that starts from an application configuration and achieves a specific goal (e.g., bringing some components of an application to specific states, or making some capabilities available) can also be solved with a visit of the graph associated with the transition system of an application's management behaviour. This is especially useful to automatically determine plans dynamically reconfiguring the topology of an application, or to restore a desired application configuration after some application components got stuck because of a fault[5] (as we discussed in our motivating scenario — Sect. 2).

Finally, it is worth characterising a weaker notion of validity, to denote those plans whose sequential traces may fail depending on the actual bindings chosen while executing them. Intuitively, a sequence of operations is weakly valid if *there exists* an application configuration $\langle G', b' \rangle$ reached by executing $o_1$ in $\langle G, b \rangle$ such that $o_2 \ldots o_n$ can all be performed. Weak validity of plans follows from that of their sequential traces.

**Definition 10 (Weakly valid plan).** *Let $A = \langle T, B \rangle$ be a composite application. Let also $G$ and $b$ be a global state and an actual binding of $A$, respectively. The sequence $o_1 o_2 \ldots o_n$ of management operations in $A$ is* weakly valid *in $\langle G, b \rangle$ iff it is empty or*

$$\exists \langle G', b' \rangle : \langle G, b \rangle \xmapsto{o_1} \langle G', b' \rangle \ \wedge \ o_2 \ldots o_n \ \textit{is weakly valid in } \langle G', b' \rangle$$

*A workflow $W$ orchestrating the management operations in $A$ is a* weakly valid *plan in $\langle G, b \rangle$ iff one of its sequential traces is weakly valid in $\langle G, b \rangle$.*

A weakly valid plan warns that it may fail. By observing its sequential traces, it is possible to understand whether such a warning can be ignored when deploying and managing a concrete instance of the application (e.g., since a problematic actual binding will never be chosen because of security policies), or whether they can cause real issues at runtime. Notably, such issues can be avoided by

---

[5] In [7] we illustrate how to recover application that are stuck because a fault was not properly handled, or because of unforeseen faults (e.g., due to non-deterministic application bugs). The approach is based on the idea of automatically determining valid plans restoring a desired application configuration, and (despite [7] assumes many-to-1 dependencies among application components) it can be easily adapted to cope with the notion of validity presented in this paper.

exploiting the above explained planning techniques to find a deterministic plan (if any) reaching the same global state reached by the successful traces of a weakly valid plan.

## 5   Related work

The problem of automating composite application management is one of the major trends in today's IT [20]. Management protocols [6,8], as well as Aeolus [12], permit automatically deploying and managing multi-component cloud applications. The underlying idea of both approaches is quite simple: Developers describe the behaviour of their components through finite-state machines, and such descriptions can be composed to model the management behaviour of a composite application. Engage [14] is another approach for processing application descriptions to automatically deploy applications. Fault-aware management protocols [7] extend management protocols [6], and differ from Aeolus [12] and Engage [14], since they permit explicitly modelling possible faults of components, as well as how to react when such faults occurs.

However, the fault-aware management protocols in [7] assume many-to-1 dependencies among the nodes forming a topology. The approach presented in this paper properly extends fault-aware management protocols by relaxing this assumption, and by also allowing to specify whether the capability used to satisfy a requirement may be dynamically changed (by choosing among the many available and connected to it in the application topology).

Other approaches worth mentioning are Rapide [22], Darwin [23] and $\pi$-ADL [25]. Rapide, Darwin and $\pi$-ADL are very close in the spirit to our approach, as they are languages for describing composite systems, whose components expose *require* and *provide* ports (corresponding to our notions of requirements and capabilities, respectively). The structure of a system is given by interconnecting such ports, and it can dynamically be changed while execution progresses. Rapide, Darwin and $\pi$-ADL however differ from our approach as they only permit many-to-1 interconnections among components, since they assume application components to be stateless (viz., once instantiated, a component assumes all its *require* ports to be satisfied, and it offers all its *provide* ports), and since they do not permit explicitly specifying how to handle the faults affecting a component (when one of the requirements it assumes stop being satisfied).

The rigorous engineering of fault-tolerant systems is a well-known problem in computer science [9], with many existing approaches targeting the design and analysis of such systems. For instance, [17] proposes a way to design object-oriented systems by starting from fault-free systems, and by subsequently refining such design by handling different types of faults. [26] and [3] instead focus on fault-localisation, thus permitting to redesign a system to avoid the occurrence of such a fault. These approaches differ from ours because they aim at obtaining applications that "never fail", since all potential faults have been identified and properly handled. Our approach is instead more recovery-oriented [10], since

we focus on applications where faults possibly occur, and we permit designing applications capable of being recovered.

Similar considerations apply to [16], [18], and [1], which however share with our approach the basic idea of modelling faults in single components and of composing the obtained models according to the dependencies between such components (i.e., according to the application topology).

[13] proposes a decentralised approach to deploy and reconfigure cloud applications in presence of failures. It models a composite application as a set of interconnected virtual machines, each equipped with a configurator managing its instantiation and destruction. The deployment and reconfiguration of the whole application is then orchestrated by a manager interacting with virtual machine configurators. [13] shares with our approach the objective of providing a decentralised and fault-aware management of a composite application, by specifying the management of each component separately. However, it differs from our approach since it focuses on recovering virtual machines that have been terminated only because of environment faults, while we also permit describing how components react to application-specific faults.

[21] proposes an approach to identify failures in a system whose components' behaviour is described by finite state machines. Even though the analyses are quite different, the modelling in [21] is quite similar to ours. It indeed relies on a sort of requirements and capabilities to model the interaction among components, and it permits "implicitly" modelling how components behave in presence of single/multiple faults. Our modelling is a strict generalisation of that in [21], since a component's state can evolve not only because of requirement unsatisfaction but also because of invoked operations, and since we permit "explicitly" handling faults (i.e., fault handling transitions are distinct from those modelling the normal behaviour of a component). Similar considerations apply to [11], whose modelling is also based on finite state machines with input and output channels (which permit fault communication and propagation by components).

In summary, to the best of our knowledge, the approach we propose in this paper is the first that permits automatically orchestrating the management of composite applications (i) accounting for many-to-many dependencies among application components, (ii) allowing to dynamically change/reconfigure application topologies, and (iii) assuming that faults possibly occur while managing composite applications.

## 6 Conclusions

Fault-aware management protocols [7] are a modular and reusable way to model the management behaviour of application components (including how they react to potential faults), and to analyse and automate the management of a complex application composed by multiple components. However, the fault-aware management protocols in [7] assume many-to-1 dependencies among the nodes in an application topology (viz., a capability can be used to satisfy multiple requirements, while a requirement can be satisfied only by a given capability), and this

does not permit dealing with composite applications having multiple nodes that can satisfy a given requirement of another node (such as the microservice-based application in our motivating scenario — Sect. 2).

In this paper we have presented an extension of fault-aware management protocols, which permits modelling and analysing composite applications whose topologies have many-to-many dependencies among components. The proposed extension is such that any capability connected to a requirement can be used to satisfy such requirement when the latter is needed by the corresponding node. The proposed extension also permits specifying whether the capability used to satisfy a requirement of a node can be dynamically changed (when executing a management operation or handling a fault), hence allowing to dynamically reconfigure the topology of an application.

The proposed extension of fault-aware management protocols paves the way for their exploitation for modelling and analysing elasticity and live updates of the components forming a composite application. Both cases require to add/remove replicas of nodes, as well as of their ingoing and outgoing dependencies, to/from the topology of a composite application, hence requiring to support many-to-many dependencies and dynamic reconfiguration. The proposed extension of fault-aware management protocols includes such support, and it can hence be exploited to model the evolution of the states of node replicas. The formalisation of the above is in the scope of our immediate future work.

We believe that the proposed modelling and analysing techniques can also pave the way towards the development of self-adaptive composite applications. Self-adaptive applications are controlled by the so-called MAPE (*Monitor*, *Analyse*, *Plan* and *Execute*) loop [27]. Indeed, our techniques can be exploited during the *Analyse* and *Plan* steps of the MAPE loop controlling a composite application. The *Monitor* and *Execute* instead require to adapt and integrate existing approaches to work with fault-aware management protocols. Such adaptation and integration are left for future work.

It is also worth noting that, even if some of the analyses we presented in Sect. 4 have exponential time complexity in the worst case, they still constitute a significant improvement with respect to state-of-the-art, as currently the management of composite applications is coordinated manually (e.g., by developing ad-hoc scripts), and it is hardly reusable since it is tightly coupled to the application. We plan to further improve the support for the analyses in Sect. 4, by offering a tool capable of validating management plans and of automatically determining valid plans reaching desired goals (by extending the prototype[6] for the fault-aware management protocols of [7]).

Finally, it is worth noting that fault-aware management protocols do not take into account costs nor QoS, since their focus is on automatically coordinating the management of the components forming a composite application. Cost and QoS are however important factors [2], and they should be taken into account when modelling and analysing the management of composite applications. This could be solved by further extending management protocols by allowing to specify the

---

[6] `https://github.com/di-unipi-socc/barrel`.

amount of resources actually offered by a capability, and "how many" of such resources are needed to satisfy a requirement connected to such capability. The extension of fault-aware management protocols to include cost and QoS is in the scope of our future work.

## References

1. Alhosban, A., Hashmi, K., Malik, Z., Medjahed, B., Benbernou, S.: Bottom-up fault management in service-based systems. ACM Trans. Internet Technol. 15(2), 7:1–7:40 (2015)
2. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., Zaharia, M.: A view of cloud computing. Commun. ACM 53(4), 50–58 (2010)
3. Betin Can, A., Bultan, T., Lindvall, M., Lux, B., Topp, S.: Eliminating synchronization faults in air traffic control software via design for verification with concurrency controllers. Automated Software Engineering 14(2), 129–178 (2007)
4. Binz, T., Breitenbücher, U., Kopp, O., Leymann, F.: TOSCA: Portable Automated Deployment and Management of Cloud Applications, pp. 527–549. Springer, New York, NY (2014)
5. Binz, T., Fehling, C., Leymann, F., Nowak, A., Schumm, D.: Formalizing the Cloud through Enterprise Topology Graphs. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on. pp. 742–749. IEEE (2012)
6. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: Dustdar, S., Leymann, F., Villari, M. (eds.) Service Oriented and Cloud Computing: 4th European Conference, ESOCC 2015, Proceedings. Lecture Notes in Computer Science, vol. 9306, pp. 19–33. Springer International Publishing (2015)
7. Brogi, A., Canciani, A., Soldani, J.: Fault-aware application management protocols. In: Aiello, M., Johnsen, B.E., Dustdar, S., Georgievski, I. (eds.) Service-Oriented and Cloud Computing: 5th IFIP WG 2.14 European Conference, ESOCC 2016, Vienna, Austria, September 5-7, 2016, Proceedings. pp. 219–234. Springer International Publishing (2016)
8. Brogi, A., Canciani, A., Soldani, J., Wang, P.: A petri net-based approach to model and analyze the management of cloud applications. In: Koutny, M., Desel, J., Kleijn, J. (eds.) Transactions on Petri Nets and Other Models of Concurrency XI. pp. 28–48. LNCS, Springer Berlin Heidelberg, Berlin, Heidelberg (2016)
9. Butler, M., Jones, C., Romanovsky, A., Troubitsyna, E.: Rigorous Development of Complex Fault-Tolerant Systems. LNCS, Springer (2007)
10. Candea, G., Brown, A.B., Fox, A., Patterson, D.: Recovery-oriented computing: Building multitier dependability. Computer 37(11), 60–67 (2004)
11. Chen, L., Jiao, J., Fan, J.: Fault propagation formal modeling based on stateflow. In: Proc. of the 1st ICRSE. pp. 1–7. IEEE (2015)
12. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: a Component Model for the Cloud. Information and Computation pp. 100–121 (2014)
13. Durán, F., Salaün, G.: Robust and reliable reconfiguration of cloud applications. Journal of Systems and Software 122, 524 – 537 (2016)
14. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: A deployment management system. In: Proc. of the 33rd PLDI. pp. 263–274. ACM (2012)

15. Fowler, M., Lewis, J.: Microservices. ThoughtWorks, `https://www.thoughtworks.com/insights/blog/microservices-nutshell` (2016)
16. Grunske, L., Kaiser, B., Papadopoulos, Y.: Model-driven safety evaluation with state-event-based component failure annotations. In: Proc. of the 8th CBSE. pp. 33–48. Springer (2005)
17. Johnsen, E., Owe, O., Munthe-Kaas, E., Vain, J.: Incremental fault-tolerant design in an object-oriented setting. In: Proc. of 2nd APAQS. pp. 223–230 (2001)
18. Kaiser, B., Liggesmeyer, P., Mäckel, O.: A new component concept for fault trees. In: Proc. of the 8th SCS. pp. 37–46. Australian Comp. Soc., Inc. (2003)
19. Killalea, T.: The hidden dividends of microservices. Commun. ACM 59(8), 42–45 (2016)
20. Leymann, F.: Cloud computing. it - Information Technology 53(4), 163–164 (2011)
21. Liggesmeyer, P., Rothfelder, M.: Improving system reliability with automatic fault tree generation. In: Proc. of the 28th FTCS. pp. 90–99. IEEE (1998)
22. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Softw. Eng. 21(4), 336–355 (1995)
23. Magee, J., Kramer, J.: Dynamic structure in software architectures. SIGSOFT Softw. Eng. Notes 21(6), 3–14 (1996)
24. OASIS: Topology and Orchestration Specification for Cloud Applications. `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.pdf` (2013)
25. Oquendo, F.: $\pi$-adl: An architecture description language based on the higher-order typed $\pi$-calculus for specifying dynamic and mobile software architectures. SIGSOFT Softw. Eng. Notes 29(3), 1–14 (2004)
26. Qiang, W., Yan, L., Bliudze, S., Xiaoguang, M.: Automatic fault localization for BIP. In: Li, X., Liu, Z., Yi, W. (eds.) Proc. of the 1st SETTA. pp. 277–283. Springer (2015)
27. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 14:1–14:42 (2009)