A Formal Approach to Microservice Architecture Deployment

A Formal Approach to Microservice Architecture Deployment*

Mario Bravetti, Saverio Giallorenzo, Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro

Abstract Following previous work on the automated deployment of component-based applications, we present a formal model specifically tailored for reasoning on the deployment of microservice architectures. The first result that we present is a formal proof of decidability of the problem of synthesizing optimal deployment plans for microservice architectures, a problem which was proved to be undecidable for generic component-based applications. Then, given that such proof translates the deployment problem into a constraint satisfaction problem, we present the implementation of a tool that, by exploiting state-of-the-art constraint solvers, can be used to actually synthesize optimal deployment plans. We evaluate the applicability of our tool on a realistic microservice architecture taken from the literature.

1 Introduction

Inspired by service-oriented computing, microservices structure software applications as highly modular and scalable compositions of fine-grained and loosely-

Mario Bravetti
Università di Bologna, Italy, e-mail: mario.bravetti@unibo.it

Saverio Giallorenzo
University of Southern Denmark, Denmark, e-mail: saverio@sdu.dk

Jacopo Mauro
University of Southern Denmark, Denmark, e-mail: mauro@sdu.dk

Iacopo Talevi
Università di Bologna, Italy, e-mail: iacopo.talevi@studio.unibo.it

Gianluigi Zavattaro
Università di Bologna, Italy, e-mail: gianluigi.zavattaro@unibo.it

* Research partly supported by the H2020-MSCA-RISE project ID 778233 “Behavioural Application Program Interfaces (BEHAPI)”.

coupled services [29]. These features support modern software engineering practices, like continuous delivery/deployment [40] and application autoscaling [3]. A relevant problem in these practices consists of the automated deployment of the microservice application, i.e., the distribution of the fine-grained components over the available computing nodes, and its dynamic modification to cope, e.g., with positive or negative peaks of user requests.

In this paper, we address the problem of planning the deployment, and redeployment, of microservice architectures in a formal manner, by presenting an approach for modeling microservice architectures, that allows us to both prove formal properties and realize an implemented solution. We follow the approach taken by the *Aeolus component model* [23, 25, 26], which was used to formally define the problem of deploying component-based software systems and to prove that, in the general case, such problem is undecidable [26]. The basic idea of Aeolus is to enrich the specification of components with a finite state automaton that describes their deployment life cycle. Previous work identified decidable fragments of the Aeolus model: removing from Aeolus replication constraints, used e.g., to specify a minimal amount of services connected to a load balancer, makes the deployment problem decidable, but non-primitive recursive [25]; removing also conflicts, used e.g., to express the impossibility to deploy in the same system two types of components, makes the problem PSpace-complete [44] or even poly-time [26], but under the assumption that every required component can be (re)deployed from scratch.

In a recent paper [15], we adapted the Aeolus model to formally reason on the deployment of microservices. To achieve our goal, we significantly revisited the formalization of the deployment problem, replacing Aeolus components with a model of microservices. The main difference between our model of microservices and Aeolus components lies in the specification of their deployment life cycle. Instead of using the full power of finite state automata, like in Aeolus and other TOSCA-compliant deployment models [20], we assume microservices to have two states: (i) creation and (ii) binding/unbinding. Concerning creation, we use *strong* dependencies to express which microservices must be immediately connected to newly created ones. After creation, we use *weak* dependencies to indicate additional microservices that can be bound/unbound. The principle that guided this modification comes from state-of-the-art microservice deployment technologies like Docker [45] and Kubernetes [39]. In particular, the weak and strong dependencies have been inspired by Docker Compose [27], a language for defining multi-container Docker applications, where it is possible to specify different relationships among microservices using, e.g., the depends_on (resp. external_links) modalities that force (resp. do not force) a specific startup order similarly to our strong (resp. weak) dependencies. Weak dependencies are also useful to model horizontal scaling, e.g., a load balancer that is bound to/unbound from many microservice instances during its life cycle.

In addition, w.r.t. the Aeolus model, we also consider resource/cost-aware deployments, taking inspiration from the memory and CPU resources found in Kubernetes. Microservice specifications are enriched with the amount of resources they need to run. In a deployment, a system of microservices runs within a set of computation *nodes*. Nodes represent computational units, e.g., virtual machines in an

Infrastructure-as-a-Service Cloud deployment. Each node has a cost and a set of resources available to the microservices it hosts.

On the model above, it is possible to define the *optimal deployment problem* as follows: given an initial microservice system, a set of available nodes, and a new target microservice to be deployed, find a sequence of reconfiguration actions that, once applied to the initial system, leads to a new deployment that includes the target microservice. Such a deployment is expected to be *optimal*, meaning that the total cost, i.e., the sum of the costs, of the nodes used is minimal. This problem was proved to be decidable [15] by presenting an algorithm working in three phases: (1) generate a set of constraints whose solution indicates the microservices to be deployed and their distribution over the nodes; (2) generate another set of constraints whose solution indicates the connections to be established; (3) synthesize the corresponding deployment plan. The set of constraints includes optimization metrics that minimize the overall cost of the computed deployment.

The algorithm has NEXPTIME complexity because, in the worst-case, the length of the deployment plan could be exponential in the size of the input. However, since in practice the number of microservices deployable on one node is limited by the available resources, if each node can host at most a polynomial amount of microservices the deployment problem is NP-complete and the problem of deploying a system minimizing its total cost is an NP-optimization problem. Moreover, having reduced the deployment problem in terms of constraints, it is possible to exploit state-of-the-art constraint solvers [22, 34, 35], that are frequently used in practice to cope with NP-hard problems. In particular, we investigate the possibility to actually solve the deployment problem for microservices by exploiting Zephyrus2 [1], a configurator optimizer that was originally envisaged for the Aeolus model [24] but later extended and improved to support a new specification language and the possibility to have preferences on the metrics to optimize, e.g., minimize not only the cost but also the number of microservices. We have selected and customized Zephyrus2 because it can easily support the solution of the optimization problems to which we reduce the optimal deployment problem for microservices.

We have evaluated the actual exploitability of our implemented solution by computing the initial optimal deployment, and some possible reconfigurations, for a real-world microservice architecture, inspired by the reference email processing pipeline from Iron.io [33]. That architecture is modeled in the Abstract Behavioral Specification (ABS) language, a high-level object-oriented language that supports deployment modeling [41]. Our technique is then used to compute two types of deployments: an initial one, with one instance for each microservice, and a set of deployments to horizontally scale the system depending on small, medium or large increments in the number of emails to be processed. The experimental results are encouraging in that we were able to compute deployment plans that add more than 30 new microservice instances, assuming availability of hundreds of machines of three different types, and guaranteeing optimality.

Structure of the chapter. In Section 2 we formally study the microservice deployment problem. In Section 3 we discuss Zephyrus2, the tool used to solve such a problem, while in Section 4 we report the experimental results obtained by applying

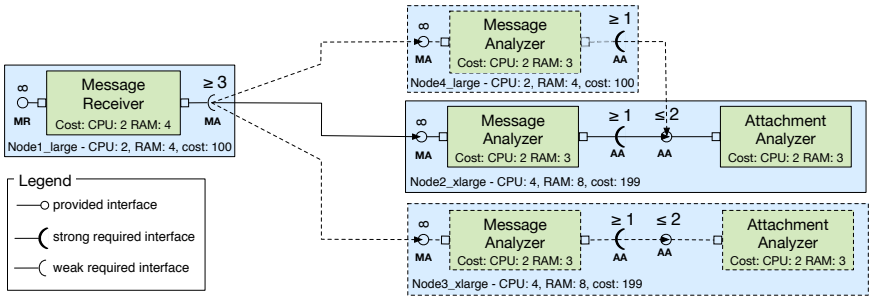

Fig. 1 Example of microservice deployment (blue boxes: nodes; green boxes: microservices; continuous lines: the initial configuration; dashed lines: full configuration).

it to a real-world case-study. Finally, Section 5 discusses related work and draws some concluding remarks.

Note that this chapter mainly reports and extends results published in [15] with an additional section, namely Section 3, to provide more details on the Zephyrus2 tool and the extensions we implemented.

2 The microservice optimal deployment problem

In this section we present our model for representing microservice systems and their deployment. We start from an informal presentation of the model and then we move to define microservice deployment configurations, reconfiguration plans and the optimal deployment problem, providing its decidability proof and an analysis of its complexity.

2.1 Representing microservice systems and their deployment

We model microservice systems as aggregations of components with ports. Each port instantiates either a provided or a required interface. Interfaces describe offered and required functionalities. Microservices are connected by means of bindings indicating which port provides the functionality required by another port. As discussed in Section 1, we consider two kinds of requirements: strong required interfaces, that need to be already fulfilled when the microservice is created, and weak required interfaces, that must be fulfilled at the end of a deployment, or reconfiguration, plan. Microservices are enriched with the specification of the resources they need to properly run. Such resources are provided to the microservices by nodes. Nodes can be seen as the unit of computation executing the tasks associated to each microservice.

As an example, in Figure 1 we report the representation of the deployment of a microservice system where a Message Receiver microservice handles inbound requests, passing them to a Message Analyzer that checks the email content and sends the attachments for inspection to an Attachment Analyzer. The Message Receiver has a port with a *weak* required interface that can be fulfilled by Message Analyzer instances. This requirement is weak, meaning that the Message Receiver can be initially deployed without any connection to instances of Message Analyzer. These connections can be established afterwards and reflect the possibility to horizontally scale the application by adding/removing instances of Message Analyzer. This last microservice has instead a port with a *strong* required interface that can be fulfilled by Attachment Analyzer instances. This requirement is strong to reflect the need to immediately connect a Message Analyzer to its Attachment Analyzer.

Figure 1 presents a reconfiguration that, starting from the initial deployment depicted in continuous lines, adds the elements depicted with dashed lines. Namely, a couple of new instances of Message Analyzer and a new instance of Attachment Analyzer are deployed. This is done in order to satisfy numerical constraints associated to both required and provided interfaces. For required interfaces, the numerical constraints indicate lower bounds to the outgoing bindings, while for provided interfaces they specify upper bounds to the incoming connections. Notice that the constraint ≥ 3 associated to the weak required interface of Message Receiver is not initially satisfied; this is not problematic because constraints on weak interfaces are relevant only at the end of a reconfiguration. In the final deployment, such a constraint is satisfied thanks to the two new instances of Message Analyzer. These two instances need to be immediately connected to an Attachment Analyzer: only one of them can use the initially available Attachment Analyzer, because of the constraint ≤ 2 associated to the corresponding provided interface. Hence, a new instance of Attachment Analyzer is added.

We also model resources: each microservice has associated resources that it consumes, see the CPU and RAM quantities associated to the microservices in Figure 1. Resources are provided by nodes, that we represent as containers for the microservice instances, providing them the resources they require. Notice that nodes have also costs: the total cost of a deployment is the sum of the costs of the used nodes, e.g., in the example the total cost is 598 cents per hour, corresponding to the cost of 4 nodes: 2 C4 large and 2 C4 xlarge virtual machine instances of the Amazon public Cloud.

2.2 Microservices, nodes and deployment configurations

We now move to formal definitions. Here we will introduce microservices (including their required/provided/conflicting interfaces and consumed resources), nodes, and deployment configurations.

We start from the definition of types of microservices, e.g., Attachment Analyzer, Message Receiver, and Message Analyzer in the example of Figure 1, which can

be instantiated when deploying microservice systems. In the following, we assume $\mathcal{I}$ to denote the set of all possible interfaces and $\mathcal{R}$ to be a finite set of kinds of resources. Moreover, we use $\mathbb{N}$ to denote natural numbers, $\mathbb{N}^+$ for $\mathbb{N} \setminus \{0\}$, and $\mathbb{N}^+_\infty$ for $\mathbb{N}^+ \cup \{\infty\}$.

Definition 1 (Microservice type) The set Γ of *microservice types*, ranged over by $\mathcal{T}_1, \mathcal{T}_2, \ldots$, contains 5-ples $\langle P, D_s, D_w, C, R\rangle$ where:

- $P = (\mathcal{I} \nrightarrow \mathbb{N}^+_\infty)$ are the provided interfaces, defined as a partial function from interfaces to corresponding numerical constraints, indicating the maximum number of connected microservices;
- $D_s = (\mathcal{I} \nrightarrow \mathbb{N}^+)$ are the *strong* required interfaces, defined as a partial function from interfaces to corresponding numerical constraints, indicating the minimum number of connected microservices;
- $D_w = (\mathcal{I} \nrightarrow \mathbb{N})$ are the *weak* required interfaces, defined as the strong ones, with the difference that also the constraint 0 can be used indicating that it is not strictly necessary to connect microservices;
- $C \subseteq \mathcal{I}$ are the conflicting interfaces;
- $R = (\mathcal{R} \to \mathbb{N})$ specifies resource consumption, defined as a total function from resources to corresponding quantities indicating the amount of required resources.

We assume sets $\mathtt{dom}(D_s)$, $\mathtt{dom}(D_w)$ and C to be pairwise disjoint.[2]

Notation: In the remainder of the paper, we denote the name of a microservice interface with the upper-case acronym of the name of its microservice, e.g., the interface of the Message Analyzer is denoted MA.

Given a microservice type $\mathcal{T} = \langle P, D_s, D_w, C, R\rangle$, we use the following postfix projections .prov, .req$_\mathtt{S}$, .req$_\mathtt{W}$, .conf, and .res to decompose it:

- .prov returns the partial function associating arities to provided interfaces, e.g., in Figure 1, Message Receiver.prov(MR) = ∞;
- .req$_\mathtt{S}$ returns the partial function associating arities to strong required interfaces, e.g., in Figure 1, Message Analyzer.req$_\mathtt{S}$(AA) = 1;
- .req$_\mathtt{W}$ returns the partial function associating arities to weak required interfaces, e.g., in Figure 1, Message Receiver.req$_\mathtt{W}$(MA) = 3;
- .conf returns the conflicting interfaces;
- .res returns the total function from resources to their required quantities, e.g., in Figure 1, Message Receiver.res(RAM) = 4.

When the numerical constraints are not explicitly indicated, we assume as default value ∞ for provided interfaces, i.e., they can satisfy an unlimited amount of ports requiring the same interface, and 1 for required interfaces, i.e., one connection with a port providing the same interface is sufficient.

Inspired by [25], we allow a microservice to specify a conflicting interface that, intuitively, forbids the deployment of other microservices providing the same interface. Conflicting interfaces can be used to express conflicts among microservices,

[2] Given a partial function f, we use $\mathtt{dom}(f)$ to denote the domain of f, i.e., the set $\{e \mid \exists e' : (e, e') \in f\}$.

preventing both of them to be present at the same time, or cases in which only one microservice instance can be deployed, e.g., a consistent and available microservice that can not be replicated.

Definition 2 (Nodes) The set $\mathcal{N}$ of *nodes* is ranged over by $o_1, o_2, \ldots$ We assume the following information to be associated to each node o in $\mathcal{N}$.

- A function $R = (\mathcal{R} \to \mathbb{N})$ that specifies node *resource availability*: we use $o.\texttt{res}$ to denote such a function.
- A value in $\mathbb{N}$ that specifies node *cost*: we use $o.\texttt{cost}$ to denote such a value.

As example, in Figure 1, the node Node1_large is such that Node1_large.$\texttt{res}$(RAM) = 4 and Node1_large.$\texttt{cost}$ = 100.

Notice that, both in Definition 1 and 2, we use the same symbol R to denote the *resource* function: in the former case it quantifies resources consumed by microservice instances, in the latter it quantifies resources made available by nodes.

We now define configurations that describe systems composed of microservice instances and bindings that interconnect them. We use $\mathcal{Z}$ to denote the set of all possible microservice instances. A configuration, ranged over by $C_1, C_2, \ldots$, is given by a set of deployed microservice instances, with their associated type and node hosting them, and a set of bindings. Formally:

Definition 3 (Configuration) A *configuration* C is a 4-ple $\langle Z, T, N, B \rangle$ where:

- $Z \subseteq \mathcal{Z}$ is the set of the currently deployed *microservices*;
- $T = (Z \to \mathcal{T})$ are the *microservice types*, defined as a function from deployed microservices to microservice types;
- $N = (Z \to \mathcal{N})$ are the *microservice nodes*, defined as a function from deployed microservices to nodes that host them;
- $B \subseteq \mathcal{I} \times Z \times Z$ is the set of *bindings*, namely 3-ples composed of an interface, the microservice that requires that interface, and the microservice that provides it; we assume that, for $(p, z_1, z_2) \in B$, the two microservices z_1 and z_2 are distinct and $p \in (\texttt{dom}(T(z_1).\texttt{req}_\texttt{S}) \cup \texttt{dom}(T(z_1).\texttt{req}_\texttt{W})) \cap \texttt{dom}(T(z_2).\texttt{prov})$.

In our example, we have the binding (MA, $\mathsf{inst}_{\mathsf{mr}}$, $\mathsf{inst}_{\mathsf{ma}}$) where $\mathsf{inst}_{\mathsf{mr}}$ and $\mathsf{inst}_{\mathsf{ma}}$ are the two initial instances in continuous lines of Message Receiver and Message Analyzer type, respectively. Notice that the interface MA satisfies the inclusion constraint at the end of Definition 3 in that MA is a required interface of the Message Receiver type, while it is a provided interface of the Message Analyzer type. Moreover, concerning the microservice placement function N, we have $N(\mathsf{inst}_{\mathsf{mr}})$ = Node1_large and $N(\mathsf{inst}_{\mathsf{ma}})$ = Node2_xlarge.

2.3 Microservice deployment plans

We are now ready to formalize the notion of a microservice deployment plan, which represents a sequence of deployment configurations, with the aim of reaching a final configuration as in the example of Figure 1, by means of *reconfiguration actions*.

The configurations traversed during a microservice deployment plan must satisfy a correctness constraint related to the intended meaning of strong and weak required interfaces and conflicts (see Definition 1). We first define *provisional correctness*, considering only constraints on strong required and provided interfaces, and then we define a general notion of configuration correctness, considering also weak required interfaces and conflicts. The former is intended for transient configurations traversed during the execution of a sequence of reconfigurations, while the latter is intended for the final configuration.

Definition 4 (Provisionally correct configuration) A configuration $C = \langle Z, T, N, B \rangle$ is *provisionally correct* if, for each node $o \in \mathtt{ran}(N)$, it holds[3]

$$\forall r \in \mathcal{R}.\ \ o.\mathtt{res}(r) \geq \sum_{z \in Z, N(z)=o} T(z).\mathtt{res}(r)$$

and, for each microservice $z \in Z$, both following conditions hold:

- $(p \mapsto n) \in T(z).\mathtt{req_S}$ implies that there exist n distinct microservices $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that, for every $1 \leq i \leq n$, we have $\langle p, z, z_i \rangle \in B$;
- $(p \mapsto n) \in T(z).\mathtt{prov}$ implies that there exist no m distinct microservices $z_1, \ldots, z_m \in Z \setminus \{z\}$, with $m > n$, such that, for every $1 \leq i \leq m$, we have $\langle p, z_i, z \rangle \in B$.

In the above definition, the initial inequality guarantees that the amount of resources provided by the nodes are sufficient to satisfy the requests of all the hosted microservices. The first item means that the strong requirements of all components are all satisfied because there are at least as many bindings on those ports as the associated lower bounds. The second item, on the other hand, guarantees that there are no extra connections on provided interfaces, because all the ports exposing a provided interface have no more bindings than the associated upper bound.

Definition 5 (Correct configuration) A configuration $C = \langle Z, T, N, B \rangle$ is *correct* if C is provisionally correct and, for each microservice $z \in Z$, both following conditions hold:

- $(p \mapsto n) \in T(z).\mathtt{req_W}$ implies that there exist n distinct microservices $z_1, \ldots, z_n \in Z \setminus \{z\}$ such that, for every $1 \leq i \leq n$, we have $\langle p, z, z_i \rangle \in B$;
- $p \in T(z).\mathtt{conf}$ implies that, for each $z' \in Z \setminus \{z\}$, we have $p \notin \mathtt{dom}(T(z').\mathtt{prov})$.

In the definition above, besides the guarantees already given by Definion 4, we have that also weak requirements are satisfied (first item), as well as conflicts (second item): i.e., if an instantiated microservice has a conflict on an interface, such an interface cannot be provided by any other microservice in the configuration.

Notice that, in the example in Figure 1, the initial configuration in continuous lines is only provisionally correct in that the weak required interface MA, with arity 3,

[3] Given a (partial) function f, we use $\mathtt{ran}(f)$ to denote the range of f, i.e., the function image set $\{f(e) \mid e \in \mathtt{dom}(f)\}$.

of the Message Receiver is not satisfied, because there is only one outgoing binding. The full configuration — including also the elements in dotted lines — is instead correct: all the constraints associated to the interfaces are satisfied.

We now formalize how configurations evolve by means of atomic actions: we have *bind*/*unbind* actions to create/destroy bindings on ports with weak required interfaces; *new* to instantiate a new microservice instance and the necessary bindings on ports with strong required interface; and *del* to destroy a microservice and, implicitly, its bindings.

Definition 6 (Actions) The set $\mathcal{A}$ contains the following actions:

- $bind(p, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$, with $z_1 \neq z_2$, and $p \in \mathcal{I}$: add a binding between z_1 and z_2 on interface p, which is supposed to be a weak required interface of z_1 and a provide interface of z_2;
- $unbind(p, z_1, z_2)$ where $z_1, z_2 \in \mathcal{Z}$, with $z_1 \neq z_2$, and $p \in \mathcal{I}$: remove the specified binding on p, which is supposed to be a weak required interface of z_1 and a provide interface of z_2;
- $new(z, \mathcal{T}, o, B_s)$ where $z \in \mathcal{Z}$, $\mathcal{T} \in \Gamma$, $o \in \mathcal{N}$ and $B_s = (\mathtt{dom}(\mathcal{T}.\mathtt{req_S}) \to 2^{\mathcal{Z}-\{z\}})$; with B_s representing bindings from strong required interfaces in $\mathcal{T}$ to sets of microservices, being such that, for each $p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_S})$, it holds $|B_s(p)| \geq \mathcal{T}.\mathtt{req_S}(p)$: add a new microservice z of type $\mathcal{T}$ hosted in o and bind each of its strong required interfaces to a set of microservices as described by B_s;[4]
- $del(z)$ where $z \in \mathcal{Z}$: remove the microservice z from the configuration and all bindings involving it.

In our example, assuming that the initially available Attachment Analyzer is named $\mathsf{inst_{aa}}$, we have that the action to create the initial instance of Message Analyzer is $new(\mathsf{inst_{ma}}, \text{Message Analyzer}, \text{Node2\_xlarge}, (\text{AA} \mapsto \{\mathsf{inst_{aa}}\}))$. Notice that it is necessary to establish the binding with the Attachment Analyzer because of the corresponding strong required interface.

The execution of actions can now be formalized using a labeled transition system on configurations, which uses actions as labels.

Definition 7 (Reconfigurations) Reconfigurations are denoted by transitions $C \xrightarrow{\alpha} C'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration C produces a new configuration C'. The transitions from a configuration $C = \langle Z, T, N, B \rangle$ are defined as follows:

[4] Given sets S and S' we use: 2^S to denote the power set of S, i.e., the set $\{S' \mid S' \subseteq S\}$; $S - S'$ to denote set difference; and $|S|$ to denote the cardinality of S.

$C \xrightarrow{bind(p,z_1,z_2)} \langle Z, T, N, B \cup \langle p, z_1, z_2 \rangle \rangle$
if $\langle p, z_1, z_2 \rangle \notin B$ and
$p \in \mathtt{dom}(T(z_1).\mathtt{req_W}) \cap \mathtt{dom}(T(z_2).\mathtt{prov})$

$C \xrightarrow{unbind(p,z_1,z_2)} \langle Z, T, N, B \setminus \langle p, z_1, z_2 \rangle \rangle$
if $\langle p, z_1, z_2 \rangle \in B$ and
$p \in \mathtt{dom}(T(z_1).\mathtt{req_W}) \cap \mathtt{dom}(T(z_2).\mathtt{prov})$

$C \xrightarrow{new(z,\mathcal{T},o,B_s)} \langle Z \cup \{z\}, T', N', B' \rangle$
if $z \notin Z$ and
$\forall p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_S}).\ \forall z' \in B_s(p).$
$p \in \mathtt{dom}(T(z').\mathtt{prov})$ and
$T' = T \cup \{(z \mapsto \mathcal{T})\}$ and
$N' = N \cup \{(z \mapsto o)\}$ and
$B' = B \cup \{\langle p, z, z' \rangle \mid z' \in B_s(p)\}$

$C \xrightarrow{del(z)} \langle Z \setminus \{z\}, T', N', B' \rangle$
if $T' = \{(z' \mapsto \mathcal{T}) \in T \mid z \neq z'\}$ and
$N' = \{(z' \mapsto o) \in N \mid z \neq z'\}$ and
$B' = \{\langle p, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\}\}$

A *deployment plan* is simply a sequence of actions that transform a provisionally correct configuration without violating provisional correctness along the way and, finally, reach a correct configuration.

Definition 8 (Deployment plan) A *deployment plan* P from a provisionally correct configuration C_0 is a sequence of actions $\alpha_1, \ldots, \alpha_m$ such that:

- there exist $C_1, \ldots, C_m$ provisionally correct configurations, with $C_{i-1} \xrightarrow{\alpha_i} C_i$ for $1 \leq i \leq m$, and
- C_m is a correct configuration.

Deployment plans are also denoted with $C_0 \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} C_m$.

In our example, a deployment plan that reconfigures the initial provisionally correct configuration into the final correct one is as follows: a *new* action to create the new instance of Attachment Analyzer, followed by two *new* actions for the new Message Analyzers[5] and finally two *bind* actions to connect the Message Receiver to the two new instances of Message Analyzer.

Notice that, since in deployment plans the requirements associated with strong interfaces must be satisfied immediately after each reconfiguration action, which must yield a provisionally correct configuration, it is possible to deploy a configuration with circular dependencies *only if at least one weak required interface is involved in the cycle*. In fact, having a cycle with only strong required interfaces would require to deploy all the microservices involved in the cycle simultaneously. We now formalize a well-formedness condition on microservice types to guarantee the absence of such configurations.

Definition 9 (Well-formed Universe) Given a finite set of microservice types U, that we also call *universe*, the strong dependency graph of U is as follows: $G(U) = (U, V)$ with $V = \{(\mathcal{T}, \mathcal{T}') \mid \mathcal{T}, \mathcal{T}' \in U \wedge \exists p \in \mathcal{I}.p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_S}) \cap \mathtt{dom}(\mathcal{T}'.\mathtt{prov})\}$. The universe U is well-formed if $G(U)$ is acyclic.

In the following, we always assume universes to be well-formed. Well-formedness does not prevent the specification of microservice systems with circular dependencies, which are captured by cycles with at least one weak required interface.

[5] Notice that the connection between the Message Analyzers and the corresponding Attachment Analyzers is part of these *new* actions.

2.4 Microservice optimal deployment problem

We now have all the ingredients to define the *optimal deployment problem*, that is our main concern: given a universe of microservice types, a set of available nodes and an initial configuration, we want to know whether and how it is possible to deploy at least one microservice of a given microservice type $\mathcal{T}$ by optimizing the overall cost of nodes hosting the deployed microservices.

Definition 10 (Optimal deployment problem) The *optimal deployment problem* has, as input, a finite well-formed universe U of microservice types, a finite set of available nodes O, an initial provisionally correct configuration C_0 and a microservice type $\mathcal{T}_t \in U$. The output is:

- A **deployment plan** $\mathsf{P} = C_0 \xrightarrow{\alpha_1} C_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} C_m$ such that
  - for all $C_i = \langle Z_i, T_i, N_i, B_i \rangle$, with $1 \leq i \leq m$, it holds $\forall z \in Z_i.\ T_i(z) \in U \wedge N_i(z) \in O$, and
  - $C_m = \langle Z_m, T_m, N_m, B_m \rangle$ satisfies $\exists z \in Z_m : T_i(z) = \mathcal{T}_t$;

  if there exists one. In particular, among all deployment plans satisfying the constraints above, one that minimizes $\sum_{o \in O.(\exists z. N_m(z)=o)} o.\mathtt{cost}$, i.e., the overall cost of nodes in the last configuration C_m, is outputted.
- **no** (stating that no such plan exists); otherwise.

In the remainder of this section we present an algorithm for solving the optimal deployment problem. This will allow us to complete the section by stating our main result on the decidability of such a problem.

We assume that the input to the problem to be solved is given by U (the microservice types), O (the set of available nodes), C_0 (the initial provisionally correct configuration), and $\mathcal{T}_t \in U$ (the target microservice type). We use $\mathcal{I}(U)$ to denote the set of interfaces used in the considered microservice types, namely $\mathcal{I}(U) = \bigcup_{\mathcal{T} \in U} \mathtt{dom}(\mathcal{T}.\mathtt{req_S}) \cup \mathtt{dom}(\mathcal{T}.\mathtt{req_W}) \cup \mathtt{dom}(\mathcal{T}.\mathtt{prov}) \cup \mathcal{T}.\mathtt{conf}$.

The algorithm is based on three phases.

Phase 1 The first phase consists of the generation of a set of constraints that, once solved, indicates how many instances should be created for each microservice type $\mathcal{T}$ (denoted with $\mathtt{inst}(\mathcal{T})$), and how many of them should be deployed on node o (denoted with $\mathtt{inst}(\mathcal{T}, o)$). We denote with $\mathtt{bind}(p, \mathcal{T}, \mathcal{T}')$ the number of bindings that should be established for each interface p from instances of type $\mathcal{T}$ — considering both weak and strong required interfaces — to instances of type $\mathcal{T}'$. We also generate an optimization function that guarantees that the generated configuration is minimal w.r.t. its total cost.

We now incrementally report the generated constraints. The first group of constraints deals with the number of bindings:

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U,\, p\in\mathsf{dom}(\mathcal{T}.\mathsf{req_S})} \mathcal{T}.\mathsf{req_S}(p)\cdot \mathtt{inst}(\mathcal{T}) \leq \sum_{\mathcal{T}'\in U} \mathtt{bind}(p,\mathcal{T},\mathcal{T}') \tag{1a}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U,\, p\in\mathsf{dom}(\mathcal{T}.\mathsf{req_W})} \mathcal{T}.\mathsf{req_W}(p)\cdot \mathtt{inst}(\mathcal{T}) \leq \sum_{\mathcal{T}'\in U} \mathtt{bind}(p,\mathcal{T},\mathcal{T}') \tag{1b}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U,\, \mathcal{T}.\mathsf{prov}(p)<\infty} \mathcal{T}.\mathsf{prov}(p)\cdot \mathtt{inst}(\mathcal{T}) \geq \sum_{\mathcal{T}'\in U} \mathtt{bind}(p,\mathcal{T}',\mathcal{T}) \tag{1c}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U,\, \mathcal{T}.\mathsf{prov}(p)=\infty} \mathtt{inst}(\mathcal{T}) = 0 \;\Rightarrow\; \sum_{\mathcal{T}'\in U} \mathtt{bind}(p,\mathcal{T}',\mathcal{T}) = 0 \tag{1d}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U,\, p\notin\mathsf{dom}(\mathcal{T}.\mathsf{prov})} \sum_{\mathcal{T}'\in U} \mathtt{bind}(p,\mathcal{T}',\mathcal{T}) = 0 \tag{1e}$$

Constraints (1a) and (1b) guarantee that there are enough bindings to satisfy all the required interfaces, considering both strong and weak requirements. Symmetrically, constraint (1c) guarantees that the number of bindings is not greater than the total available capacity, computed as the sum of the single capacities of each provided interface. In case the capacity is unbounded (i.e., ∞), it is sufficient to have at least one instance that activates such port to support any possible requirement, see constraint (1d). Finally, constraint (1e) guarantees that no binding is established connected to provided interfaces of microservice types that are not deployed.

The second group of constraints deals with the number of instances of microservices to be deployed.

$$\mathtt{inst}(\mathcal{T}_t) \geq 1 \tag{2a}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\substack{\mathcal{T}\in U,\\ p\in\mathcal{T}.\mathsf{conf}}} \bigwedge_{\substack{\mathcal{T}'\in U-\{\mathcal{T}\},\\ p\in\mathsf{dom}(\mathcal{T}'.\mathsf{prov})}} \mathtt{inst}(\mathcal{T}) > 0 \;\Rightarrow\; \mathtt{inst}(\mathcal{T}') = 0 \tag{2b}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\substack{\mathcal{T}\in U,\, p\in\mathcal{T}.\mathsf{conf}\,\wedge\\ p\in\mathsf{dom}(\mathcal{T}.\mathsf{prov})}} \mathtt{inst}(\mathcal{T}) \leq 1 \tag{2c}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U} \bigwedge_{\mathcal{T}'\in U-\{\mathcal{T}\}} \mathtt{bind}(p,\mathcal{T},\mathcal{T}') \leq \mathtt{inst}(\mathcal{T})\cdot\mathtt{inst}(\mathcal{T}') \tag{2d}$$

$$\bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{\mathcal{T}\in U} \mathtt{bind}(p,\mathcal{T},\mathcal{T}) \leq \mathtt{inst}(\mathcal{T})\cdot(\mathtt{inst}(\mathcal{T})-1) \tag{2e}$$

The first constraint (2a) guarantees the presence of at least one instance of the target microservice. Constraint (2b) guarantees that no two instances of different types will be created if one activates a conflict on an interface provided by the other one. Constraint (2c), consider the other case in which a type activates the same interface both in conflicting and provided modality: in this case, at most one instance of such type can be created. Finally, constraints (2d) and (2e) guarantee that there are enough pairs of distinct instances to establish all the necessary bindings. Two distinct constraints are used: the first one deals with bindings between microservices of two different types, the second one with bindings between microservices of the same type.

The last group of constraints deals with the distribution of microservice instances over the available nodes O.

$$\texttt{inst}(\mathcal{T}) = \sum_{o \in O} \texttt{inst}(\mathcal{T}, o) \tag{3a}$$

$$\bigwedge_{r \in \mathcal{R}} \bigwedge_{o \in O} \sum_{\mathcal{T} \in U} \texttt{inst}(\mathcal{T}, o) \cdot \mathcal{T}.\texttt{res}(r) \leq o.\texttt{res}(r) \tag{3b}$$

$$\bigwedge_{o \in O} \Big(\sum_{\mathcal{T} \in U} \texttt{inst}(\mathcal{T}, o) > 0 \Big) \Leftrightarrow \texttt{used}(o) \tag{3c}$$

$$\min \sum_{o \in O,\, \texttt{used}(o)} o.\texttt{cost} \tag{3d}$$

Constraint (3a) simply formalizes the relationship among the variables $\texttt{inst}(\mathcal{T})$ and $\texttt{inst}(\mathcal{T}, o)$: the total amount of all instances of a microservice type, should correspond to the sum of the instances locally deployed on each node. Constraint (3b) checks that each node has enough resources to satisfy the requirements of all the hosted microservices. The last two constraints define the optimization function used to minimize the total cost: constraint (3c) introduces the boolean variable $\texttt{used}(o)$ which is true if and only if node o contains at least one microservice instance; constraint (3d) is the function to be minimized, i.e., the sum of the costs of the used nodes.

All the constraints of *Phase 1*, and the optimization function, are expected to be given in input to a constraint/optimization solver. If a solution is not foundit is not possible to deploy the required microservice system; otherwise, the next phases of the algorithm are executed to synthesize the optimal deployment plan.

Phase 2 The second phase consists of the generation of another set of constraints that, once solved, indicates the bindings to be established between any pair of microservices to be deployed. More precisely, for each type $\mathcal{T}$ such that $\texttt{inst}(\mathcal{T}) > 0$, we use $s_i^{\mathcal{T}}$, with $1 \leq i \leq \texttt{inst}(\mathcal{T})$, to identify the microservices of type $\mathcal{T}$ to be deployed. We also assume a function N that associates microservices to available nodes O, which is compliant with the values $\texttt{inst}(\mathcal{T}, o)$ already computed in *Phase 1*, i.e., given a type $\mathcal{T}$ and a node o, the number of $s_i^{\mathcal{T}}$, with $1 \leq i \leq \texttt{inst}(\mathcal{T})$, such that $N(s_i^{\mathcal{T}}) = o$ coincides with $\texttt{inst}(\mathcal{T}, o)$.

In the constraints below we use the variable $\texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$, with $i \neq j$, if $\mathcal{T} = \mathcal{T}'$: its value is 1 if there is a connection between the required interface p of $s_i^{\mathcal{T}}$ and the provided interface p of $s_j^{\mathcal{T}'}$, 0 otherwise. We use n and m to denote $\texttt{inst}(\mathcal{T})$ and $\texttt{inst}(\mathcal{T}')$, respectively, and an auxiliary total function $\mathit{limProv}(\mathcal{T}', p)$ that extends $\mathcal{T}'.\texttt{prov}$ associating 0 to interfaces outside its domain.

$$\bigwedge_{\mathcal{T}\in U}\ \bigwedge_{p\in\mathcal{I}(U)}\ \bigwedge_{i\in 1\ldots n}\ \sum_{j\in(1\ldots m)\setminus\{i\mid\mathcal{T}=\mathcal{T}'\}} \mathsf{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \leq limProv(\mathcal{T}', p) \tag{4a}$$

$$\bigwedge_{\mathcal{T}\in U}\ \bigwedge_{p\in\mathsf{dom}(\mathcal{T}.\mathtt{req_S})}\ \bigwedge_{i\in 1\ldots n}\ \sum_{j\in(1\ldots m)\setminus\{i\mid\mathcal{T}=\mathcal{T}'\}} \mathsf{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \geq \mathcal{T}.\mathtt{req_S}(p) \tag{4b}$$

$$\bigwedge_{\mathcal{T}\in U}\ \bigwedge_{p\in\mathsf{dom}(\mathcal{T}.\mathtt{req_W})}\ \bigwedge_{i\in 1\ldots n}\ \sum_{j\in(1\ldots m)\setminus\{i\mid\mathcal{T}=\mathcal{T}'\}} \mathsf{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \geq \mathcal{T}.\mathtt{req_W}(p) \tag{4c}$$

$$\bigwedge_{\mathcal{T}\in U}\ \bigwedge_{p\notin\mathsf{dom}(\mathcal{T}.\mathtt{req_S})\cup\mathsf{dom}(\mathcal{T}.\mathtt{req_W})}\ \bigwedge_{i\in 1\ldots n}\ \sum_{j\in(1\ldots m)\setminus\{i\mid\mathcal{T}=\mathcal{T}'\}} \mathsf{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) = 0 \tag{4d}$$

Constraint (4a) considers the provided interface capacities to fix upper bounds to the bindings to be established, while constraints (4b) and (4c) fix lower bounds based on the required interface capacities, considering both the weak (constraint (4b)) and the strong (constraint (4c)) ones. Finally, constraint (4d) indicates that it is not possible to establish connections on interfaces that are not required.

A solution for these constraints exists because, as also shown in [23], the constraints (1a) to (2e), already solved during *Phase 1*, guarantee that the configuration to be synthesized contains enough capacity on the provided interfaces to satisfy all the required interfaces.

Phase 3 In this last phase we synthesize the deployment plan that, when applied to the initial configuration C_0, reaches a new configuration C_t with nodes, microservices and bindings as computed in the first two phases of the algorithm. Without loss of generality, in this decidability proof we show the existence of a simple plan that first removes the elements in the initial configuration and then deploys the target configuration from scratch. However, as also discussed in Section 4, in practice it is possible to define more complex planning mechanisms that re-use microservices already deployed.

Reaching an empty configuration is a trivial task since it is always possible to perform in the initial configuration unbind actions for all the bindings connected to weak required interfaces. Then, the microservices can be safely deleted. Thanks to the well-formedness assumption (Definition 9) and using a topological sort, it is possible to order the microservices to be removed without violating any strong required interface, e.g., first remove the microservice not requiring anything and repeat until all the microservices have been deleted.

The deployment of the target configuration follows a similar pattern. Given the distribution of microservices over nodes –computed in *Phase 1*– and the corresponding bindings –computed in *Phase 2*–, the microservices can be created by following a topological sort considering the microservices dependencies following from the strong required interfaces. When all the microservices are deployed on the corresponding nodes, the remaining bindings, on weak required ports, may be added in any possible order.

Given the existence of the above algorithm for solving the optimal deployment problem, we can now formally state our main result.

Theorem 1 *The optimal deployment problem is decidable.*

From the complexity point of view, it is possible to show that the decision versions of the optimization problem solved in *Phase 1* is NP-complete, in *Phase 2* is in NP, while the planning in *Phase 3* is synthesized in polynomial time. Unfortunately, due to the fact that numeric constraints can be represented in log space, the output of *Phase 2* requiring the enumeration of all the microservices to deploy can be exponential in the size of the output of *Phase 1*, indicating only the total number of instances for each type. For this reason, the optimal deployment problem is in NEXPTIME. However, we would like to note that this applies only when an exponential number of microservices is required to be installed in a node. In practice, this does not happen since every node provides some resources that are enough to deploy only a small number of microservices. If at most a polynomial number of microservices can be deployed on each node, we have that the optimal deployment problem becomes an NP-optimization problem and its decision version is NP-complete. See the technical report [16] for the formal proofs of complexity.

3 Zephyrus

In this section we describe the Zephyrus2 tool and how it can be used to actually solve the optimal deployment problem as formalized in the previous section. Zephyrus2 is a configurator optimizer that was originally envisaged for the Aeolus model [24] but later extended and improved to support a new specification language and the possibility to have preferences on the metrics to optimize, e.g., minimize not only the cost but, for instance, also the number of microservices [1].

Zephyrus2 in particular can be used to solve the optimization problems of the first two phases described before, namely the distribution of the microservices on the nodes, and the instantiation of the bindings between the different microservices.

3.1 Optimal distribution of microservices

Differently from what formally described before, for usability sake, Zephyrus2 allows a far richer way of defining what are the deployment constraints of the users. Indeed, while in the previous section the goal was to deploy at least a given microservice (see constraint (2a)),[6] Zephyrus2 natively supports a richer language powerful enough to express, e.g., the presence of a given number of microservices and their co-installation requirements or conflicts. For example, the user might require the presence of at least one `Message Receiver` and 3 `Message Analyzer` and that, for

[6] Note that despite this formal limitation, the possibility to install one microservice is enough to encode far more elaborate constraints. Indeed, by using the strong requirements, it is possible to create for example a dummy target microservice that forces other microservices to be present in a certain amount.

fault tolerance reasons, no two `Message Analyzer` instances should be installed on the same node.

For microservice and nodes specifications, Zephyrus2 supports the JavaScript Object Notation (JSON) format.[7] As an example, the following JSON snippet defines the `Message Receiver` microservice in Figure 1.

```
"MessageReceiver": {
  "resources": { "CPU": 2, "RAM": 4 },
  "requires": { "MA": 3 },
  "provides": [ { "ports": [ "MR" ], "num": -1 } ]
}
```

In the first line the name of the microservice is defined. Microservice names allow for the usage of only letters, numbers, the underscore character, and they should start with a letter. For this reason, here and in the following examples, in the Zephyrus2 snippets we will rename the services removing the trailing spaces (e.g., `Message Receiver` becomes `MessageReceiver`).
In the second line, with the keyword `resources`, it is declared that `Message Receiver` consumes 2 vCPUs and 4 units of RAM. The keyword `requires` defines that the microservice has a requirement on interface `MA` with a capacity constraint "≥ 3". Similarly, the `provides` keyword declares that the microservice provides the interface `MR` to a possibly unbounded number of microservices, represented by -1. Note that here, Zephyrus2 does not distinguish between strong and weak requirements since this notion becomes relevant only later, namely, in *Phase 2*.

The definition of nodes is also done in JSON. For instance, the JSON input to define 10 `xlarge` Amazon virtual machines is the following.

```
"xlarge": {
  "num": 10,
  "resources": { "CPU": 4, "RAM": 8 },
  "cost": 199
}
```

For specifying the target configuration, Zephyrus2 introduces a new specification language for expressing the deployment constraints to allows DevOps teams to express more complex cloud- and application-specific constraints.

As shown in Figure 2 that reports the grammar of the specification language defined using the ANTLR tool [5], a deployment constraint is a logical combination of comparisons between arithmetic expressions. Besides integers, expressions may refer to microservice names representing the total number of deployed instances of a microservice. Location instances are identified by a location name followed by the instance index, starting at zero, in square brackets. A microservice name prefixed by a node stays for the number of microservice instances deployed on the given node.

For example, the following formula requires the presence of at least one `Message Receiver` on the second `large` node, and exactly 3 `Message Analyzer` in the entire system.

[7] The formal JSON Schema of Zephyrus2 input is available at [43]. JSON was used since it is one of the most common data formats for information exchange, thus easing a possible support of external tools and standards.

```
b_expr : b_term (bool_binary_op b_term )* ;
b_term : ('not')? b_factor ;
b_factor : 'true' | relation ;
relation : expr (comparison_op expr)? ;
expr : term (arith_binary_op term)* ;
term : INT                                                    |
  ('exists' | 'forall') VARIABLE 'in' type ':' b_expr        |
  'sum' VARIABLE 'in' type ':' expr                          |
  (( ID | VARIABLE | ID '[' INT ']' ) '.')? microservice     |
  arith_unary_op expr                                        |
  '(' b_expr ')'                                             ;
microservice :  ID | VARIABLE ;
type : 'components' | 'locations' | RE ;
bool_binary_op : 'and' | 'or' | 'impl' | 'iff' ;
arith_binary_op : '+' | '-' | '*' ;
comparison_op : '<=' | '=' | '>=' | '<' | '>' | '!=' ;
preferences: ('cost' | expr ) ( ';' 'cost' | expr )*
VARIABLE : '?'[a-zA-Z_][a-zA-Z0-9_]*;
ID : [a-zA-Z_][a-zA-Z0-9_]* ;
INT : [0-9]+ ;
```

Fig. 2 User desiderata specification language grammar.

```
large[1].MessageReceiver > 0 and MessageAnalyzer = 3
```

For quantification and for building sum expressions, Zephyrus2 use identifiers prefixed with a question mark as variables. Quantification and sum constructs can range over microservices –when the `'components'` keyword is used–, nodes, –when the `'locations'` keyword is used–, or over microservices/nodes whose names match a given regular expression (`RE`). Using such constraints, it is possible to express more elaborate properties such as the co-location or distribution of microservices, or limit the amount of microservices deployed on a given location. For example, the constraint

```
forall ?x in locations: ( ?x.MessageReceiver > 0 impl
  ?x.MessageAnalyzer = 0)
```

states that the presence of an instance of a `Message Receiver` deployed on any node `x` implies that no `Message Analyzer` can be deployed on the same node. As another example, requiring the `Message Receiver` to be installed alone on a virtual machine can be done by requiring that if a `Message Receiver` is installed on a given node then the sum of the microservices installed on that node should be exactly 1. This can be done by stating the following constraint.

```
forall ?x in locations: ( ?x.MessageReceiver > 0 impl
  (sum ?y in components: ?x.?y) = 1 )
```

For defining the optimization metrics, Zephyrus2 extends what has been formally presented in the previous section by allowing the user to express her preferences over valid configurations in the form of a list of arithmetic expressions whose values

should be minimized in the given priority order (see `preferences` in Line 17 of Table 2). While in the formalization in Section 2 the metric to optimize was only the cost, Zephyrus2 solves instead a multi optimization problem taking into account different metrics. For example, since the keyword `cost` (line 17 of Figure 2) can be used to require the minimization of the total cost of the used nodes, the following listing specifies in the Zephyrus2 syntax, the metric to minimize first the total cost of the application and then the total number of microservices.

```
cost; ( sum ?x in components: ?x )
```

This is also the default metric used if the user does not specify her own preferences.

3.2 Bindings optimization

As described in Section 2, the second phase of the approach consists of the instantiation of the bindings among the microservices. In particular, the constraints (4a) to (4d) enforce the satisfaction of the capacity constraints of the interfaces. However, in a real application, a user often has preferences on how microservices are connected. For instance, usually public clouds are composed by different data centers available in different regions, and load balancers deployed in a region are connected only with the back-end services deployed on the same region.

To capture this kind of preferences, one can easily enrich the constraints (4a) to (4d) with new metrics to optimize. For example, to maximize the local bindings (i.e., give a preference to the connections among microservices hosted in the same node) the following metric can be added.

$$\min \sum_{\mathcal{T},\mathcal{T}' \in U, i \in 1 \ldots \mathsf{inst}(\mathcal{T}), j \in 1 \ldots \mathsf{inst}(\mathcal{T}'), p \in \mathcal{I}(U), N(s_i^{\mathcal{T}}) \neq N(s_j^{\mathcal{T}'})} \mathsf{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$$

Another example, used in the case study discussed in Section 4, is the following metric that maximizes the number of bindings:[8]

$$\max \sum_{s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}, p \in \mathcal{I}(U)} \mathsf{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$$

Zephyrus2 supports the possibility to specify these binding preferences. The grammar to express a preference is defined in Figure 3. A preference may be either the string `local` or an arithmetic expression (Line 1). The `local` preference is used to maximize the number of bindings among the microservices deployed in the same node. Arithmetic expressions are used instead to capture more advanced preferences. These expressions are built by using as basic atoms integers (Line 2) and the predicate

[8] We model a load balancer as a microservice having a weak required interface, with arity 0, that can be provided by its back-end service. By adopting the above maximization metric, the synthesized configuration connects all possible services to such required interface, thus allowing the load balancer to forward requests to all of them.

```
1  preference: 'local' | expr ;
2  term : INT                          |
3   'bind' '(' VARIABLE ',' VARIABLE ',' var_or_port ')'  |
4   ('exists' | 'forall') VARIABLE ('of' 'type' RE)?
5     'in' typeV ':' b_expr            |
6   'sum' VARIABLE ('of' 'type' RE)?
7     'in' typeV ':' expr              |
8   '(' b_expr ')'                     ;
9  microservice :  ID | ID '[' ID ']' | ID '[' RE ']' ;
10 typeV : 'ports' | 'locations' | RE ;
11 var_or_port : ID | VARIABLE ;
```

Fig. 3 Grammar to express binding preferences (missing non terminals are as defined in Figure 2).

`bind(?x,?y,z)`, which is assumed to be evaluated to 1 if the microservice referenced by the variable x is connected with the microservice y using interface z, 0 otherwise. Notice that in this case z can be a concrete interface name or an interface variable. In order to instantiate the variables of the term `bind`, quantifiers (Line 4-8) and sum expressions (Line 6-7) may be used.

As an example, assume that we have two kinds of node: those available in region A and those available in region B. The first nodes can be distinguished from the second ones thanks to their name. Node names from region A end with `'_A'` while the other node names end with `'_B'`. If we would like a `Message Analyzer` deployed in region A to be connected with all the `Message Receivers` in the same region we can add the following preference.

```
sum ?x of type MessageAnalyzer in '.*_A' :
  forall ?y of type MessageReceiver in '.*_A' :
    bind(?x,?y,MA)
```

In the first line we use the `sum` expression to match to the variable `?x` all the `Message Analyzer` instances hosted by a node whose name matches the regular expression `'.*_A'`. Similarly, in the second line we use the `forall` expression to match to the variable `?y` all the `Message Receiver` deployed in a node having a name ending with `'_A'`. The `forall` expression is evaluated to 1 if, fixing the possible assignments of the variable `?y`, the predicate `bind(?x,?y,MA)` is true (`MA` is the name of the interface required by a `Message Receiver` and provided by a `Message Analyzer`, see Figure 1). If instead there is an instance of a `Message Receiver` in region A that is not connected to the `Message Analyzer` `?x` than the forall expression returns 0. Due to the fact that the first expression is a sum expression, the final behaviors of the preference is to maximize the number of instances of `Message Analyzer` deployed in region A that are connected to all the instances of `Message Analyzer` deployed in the same region. Note that, if the `Message Receiver` is seen as a kind of loadbalancer for the `Message Analyzer` instances, what we have achieved is to state the preference that all the backend in a region should be connected with all their loadbalancers deployed in the same region.

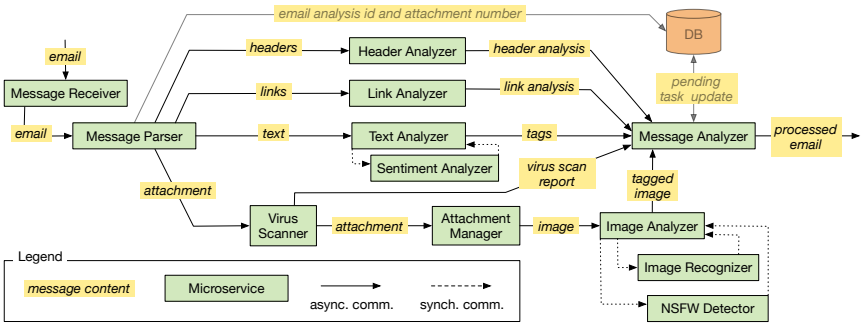

Fig. 4 Microservice architecture for email processing pipeline.

Zephyrus2 solves the previously described multi optimization problems, by translating them into *Constraint Optimization Problems* (*COP*) encoded in MiniZinc [46] and using state-of-the-art solvers such as Chuffed [22], Or-Tools [35], or Gecode [34]. In particular, preferences are given in a list based on the user priority. The early the preference comes in the list, the higher is its priority. Zephyrus2 optimizes the preference with highest priority first, and then proceeds with the other preferences sequentially based on their priority.

4 Application of the technique to the case-study

In this section, we evaluate the applicability of our solution by modeling several deployment configurations of a real-world microservice architecture, namely the email processing pipeline described in [33].

The considered architecture separates and routes the components found in an email (headers, links, text, attachments) into distinct, parallel sub-pipelines with specific tasks, e.g., check the format of the email, tag its content, detect malicious attachments. We report in Figure 4 a depiction of the architecture. The Message Receiver microservice is the entry-point of the architecture and acts as a proxy by receiving and triggering the analysis of incoming emails. The Message Receiver forwards a inbound email to the Message Parser, which performs some preliminary validity checks. If the message is well-formatted, the Message Parser first stores a pending-analysis task under a unique identifier for the current email in a companion database (DB) service. The DB maintains the status of all pending analyses in the system and it is an element external to the architecture — this is represented by the faded part at the top of Figure 4. After storing the pending task, the Message Parser *i*) splits the parsed email into four components: header, links, text, and attachments, *ii*) it tags them with the unique identifier of the pending-analysis task, and *iii*) it sends the four components to their corresponding sub-pipelines. The first two sub-pipelines from the top of Figure 4 include just one microservice, which

respectively analyze the headers (Header Analyzer) and the links (Link Analyzer) contained in the mail. The third sub-pipeline includes a Text Analyzer that synchronously invokes a Sentiment Analyzer, to add tags to the body of the message. The last sub-pipeline handles attachments and it is the most complex in the system. The first microservice in the sub-pipeline is a Virus Scanner, which checks each attachment for the presence of malicious software. If an attachment results malicious, it is deleted and signaled as dangerous to the Message Analyzer, as described later. Safe attachments are forwarded to an Attachment Manager for further analyses. The Attachment Manager inspects each attachment to identify its content type (image, audio, archive) and route it to the appropriate part of the sub-pipeline. In Figure 4 we just exemplify the concept with an Image Analyzer which synchronously calls two microservices to tag the content of each image (Image Recognizer) and whether it does not include explicit content (NSFW Detector). All sub-pipelines forward the result of their (asynchronous) analysis to the Message Analyzer, which collects them in the DB. After all analyses belonging to the same pending task are completed, the Message Analyzer combines them and reports the result of the processing.

To model the system above, we use the Abstract Behavioral Specification (ABS) language, a high-level object-oriented language that supports deployment modeling [41]. ABS is agnostic w.r.t. deployment platforms (Amazon AWS, Microsoft Azure) and technologies (e.g., Docker or Kubernetes) and it offers high-level deployment primitives for the creation of new *deployment components* and the instantiation of objects inside them. Here, we use ABS deployment components as computation nodes, ABS objects as microservice instances, and ABS object references as bindings. Strong required interfaces are modeled as class annotations indicating mandatory parameters for the class constructor: such parameters contain the references to the objects corresponding to the microservices providing the strongly required interfaces. Weak required interfaces are expressed as annotations concerning specific methods used to pass, to an already instantiated object, the references to the objects providing the weakly required interfaces. We define a class for each microservice type, plus one *load balancer* class for each microservice type. A load balancer distributes requests over a set of instances that can scale horizontally. Finally, we model nodes corresponding to Amazon EC2 instances: `c4_large`, `c4_xlarge`, and `c4_2xlarge`, with the corresponding provided resources and costs.

Finally, to compute deployment plans for our case-study, we exploit SmartDepl [36], an extension of ABS that supports the possibility to include into ABS additional deployment annotations that, besides the other annotations describing strong and weak required interfaces and the available computing nodes, are used as input for Zephyrus2. In this way, Zephyrus2 can compute optimal deployment plans, that are then translated into corresponding ABS code.

Each microservice in the architecture has a given resource consumption, expressed in terms of CPU and memory. As expected, the processing of each email component entails a specific load. Some microservices can handle large inputs, e.g., in the range of 40K simultaneous requests like the Header Analyzer that processes short and uniform inputs. Other microservices sustain heavier computations, like the Image

Microservice (max computational load)	Initial (10K)	+20K	+50K	+80K
MessageReceiver(∞)	1	-	-	-
MessageParser(40K)	1	-	+1	-
HeaderAnalyzer(40K)	1	-	+1	-
LinkAnalyzer(40K)	1	-	+1	-
TextAnalyzer(15K)	1	+1	+2	+2
SentimentAnalyzer(15K)	1	+3	+4	+6
VirusScanner(13K)	1	+3	+4	+6
AttachmentsManager(30K)	1	+1	+2	+2
ImageAnalyzer(30K)	1	+1	+2	+2
NSFWDetector(13K)	1	+3	+4	+6
ImageRecognizer(13K)	1	+3	+4	+6
MessageAnalyzer(70K)	1	+1	+2	+2

Table 1 Description of different scaling scenarios.

Recognizer, and can handle smaller simultaneous inputs, e.g., in the range of 10K requests.

In Table 1, we report the result of our algorithm w.r.t. four incremental deployments: the initial in column 2 and under incremental loads in 3–5. We also consider an availability of 40 nodes for each of the three node types. In the first column of Table 1, next to a microservice type, we report its corresponding maximum computational load, expressed as the maximal number of simultaneous requests that a microservice can manage. In the column, we use the standard suffix K to represent numbers in the thousands, e.g., 30K corresponds to 30.000 simultaneous requests. In our example, the maximal computational load of each microservice comes from an educated guess drawn from the experience of the authors. Concretely, those estimations are straightforward to obtain through e.g., a measurement of the performance like the response times of each microservice, under increasing simulated traffic loads. As visible in columns 2–5, different maximal computational loads imply different scaling factors w.r.t. a given number of simultaneous requests. In the initial configuration we consider 10K simultaneous requests and we have one instance of each microservice type and of the corresponding load balancer. The other deployment configurations deal with three scenarios of horizontal scaling, assuming three increasing increments of inbound messages: +20K, +50K, and +80K. Concerning the deployment plan synthesis, in the three scaling scenarios, we do not implement the planning algorithm described in *Phase 3* of the proof of Theorem 1. We take advantage of the presence of the load balancers: instead of emptying the current configuration and deploy the new one from scratch, we keep the load balancers in the configuration and simply connect to them the newly deployed microservice instances. This is achieved, as described in Section 3, with an optimization function that maximizes the number of bindings of the load balancers.

For every scenario, we use SmartDepl to generate the ABS code for the plan that deploys an optimal configuration, setting a timeout of 30 minutes for the computation of every deployment scenario.[9] The ABS code modeling the system and the

[9] Here, 30 minutes are a reasonable timeout since we predict different system loads and we compute in advance a different deployment plan for each of them. An interesting future work would aim at

generated code are publicly available at [14]. A graphical representation of the initial configuration is available in the technical report [16].

5 Related Work and Conclusion

In this work, we consider a fundamental building block of modern Cloud systems, microservices, and prove that the generation of a deployment plan for an architecture of microservices is decidable and fully automatable; spanning from the synthesis of the optimal configuration to the generation of the deployment actions. To illustrate our technique, we model a real-world microservice architecture in the ABS [41] language and we compute a set of deployment plans.

The context of our work regards automating Cloud application deployment, for which there exist many specification languages [8, 21], reconfiguration protocols [10, 30], and system management tools [37, 42, 47, 48]. Those tools support the specification of deployment plans but they do not support the automatic distribution of software instances over the available machines. The proposals closest to ours are those by Feinerer [31] and by Fischer at al. [32]. Both proposals rely on a solver to plan deployments. The first is based on the UML component model, which includes conflicts and dependencies, but lacks the modeling of nodes. The second does not support conflicts in the specification language. Neither proposals support the computation of optimal deployments. Notice that our work focuses on architectural aspects of (deployed) microservices and not on their low-level invocation flow, which regards issues of service behavioural compliance (see, e.g., [4, 12, 13, 18] where process algebra [7] related techniques are adopted) or deadlock/termination analysis (see, e.g., [9, 19]) that are not a concern of this paper.

Three projects inspire our proposal: Aeolus [23, 25], Zephyrus [1], and ConfSolve [38]. The Aeolus model paved the way to reason on deployment and reconfiguration, proving some decidability results. Zephyrus is a configuration tool based on Aeolus and it constitutes the first phase of our approach. ConfSolve is a tool for the optimal allocation of virtual machines to servers and of applications to virtual machines. Both tools do not synthesize deployment plans.

Regarding autoscaling, existing solutions [2, 6, 28, 39] support the automatic increase or decrease of the number of instances of a service/container, when some conditions, e.g., CPU average load greater than 80, are met. Our work is an example of how we can go beyond single-component horizontal scaling policies, as analyzed, e.g., in [17] by using Markovian process algebras [11].

As future work, we want to investigate local search approaches to speed-up the solution of the optimization problems behind the computation of a deployment plan. Shorter computation times would open our approach to contexts where it is unfeasible to compute plans ahead of time, e.g., due to unpredictable loads.

---

shortening the computation to a few minutes (e.g., around the average start-up time of a virtual machine in a public Cloud) to obtain on-the-fly deployment plans tailored to unpredictable system loads.

References

1. Ábrahám, E., Corzilius, F., Johnsen, E.B., Kremer, G., Mauro, J.: Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies. In: SETTA. LNCS, vol. 9984, pp. 229–245 (2016)
2. Amazon: Amazon cloudwatch. https://aws.amazon.com/cloudwatch/, accessed on January, 2019
3. Amazon: AWS auto scaling. https://aws.amazon.com/autoscaling/, accessed on January, 2019
4. Ancona, D., Bono, V., Bravetti, M., Campos, J., Castagna, G., Deniélou, P., Gay, S.J., Gesbert, N., Giachino, E., Hu, R., Johnsen, E.B., Martins, F., Mascardi, V., Montesi, F., Neykova, R., Ng, N., Padovani, L., Vasconcelos, V.T., Yoshida, N.: Behavioral types in programming languages. Foundations and Trends in Programming Languages **3**(2-3), 95–230 (2016)
5. ANTLR (ANother Tool for Language Recognition). http://www.antlr.org/, accessed on January, 2019
6. Apache: Apache mesos. http://mesos.apache.org/, accessed on January, 2019
7. Baeten, J.C.M., Bravetti, M.: A ground-complete axiomatisation of finite-state processes in a generic process algebra. Mathematical Structures in Computer Science **18**(6), 1057–1089 (2008)
8. Bergmayr, A., Breitenbücher, U., Ferry, N., Rossini, A., Solberg, A., Wimmer, M., Kappel, G., Leymann, F.: A systematic review of cloud modeling languages. ACM Comput. Surv. **51**(1), 22:1–22:38 (2018)
9. de Boer, F.S., Bravetti, M., Lee, M.D., Zavattaro, G.: A petri net based modeling of active objects and futures. Fundam. Inform. **159**(3), 197–256 (2018)
10. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: ICSE. pp. 13–22. IEEE Computer Society (2013)
11. Bravetti, M.: Reduction semantics in markovian process algebra. J. Log. Algebr. Meth. Program. **96**, 41–64 (2018)
12. Bravetti, M., Carbone, M., Zavattaro, G.: Undecidability of asynchronous session subtyping. Inf. Comput. **256**, 300–320 (2017)
13. Bravetti, M., Carbone, M., Zavattaro, G.: On the boundary between decidability and undecidability of asynchronous session subtyping. Theor. Comput. Sci. **722**, 19–51 (2018)
14. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Code repository for the email processing example. https://github.com/IacopoTalevi/SmartDeploy-ABS-ExampleCode, accessed on January, 2019
15. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Optimal and Automated Deployment for Microservices. In: FASE (2019)
16. Bravetti, M., Giallorenzo, S., Mauro, J., Talevi, I., Zavattaro, G.: Optimal and automated deployment for microservices. https://arxiv.org/abs/1901.09782 (2019), Technical Report
17. Bravetti, M., Gilmore, S., Guidi, C., Tribastone, M.: Replicating web services for scalability. In: TGC. LNCS, vol. 4912, pp. 204–221. Springer (2008)
18. Bravetti, M., Lanese, I., Zavattaro, G.: Contract-driven implementation of choreographies. In: Kaklamanis, C., Nielson, F. (eds.) Trustworthy Global Computing, 4th International Symposium, TGC 2008, Barcelona, Spain, November 3-4, 2008, Revised Selected Papers. Lecture Notes in Computer Science, vol. 5474, pp. 1–18. Springer (2009)
19. Bravetti, M., Zavattaro, G.: On the expressive power of process interruption and compensation. Mathematical Structures in Computer Science **19**(3), 565–599 (2009)
20. Brogi, A., Canciani, A., Soldani, J.: Modelling and analysing cloud application management. In: ESOCC. LNCS, vol. 9306, pp. 19–33. Springer (2015)
21. Chardet, M., Coullon, H., Pertin, D., Pérez, C.: Madeus: A formal deployment model. In: HPCS. pp. 724–731. IEEE (2018)
22. Chuffed Team: The CP solver. https://github.com/geoffchu/chuffed, accessed on January, 2019

23. Di Cosmo, R., Lienhardt, M., Mauro, J., Zacchiroli, S., Zavattaro, G., Zwolakowski, J.: Automatic application deployment in the cloud: from practice to theory and back (invited paper). In: CONCUR. LIPIcs, vol. 42, pp. 1–16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)
24. Di Cosmo, R., Lienhardt, M., Treinen, R., Zacchiroli, S., Zwolakowski, J., Eiche, A., Agahi, A.: Automated synthesis and deployment of cloud applications. In: ASE (2014)
25. Di Cosmo, R., Mauro, J., Zacchiroli, S., Zavattaro, G.: Aeolus: A component model for the cloud. Inf. Comput. **239**, 100–121 (2014)
26. Di Cosmo, R., Zacchiroli, S., Zavattaro, G.: Towards a Formal Component Model for the Cloud. In: SEFM 2012. LNCS, vol. 7504 (2012)
27. Docker: Docker compose documentation. https://docs.docker.com/compose/, accessed on January, 2019
28. Docker: Docker swarm. https://docs.docker.com/engine/swarm/, accessed on January, 2019
29. Dragoni, N., Giallorenzo, S., Lluch-Lafuente, A., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.: Microservices: Yesterday, today, and tomorrow. In: PAUSE, pp. 195–216. Springer (2017)
30. Durán, F., Salaün, G.: Robust and reliable reconfiguration of cloud applications. Journal of Systems and Software **122**, 524–537 (2016)
31. Feinerer, I.: Efficient large-scale configuration via integer linear programming. AI EDAM **27**(1), 37–49 (2013)
32. Fischer, J., Majumdar, R., Esmaeilsabzali, S.: Engage: a deployment management system. In: PLDI (2012)
33. Fromm, K.: Thinking Serverless! How New Approaches Address Modern Data Processing Needs. https://read.acloud.guru/thinking-serverless-how-new-approaches-address-modern-data-processing-needs-part-1-af6a158a3af1, accessed on January, 2019
34. GECODE: An open, free, efficient constraint solving toolkit. http://www.gecode.org, accessed on January, 2019
35. Google: Optimization tools. https://developers.google.com/optimization/, accessed on January, 2019
36. de Gouw, S., Mauro, J., Nobakht, B., Zavattaro, G.: Declarative Elasticity in ABS. In: ESOCC. LNCS, vol. 9846, pp. 118–134. Springer (2016)
37. Hat, R.: Ansible. https://www.ansible.com/, accessed on January, 2019
38. Hewson, J.A., Anderson, P., Gordon, A.D.: A Declarative Approach to Automated Configuration. In: LISA (2012)
39. Hightower, K., Burns, B., Beda, J.: Kubernetes: Up and Running Dive into the Future of Infrastructure. O'Reilly Media, Inc., 1st edn. (2017)
40. Humble, J., Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison-Wesley Professional (2010)
41. Johnsen, E.B., Hähnle, R., Schäfer, J., Schlatte, R., Steffen, M.: ABS: A Core Language for Abstract Behavioral Specification. In: FMCO (2010)
42. Kanies, L.: Puppet: Next-generation configuration management. ;login: the USENIX magazine **31**(1) (2006)
43. Mauro, J.: Zephyrus2 code repository. https://bitbucket.org/jacopomauro/zephyrus2
44. Mauro, J., Zavattaro, G.: On the complexity of reconfiguration in systems with legacy components. In: MFCS. LNCS, vol. 9234, pp. 382–393. Springer (2015)
45. Merkel, D.: Docker: lightweight Linux containers for consistent development and deployment. Linux Journal **2014**(239), 2 (2014)
46. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: MiniZinc: Towards a standard CP modelling language. In: CP. pp. 529–543 (2007), http://dl.acm.org/citation.cfm?id=1771668.1771709
47. Opscode: Chef. https://www.chef.io/chef/, accessed on January, 2019
48. Puppet Labs: Marionette collective. http://docs.puppetlabs.com/mcollective/, accessed on January, 2019