ALMA MATER STUDIORUM · UNIVERSITÀ DI BOLOGNA

SCUOLA DI SCIENZE

Corso di Laurea Magistrale in Informatica

Optimal and Automated Microservice Deployment: formal definition, implementation and validation of a deployment engine

Relatore:
Chiar.mo Prof.
Gianluigi Zavattaro

Co-Relatore:
Chiar.mo Prof.
Jacopo Mauro

Presentata da:
Iacopo Talevi

Sessione I

Anno Accademico 2018-2019

Introduction

The idea of this project was born in July 2018. Its realization involved Professor Mario Bravetti and my supervisor Professor Gianluigi Zavattaro from the University of Bologna, Professor Jacopo Mauro and post-doctoral researcher Saverio Giallorenzo from the University of Southern Denmark, and myself. The main purpose of our work was to study the problem of optimal and automated deployment and reconfiguration (at the architectural level) of microservice systems, proving formal properties and realizing an implemented solution.

Our work started from the theory described in [1], where the *Aeolus component model* was used to formally define the problem of deploying component-based software systems and to prove different results about decidability and complexity. In particular, the authors formally prove that, in the general case, such problem is undecidable. But they also show that by inserting limitations on the model expressivity, the analysed problem becomes decidable but very complex (Ackermann-hard) in one version, and even polynomial in time in another.

Starting from these results we expanded on the analysis of automated deployment and scaling, focusing on microservice architecture. Microservices are a variant of the service-oriented architecture (SOA) based on small, fine-grained and loosely coupled services. Using a model inspired by Aeolus, considering the characteristics of microservices, we formally proved that the optimal and automated deployment and scaling for microservice architectures are algorithmically treatable. However, the decision version of the problem is NP-complete and to obtain the optimal solution it is necessary to solve an NP-optimization problem. Thanks to a formulation of the algorithm based on the constraint programming paradigm, state-of-the-art constraint solvers can be used to search for the optimal solution.

To show the applicability of our approach we decided to also realize a model of a simple but realistic case-study. We selected a microservice architecture that implements an email processing pipeline, inspired by Iron.io and described in [2]. The model is developed using the Abstract Behavioral Specification (ABS) language [3, 4], and to calculate the different deployment and scaling plans we used an ABS tool called SmartDepl [8]. The tool allows to specify through annotations on the ABS code all the necessary information and it returns the solution in the form of ABS classes. To solve the problem, SmartDepl relies on Zephyrus2 [10]. Zephyrus2 is a configuration optimizer that allows to compute the optimal deployment configuration of described applications.

This work resulted in an extended abstract [12] accepted at the Microservices 2019 conference in Dortmund (Germany) [13], a paper [14] accepted at the FASE 2019 (part of ETAPS) conference in Prague (Czech Republic) [16], and an accepted book chapter [17].

In addition to the opportunity to work side-by-side with professors, this experience gave me the chance to actively collaborate to research activities and to participate to my first scientific conference in Dortmund. There I presented our work and shared ideas and solutions with students, researchers, and professors, as well as with representative of the corporate sector.

This dissertation starts with Chapter 1 that introduces the Aeolus model and the decidability and complexity results obtained in [1]. These are the starting points of our work and the chapter introduces ideas and concepts that are re-used in the new model proposed. Then, our novel contributions are introduced in Chapter 2. We present the new proposed model, specifically built for microservices, and the formal results obtained on it. To conclude, Chapter 3 discusses the case-study used to show the applicability of the solution proposed. It also briefly introduces SmartDepl to allow the reader to try and use our approach on different applications. Finally, the conclusion provides a summary of the contributions presented and briefly discusses possible evolutions.

Contents

List of Figures

List of Tables

Chapter 1

Aeolus Component Model and Undecidability Proof

The Aeolus Component Model is a previous contribution from Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli and Gianluigi Zavattaro, presented in [1]. It provides a formal way to represent component-based software configuration and deployment in complex distributed cloud applications.
Cloud applications are software that run in a cloud environment, where the different components are executed through virtual machines hosted in heterogeneous hardware. They are usually offered by cloud providers and payed with a pay-per-use approach. Using this approach, the users can request more machines when necessary and drop them when the corresponding computation is finished. All the systems are managed and connected on-the-fly, so it is a very dynamic reality. A cloud environment does not necessarily imply a public setting through big providers such as Amazon Web Service [19], Google Cloud Platform [20], or Microsoft Azure [21] but it can also be simulated through a collection of private machines (Private Cloud). Cloud gives a lot of advantages such as cost reduction, flexibility, scalability, mobility, disaster recovery and many others, but it also significantly increases the complexity during the design, configuration, deployment and maintain phases of a software. To address these challenges, in the last years, differ-

ent tools have been developed both in academia and in the industry using a range of different approaches. Examples are:

- **Academic side:** Fractal Component Model [22] and FraSCati [23], ConfSolve [24]
- **Industry side:** Chef [25], CloudFoundry [26], Juju [27]

In short, these tools help users put together specifically prepared system components through a ready-made user-prepared configuration. So it is a user's responsibility to choose which components to instantiate and how to interconnect them. Clearly, if a reconfiguration is necessary, the users have to do it themselves, manually or through specifically prepared codes.

Analysing these tools, the authors of [1] detected two main and necessary characteristics for a new model:

- **Expressivity:** the model should allow a user to capture and describe all aspects of a complex distributed and scalable application. Typical aspects are: dependencies, conflicts and non functional requirements. Examples of non-functional requirements are:
  - number of instances of each component type required to guarantee a particular service level agreement(SLA), or
  - fault tolerance, or
  - location of different instances of a specific component type to ensure fast communication everywhere, or
  - replication, and others.
- **Automation:** it is necessary to have a tool that takes as input an abstract description of the requested target configuration and calculates the necessary steps to reach it starting from the current state. This need appears very clearly in systems with a high number of components because the complexity of deploying and configuring them significantly increases and these phases cannot be easily managed manually.

```
Package: mysql-server-5.5
Source:  ...
Version:  ...
Provides: mysql-server, virtual-mysql-server
Depends: libc6 (>= 2.12), zlib1g (>= 1:1.1.4), debconf, [...]
```

Figure 1.1: Debian package metadata for MySQL

The Aeolus Component Model is presented in the following sections, using definitions and examples taken from [1]. A better and more formal explanation can be found in the referenced paper. The description of the model provided here is useful to introduce and better understand the new contributions described in Chapter 2.

1.1 Introduction to Aeolus Component Model

To introduce the Aeolus Component Model it is a good idea to take inspiration from the package paradigm used for software installation. In the package paradigm, popularized by FOSS (Free/Open Source Software) distributions, a package contains all the data connected with a specific software. So it does not contain only the software artefact but also other information like configuration settings and metadata. Clearly, a package in a machine can be in two states: uninstalled before the installation process and installed at the end. But during the installation process, the package passes through other states (e.g. unpacking, configuration). The concept of **state** will be fundamental in the Aeolus component definition. In each state a package can have some **requirements** and/or offer some features that are usually called **provides**. To give a practical intuition the authors use a Debian package description as example, showed in Figure 1.1, where the described information are specified through two fields, called *Provides* and *Depends*. A package life-cycle can be represented and modelled through a **state machine**. Each state represents a step during the component life-cycle and it could have requirements that must be satisfied and provides that can be used by other

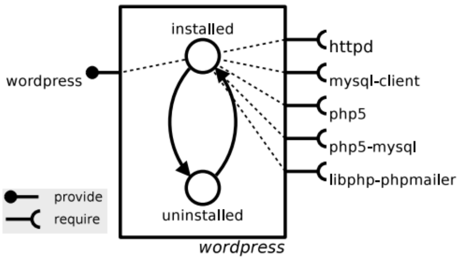

Figure 1.2: Graphical representation of wordpress component type

components to fill their requirements.
In addition to *requirements*, *provides* and *state machine*, there are other useful notions:

1.1.1 Component Type

Using the previously described ideas, it is possible to define a *component type* as 5-tuple <Q, q_0, T, <P,R>, D> where:

- Q, q_0, T are the classic components of a state machine: a set of states, an initial state and a transition function,
- while <P,R> and D are particular fields:
  - P is the set of provides,
  - R is the set of requirements,
  - and D is a function that links each state with its corresponding provides and requirements sets.

The authors use the following graphical notation, showed in Figure 1.2, to represent a component type.

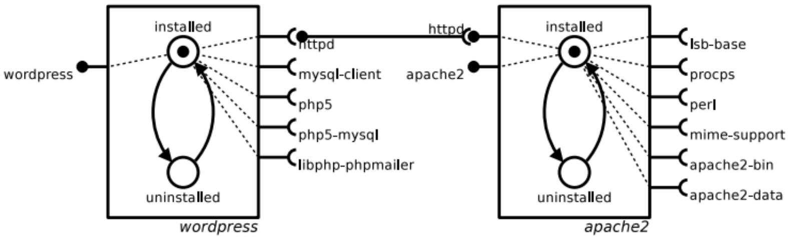

Figure 1.3: Partial configuration with two components

1.1.2 System Configuration

Using these concepts it is possible to build a definition of *system configuration*. A configuration is a collection of instances of different component types, in particular states, and the connections between them.

A connection is a link between a required port of a component and a provided port of another. It means that the second component provides the feature needed by the first one. Consequently the corresponding requirement is satisfied.

A configuration is correct when all active requirements (requirements requested in the current state) are satisfied by active provided ports. Using the previous graphical notation, it is possible to represent a partial configuration with two components, where the second one satisfies a requirement of the first one, Figure 1.3.

1.1.3 Dependencies, Capacity Constraints and Conflicts

It is possible to observe that connections represent dependencies between components. In the previous image, Figure 1.3, the *wordpress* component has a dependency with the *apache2* component on the *httpd* interface. Consequently, to allow *wordpress* to reach the installed state, *apache2* must be already installed.

In addition to dependencies, Aeolus wants to allow users to model also conflicts and non functional requirements (e.g. redundancy). To capture situations connected with redundancy, Aeolus allows to add numeric constraints to required and provided ports.

- A number n linked with a provided port means that the corrisponding port can be connected with **at most** n required ports, so it can satisfy at most n requirements,

- while a number n linked with a required port means that the corrisponding port must be connected with **at least** n provided ports, so to be satisfied it requires at least n provides from n different components.

Finally, to model conflicts, Aeolus uses the number 0 linked with a required port. This means that the required port with 0 and a provided port for the same interface (with the same name) cannot be active at the same time. Consequently this particular case can be used to model global conflicts between components.
Again the graphical representation can be used to better understand the described situations. The authors of [1] provide an example with three component types: *wordpress*, *mysql* and *varnish*. Varnish is a load balancer that in this example requires at least three Wordpress back-end instances. Wordpress has a dependency with Mysql that consequently is necessary to run a Wordpress instance. But a Mysql component can serve no more than 2 clients. This example is showed in Figure 1.4.

1.1.4 Dynamic Configuration Updates

In a cloud environment is possible to rent and release virtual machines on the fly and in the same way components can be allocated or deallocated following the load changes of the system (scale-up or scale-down). The Aeolus dynamic updates management can be showed analysing two possible situations starting from the previous example:

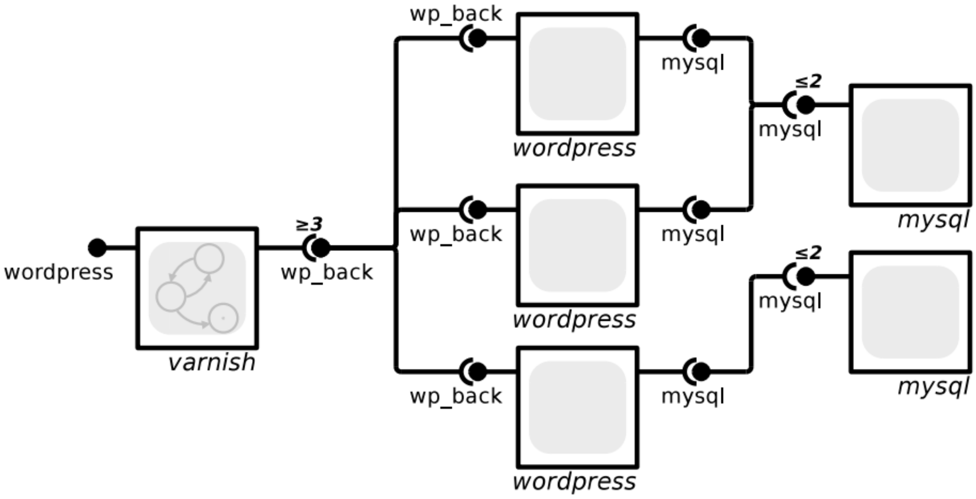

Figure 1.4: Graphical representation of capacity constraints on provided and required ports

- if the work load is very low, the user can:
  - change the capacity constraint in Varnish required port from 3 to 2
  - and consequently deallocate an instance of Wordpress.

  In an automatic way it is also possible to destroy an instance of Mysql that is no longer necessary,

- if the work load is very high, the user can:
  - decide to allocate two more instances of Wordpress.

  In an automatic way another instance of Mysql is created to satisfy the requirements of Wordpress instances. In fact one of the new instances can be connected with the already available instance of Mysql with only one link but the second one needs a new instance of Mysql because the previous ones are no longer enough.

Using the Aeolus model is therefore possible to automatically create or destroy components when they are necessary or they are no longer used.
After this intuitive introduction of the model with the related graphical notation, we present the main definitions used to define Aeolus.

1.2 Aeolus Definitions

The cited paper [1] provides a list of nine definitions to formally define the Aeolus model. In this section the ideas behind these definitions will be introduced.
As mentioned above, all components are modelled as finite state automatons. Each state has a set of provided and required ports, which describe the functionalities offered or needed by that component in that specific state. The different functionalities are called *interfaces* and their names are used as port names in the graphical representation. They are grouped in the set $\mathcal{I}$.
The first definition is the **Component Type** declaration. As already described in the previous section a component type is a 5-tuple <Q, q_0, T, <P,R>, D>:

- Q, q_0, T are the classic components of a state machine: a set of states, an initial state and a transition function,
- $< P, R >$ are the set of provides and the set of requirements. Formally they are subsets of the interface set: P,R $\subseteq \mathcal{I}$,
- D is a function that takes a state s as input and returns a couple of partial functions: $(P \rightarrow \mathbb{N}_\infty)$ and $(R \rightarrow \mathbb{N}_0)$.
  They are defined only in the provided and required ports active in the state s. They take a port as input and return the corresponding capacity constraints. The value returned is
  - the number of requirements that the selected port can satisfy (for

provided ports): from 1 to ∞ (the default case) that allows an unlimited amount of bindings.

  - the number of necessary connections between different components to satisfy the selected port (for required ports): from 1, the default case, to N.
    0 is a possible value used to represent conflicts.

In the previous section, the **Configuration** definition has already been introduced too. Formally, it is a 4-tuple with:

- the set of possible component types called *universe*,
- the set of deployed components,
- a function that takes a deployed component as input and returns its type and its current state and
- the set of bindings where each bind is represented as a 3-ples $< port, consumer, provider >$ where:
  - *port* is an interface
  - *consumer* and *provider* are two distinct deployed components.

The **Configuration Correctness** is formally defined through three points.

- The first one checks that conflicts are respected. If a component has an active required port with 0, the definition says that there cannot exist other components in the current configuration that have an active provided port with the same interface.
- The second one controls that for each required port there are enough distinct deployed components that provide the corresponding interface.
- The last one ensures that each provided port respects its capacity constraint. In particular, that it is connected with at most N required ports, where N is the value specified by the user.

When all the three points are respected the configuration is correct.
To pass from one configuration to another the authors formally define five possible **Actions**:

- *stateChange*(z, q_1, q_2): where z is a deployed component and q_1, q_2 are two possible state for it. This action controls that:

  – z is in the q_1 state,

  – exists a transition for z from q_1 to q_2

  and in that case it executes the transition arriving to a new configuration where the component z is in the new q_2 state.

- *bind*(r, z_1, z_2): where r is an interface and z_1, z_2 are two deployed components. This action controls that:

  – there is not already a binding between z_1 and z_2 on the interface r,

  – that r is a z_1 active required port and a z_2 active provided port

  and in that case it passes to a new configuration where the new connection from the cited deployed components through the described interface is established.

- *unbind*(r, z_1, z_2): where r is an interface and z_1, z_2 are two deployed components. This action controls that:

  – there is a binding between z_1 and z_2 on ports with interface r

  and in that case it passes to a new configuration where it removes the identified connection.

- *new*(z): where z is a new component. This action controls that:

  – z is not already deployed,

  - the component type of z belongs to the component type universe allowed

  and in that case it adds to the current configuration the new deployed component z in its initial state.

- *del*(z): where z is a deployed component. This action removes the deployed component z and all its connections from the current configuration.

The transition from the current configuration C to a new configuration C' after the execution of α, that is one of the actions described above, can be represented through a labelled transition systems C $\xrightarrow{\alpha}$ C'.
Notice that actions do not check the correctness of the reached configurations. These controls, that allow to understand if an action is possible (i.e. it leads to a correct configuration), will be considered during the definition of *deployment run*.
The actions described are not enough to reach all the possible configurations. For example a configuration with a circular dependency (a component *a* requires for its intallation that a component *b* is installed, and in a dual mode a component *b* requires for its intallation that a component *a* is installed) cannot be managed. To solve it, the authors introduce an extension of *stateChange* called *Multiple state change* that allows to execute a set of state change actions on different components at the same time as an atomic transition. In a later work [31], it is proved that *Multiple state change* is not really necessary. The cited paper shows that without this additional operation the Aeolus component model remains Turing complete and all the proofs obtained in the previous works remain possible and valid.
A **Deployment Run** can be defined as the application of actions to reach, from an initial configuration a target configuration ensuring that all the crossed configurations are correct and multi stage change operations used are minimal.
With all the previous definitions, it is possible to define the **Achievabil-**

ity Problem. This is the decision problem that studies whether, from an empty configuration, it is possible to reach a final configuration that contains an instance of a specified component type in a specified target state. If the necessary *deployment run* exists, it returns true, otherwise, it returns false. Notice that considering only a component in a given state is not limiting. In fact, if we want to consider more components it is enough to add to their final state a dummy provided port and to require these ports, through corresponding required ports, in the specified state of the single requested component. In this way if the achievability problem returns true, it means that there exists a deployment run to reach a configuration where our dummy component is in the requested state. But the requested state has a required port for each other components wanted which ensures that they are all correctly deployed.

1.3 Aeolus Undecidability Proof

The authors prove that the achievability problem for Aeouls model is undecidable. This is the main result of [1]. The proof will be shortly introduced here, but it can be found in a formal and complete version in the referenced paper.

They use a reduction proof showing that the achievability problem can be reduced to the reachability problem in 2 Counter Machines (2CMs) [28]. 2CM is a Turing-complete computational model. A 2 Counter Machine is an abstract machine with:

- two registers, containing non-negative numbers, used as counters
- and a finite sequence of numbered instructions. There are only two possible instruction types:
  - j : Inc(R_i): increments register R_i and passes to the next instruction in position $j + 1$;
  - j : DecJump(R_i, l): controls the value of R_i: if it is greater than 0, then it decreases it by 1 and it passes to the next instruction

with index $j + 1$; otherwise the register cannot be decreased so it jumps to the instruction with index l.

Intuitively the current state of a 2CM can be represented through a 3-ple containing the index of the next instruction and the values of the two registers. For the initial state a configuration can be used with the first instruction, index 1, and 0 as value for the two registers (1, 0, 0).

The reachability problem in 2CMs asks to understand if a specific instruction, identifiable through its index, is reachable starting from the initial state. This problem is undecidable.

The proof shows how to model a 2CMs through the Aeolus model. It defines two main component types, one to simulate the execution of the program and one to simulate a register. To simulate a value v for the register i, the proof uses v components in a particular state r_i. To obtain the value of register i it is necessary to count the number of components in state r_i. Each "active" register component has a provided port one_i. Consequently, to check if a register is empty (value = 0), during the execution of a DecJump operation, it is enough to control the absence of that port. This can be done using a conflict on it. An exact description of these components and of the protocols used to implement the 2CMs operations can be found in the paper.

The first step of the proof shows that there exists a deployment run from an empty configuration to a configuration that model the initial state of 2CM. Then, the real proof is organized in two prepositions:

- *Completeness*: proves that each step in a 2CM can be simulated through a deployment run in the built model. The authors show exactly how an increment instruction can be simulated in the prepared Aeolus model and they specify that decrement and test for zero simulations are very similar.

- *Soundness*: proves that each possible action in the prepared Aeolus model corresponds to an instruction in a 2CM. The authors analyse all the possible moves from a generic state in the Aeolus model showing that they correspond to actions in the 2CM.

At the end, they use the previous prepositions to show that a particular instruction in 2CM is reachable *if and only if* there exists a deployment run from an empty configuration to a configuration that capture a 2CM state with that instruction. Consequently, the undecidability of achievability thus follows from the undecidability of reachability for 2CMs.

1.3.1 Decidability version of Aeolus and their Complexity

In addition to the previous results, the authors define two restrictions of Aeolus model limiting the available capacity constraints. They prove that, for these two simplified versions, the achievability problem becomes decidable and they provide also a study about their complexity.

- $Aeolus^{-}$ is the simplest model where only the default capacity constraints are available. So the provided port can serve an unlimited number of requirements and required ports are satisfied with only one binding. The authors provide a polynomial decision algorithm for this model.

- Aeolus core is an extension of $Aeolus^{-}$ adding the possibility to represent conflicts with value 0 on required ports. This model is clearly more complex than the previous one, it remains decidable but not primitive recursive (i.e., Ackermann-hard). They prove the decidability using the theory of Well-Structured Transition Systems (WSTS) [29] and the complexity with a reduction proof from the coverability problem in reset Petri nets [30].

Chapter 2

Optimal and Automated Deployment for Microservices

Starting from the Aeolus component model and the undecidability result described in Chapter 1, in this research project we studied a similar problem focused on Microservice architectures. This can be summarized as the research of a method to obtain *Optimal and Automated Deployment and Scaling Plans for Microservice Architectures.*
In this work, a new model – strongly inspired by Aeolus – has been defined to better capture the microservice distinctive traits, and formal results on decidability and complexity have been obtained. The main result, which this project proved, is that the optimal deployment problem for microservices is algorithmically treatable.
With respect to Aeolus, the new model also considers the distribution of microservices over computation nodes, introducing the concepts of resources required from software components and provided by computation nodes, and the costs of the latter. The optimality is defined on the total cost, or rather, the sum of the costs of all computation nodes used.

2.1 Microservices, basic information

The Microservice approach is an architectural style inspired by service-oriented architecture (SOA). It is very popular today and its adoption in software design is rapidly growing. Here, we provide a short introduction, focusing on the main characteristics that are useful to understand our contributions.

A widely spread way to design software, particularly in the past, was the monolithic approach, where the code for all the features is contained in a single application, called monolith. With the growth of application dimensions and complexity the monolithic approach has become problematic. It has therefore been replaced by a service approach called Microservices. With this new approach, applications are structured as collections of fine-grained and loosely coupled services that can be independently developed and deployed and that are organized around business capabilities. These aspects allow to easily design, develop, test, maintain, scale and expand an application. In practice, with a Microservice approach each software functionality is isolated and developed as an independent module. Each module should solve only the single task assigned. The different modules communicate with each other through application programming interfaces called APIs.

Two images showed in Figure 2.1, taken from [32], give an intuitive but clear idea of differences between the monolithic and the microservice approach.

2.2 Introduction to a new approach for microservices management

The microservices characteristics support continuous delivery/deployment [33] and application autoscaling [34, 35] that are two modern software engineering practices. These two practices strictly follow the microservice idea, managing each component independently from the others. For example, autoscaling supports the independent increase or decrease of microservice

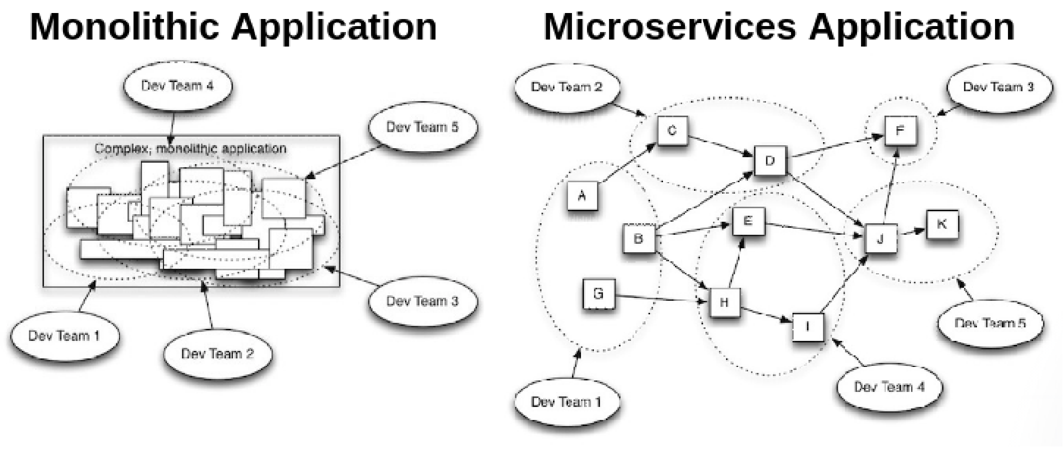

Figure 2.1: Monolithic vs Microservices approach

instances, based on the values of monitored metrics (CPU average load, response time, ...). Autoscaling independence is strongly grounded on the principle of loose-coupling at the base of microservices. An example of Amazon Web Services Autoscaling Application can be observed in Figure 2.2.

Reasoning at local microservice level, analyzing each component independently, these practices cannot exploit architecture information, in particular information about microservices interdependencies.

During this research project an alternative approach has been proposed. It tries to reason at architecture-level to have the possibility of reach the global optimization of resources usage. Reasoning at higher-level allows to add several instances of different services at once, reaching optimal placement of such instances. This objective cannot be obtained through unstructured and independently scaling actions.

A clearer idea of this new approach can be given through an example. If the user detects a peak of inbound requests on the microservice that works as entry point of a pipeline of sequentially-interdependent services, it would be more efficient in time to immediately scale all the microservices in the pipeline, instead of letting each microservice successively autoscale. In addition, managing the deployment of more instances at the same time gives the

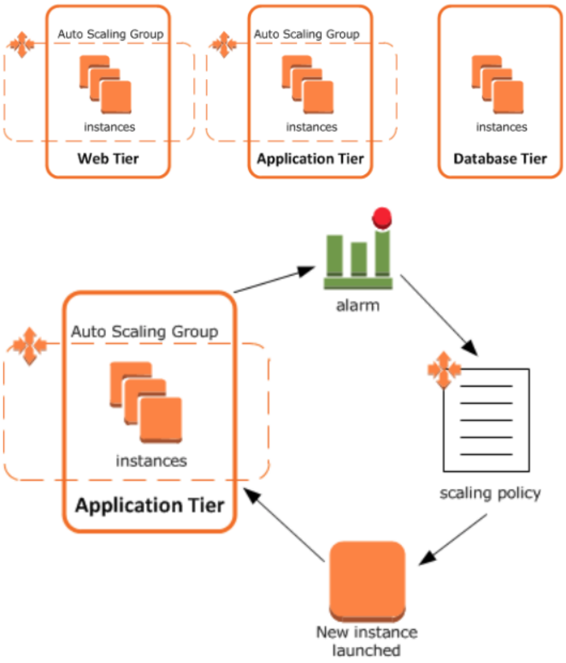

Figure 2.2: The AWS autoscaling service, from AWS documentation

possibility to be efficient also in the resources usage, computing the optimal deployment plan. This optimisation is impossible to achieve by solely relying on horizontal autoscaling.

2.3 Introduction to the new model proposed for microservices

To formally study the new approach introduced in the previous section, a new specific model for microservices has been defined and the problem of automated deployment and reconfiguration (at the architectural level) of microservice systems has been formalized. As already mentioned in the introduction of this chapter, these operations have been performed following the approach taken by *Aeolus component model* described in Chapter 1.

2.3.1 Microservice Component

The general software component of Aeolus described through a full power finite state automata is substituted with a *microservice component*. It is always modelled through a finite state automata but with a fixed set of states. The only two possible states are:

(i) **Creation:** manages the dependencies called *strong*, which identify microservices that have to be connected during the creation of a new instance. Without them, the new instance cannot be created.

(ii) **Binding/Unbinding:** manages the dependencies called *weak*. They must be fulfilled before the end of the deployment plan, but during this one the corresponding requirements can be temporarily unsatisfied.
Weak dependencies can also be used to identify microservices that can be connected/disconnected during the instance life. For example, in the case-study presented in Chapter 3, this type of dependencies is used in the load balancer definitions to allow for new instances connections during an horizontal scaling operation.

This deployment life-cycle, but also the resource modelling process described later, have been inspired by state-of-the-art microservice deployment technologies like Docker [36, 37] and Kubernetes [39, 40].
The distinction between strong and weak dependencies is inspired by Docker Compose [38]. It allows to define, configure and run a multi-container Docker application through a YAML configuration file.

Kubernetes

Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. The Kubernetes project was started in 2014 by Google.
A container is a software package containing code and all its dependencies. Containers allow an application to be easily portable and quickly executed on

different computing environments. They are created executing container images that include everything that is needed to run the corresponding applications: code, runtime, system tools, system libraries and settings. Containers are similar to virtual machines but they are based on operating-system-level virtualisation rather than hardware virtualisation. Consequently they are easier to build and lighter than VMs, and because they are decoupled from the underlying infrastructure and from the host filesystem, they are portable across clouds and OS distributions.
Some of the main Kubernetes features are:

- it automatically distributes containers onto nodes considering their resources requirements and other constraints,
- it restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond,
- it allows to easily scale-up or scale-down the application with a simple command or automatically based on different possible metrics,
- it allows to update the application run-time. Kubernetes progressively rolls out changes to the code or to the configuration. At the same time, it monitors the application to verify that everything proceed correctly. If something goes wrong, Kubernetes rolls back the change, reverting to the last stable state.

Kubernetes API objects are used to work with Kubernetes. They are used to describe the desired cluster state. The Kubernetes Control Plane, a collection of processes running on the cluster, automatically modifies the current system state to reach the described one. A lot of different possible operations can be automatically executed to reach the fixed goal, examples are: starting or restarting containers, scaling the number of replicas of a given application, and others.
A Kubernetes cluster consists of a set of nodes. A node is a worker machine that can be a VM or a physical machine. The system maintains a lot of

different information on each node, such as the node's address or the amounts of resources offered. There are two types of nodes:

- The Kubernetes master is a single node responsible for maintaining the desired state in the cluster.
- Non-master nodes that serve as worker machines executing the assigned (by the master) processes.

Kubernetes offers a set of possible abstractions to easily describe a system. These abstractions are represented by objects in the Kubernetes API. These objects are organized on two levels. The lower level contains objects such as Pods or Services, while the higher level contains objects called Controllers, including ReplicaSet or Deployment. A Pod is the basic building block of Kubernetes. It is the smallest and simplest unit that can be deployed. It can be seen as a box for an application container. Usually Pods are not directly managed by users but created and supervised by the higher-level objects. These highly simplify the management of the Pods, by handling the creation in an automatic way, conducting replication/scaling, rollout, fail-recovery and other phases of a Pod life-cycle. For example, if a Deployment is used to manage a Pod, in order to execute a scale operation it is enough to change the number of replicas in the specification of the corresponding Deployment. Then, this object produces and executes all the necessary operations to reach a new state where the selected component is scaled. The following commands show how to directly scale a Deployment or how to set an autoscaling service on a Deployment:

```
kubectl scale deployments/name −replicas=4
kubectl autoscale deployment name −min=10 −max=15 −cpu−percent=80
```

To create any object, the user has to write the relative specification. This is usually provided using a YAML file. An intuitively example for a Deployment is:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Information about the CPU and the memory (RAM) required can also be optionally inserted in a Pod specification. In that cases the Kubernetes scheduler ensures that, for each resource type, the sum of the resource requests from the scheduled Pod in a node is smaller than the capacity of that node. In general, the scheduler automatically implements a reasonable placement of Pods, but it is also possible to customize the Pods distribution over the available nodes by specifying some constraints. For this purpose a simple node selector or a more complex feature called affinity/anti-affinity can be used.

2.3.2 Resources and Costs Introduction

Resources Introduction

To consider resources, the architecture specifications are enriched with information about them. In particular, the new model considers two types of resources information:

- resources offered by computation nodes to host microservices,
- resources required by each microservice to be executed.

The resources usually considered are CPU and memory.
As mentioned above, Kubernetes uses a similar approach:

- it allows to specify the capacity of each node in a Kubernetes cluster providing:
  - **CPU**,
  - **memory** and
  - the maximum number of pods that can be scheduled onto the node.

  (This last information has not been considered during this work but it could be translated as the maximum number of instances that can be hosted onto a node.)
- it allows to specify how much **CPU** and **memory** (RAM) each Container needs. Containers can be seen as microservice instances in the current situation.

Costs Introduction

Computation node are also introduced. They must be payed to use the corresponding node and obtain the possibility to deploy instances on it. Consequently, a good example for a computation node is a virtual machine offered by a Cloud provider in a Infrastructure-as-a-Service environment. In this situation a virtual machine has a specific amount of resources provided and a cost that must be paid to use it, just like the introduced computation nodes (e.g. AWS EC2 instance: c4.large → vCPU=2, Memory=3.75 GiB, Cost:$0.10 per Hour [44]).

2.3.3 An Informal Introduction to the Optimal Deployment Problem

At this point, with reference to the new model just introduced, the *optimal deployment problem* can be informally introduced:

- Input:
    - an initial microservice system that is the starting point of the process,
    - a set of available computation nodes that can be used to extend the system. They are described through a name and the information about resources offered and the corresponding costs,
    - a description of the deployable microservices, with information on resources usage and dependencies,
    - a target microservice for which at least one instance must be deployed in the final configuration.
- Computation:
  the tool searches a sequence of reconfiguration actions that allows to move from the initial configuration to a configuration that contains the instance requested.
- Output:
  the sequence of reconfiguration actions that correspond to the deployment plan searched. Its optimality in term of total costs is guaranteed.

2.3.4 An Example

To clarify the differences between the new model and the Aeolus model, and to get the main intuition behind the formal definitions provided below, consider the following example, created from a simplified version of the case-study that will be discussed in Chapter 3. In this version of the studied Email Processing Pipeline Microservice Architecture, only three different microservices are considered: an entry-point to the system called *MessageReceiver*, an email analyser called *MessageAnalyzer* and another component called *AttachmentAnalyzer* that can be used by *MessageAnalyzer* to control the attachments, if necessary.
The three microservices are connected in the following way:

- **MessageReceiver** → **MessageAnalyzer** through a weak interface. This means that a MessageReceiver instance can be initially deployed without this connection and it can be established subsequently. This type of connection also allows to capture the possibility of horizontally scaling this part of the application by adding/removing instances following the system load. The numerical constraint on the involved required port requires that at least three MessageAnalyzer instances have to be connected in the final configuration.

- **MessageAnalyzer** → **AttachmentAnalyzer** through a strong interface. So this connection is immediately necessary. Indeed, to deploy a new instance of MessageAnalyzer an available instance of AttachmentAnalyzer that provides the necessary functionalities to the new component is required. The numerical constraint on the involved provided port specifies that at most two instances can be connected to the AttachmentAnalyzer to use the functionalities provided.

The described situation can be summed up using an extended version of the graphical notation introduced in Chapter 1, as done in Figure 2.3.

The example presented in the Figure 2.3 shows a reconfiguration process,

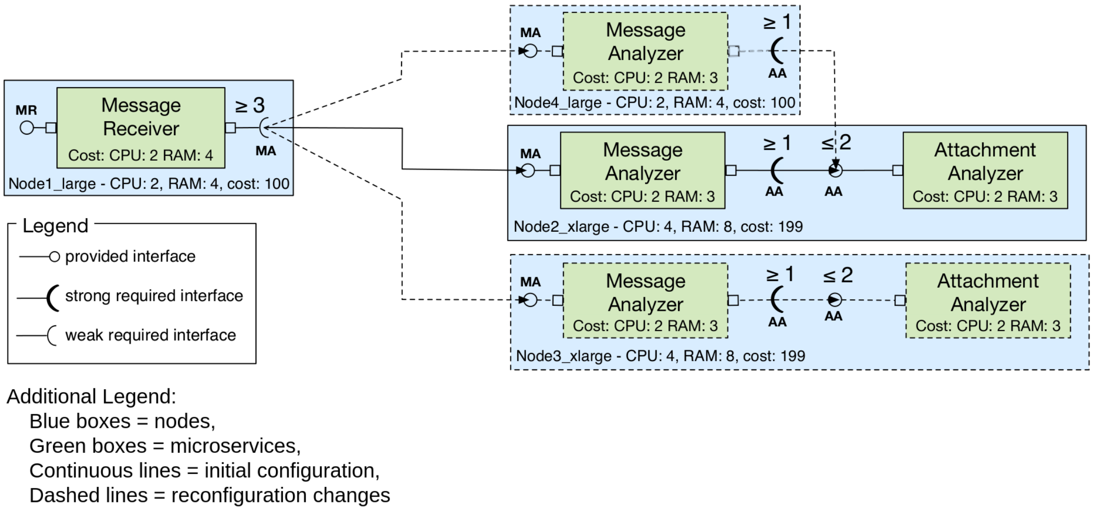

Figure 2.3: Introduction to the extended graphical notation

too. In particular, the components represented with continuous lines are already deployed and they are the initial configuration. Instead, the dashed lines represent the components that are deployed during the reconfiguration process and that are necessary to satisfy the interfaces' numerical constraints. It is possible to observe that the constraint on the *MessageReceiver* required port is not satisfied with the instances already deployed. This is not a problem because it is a weak required port so it has to be considered only at the end of the reconfiguration plan. To satisfy this constraint at least two more instances of *MessageAnalyzer* have to be deployed. The first one can connect its strong required port with the *AttachmentAnalyzer* already available. But the second one cannot use it because the corresponding provided port has a numerical constraint that blocks more than two bindings. Consequently, also a new instance of *AttachmentAnalyzer* is necessary to be able to deploy the second necessary instance of *MessageAnalyzer* to successfully satisfy the *MessageReceiver* required port constraint at the end of the reconfiguration plan.

This example uses the new graphical notation adopted to also describe resources and costs. Simply, these pieces of information are written under each component. A microservice only contains information about the required resources, while a node contains information about resources offered and the corresponding cost. Observing a configuration's graphical representation, its total cost can be easily obtained summing the costs reported in each node used. In the previous example, Figure 2.3, the total cost is 598 cents per hour. In the case-study analysed in Chapter 3 and just introduced in a simplified version, the computation node costs are inspired by Amazon Public Cloud services.

2.4 Model Formal Definition

In the new model, each microservice is modelled through a component that can be connected with others using different ports. These ports represent the required and provided functionalities described through different interfaces that are used as names for them. As described in the previous section, the requirements are divided into strong and weak and resources/costs are considered, too.
In the following definitions disjoint sets will be used: $\mathcal{I}$ for interfaces, $\mathcal{Z}$ for microservices, and a finite set $\mathcal{R}$ for kinds of resources. $\mathbb{N}$ will be used to denote natural numbers, $\mathbb{N}^+$ for $\mathbb{N} \setminus \{0\}$, and $\mathbb{N}^+_\infty$ for $\mathbb{N}^+ \cup \{\infty\}$.

Definition 1 (**Microservice type**). *A Microservice type $\mathcal{T}$ is a 5-ples $\langle P, D_s, D_w, C, R \rangle$ where:*

- $P = (\mathcal{I} \rightharpoonup \mathbb{N}^+_\infty)$ *are the provided interfaces, defined as a partial function from interfaces to corresponding numerical constraints (indicating the maximum number of allowed connections);*

- $D_s = (\mathcal{I} \rightharpoonup \mathbb{N}^+)$ *are the strong required interfaces, defined as a partial function from interfaces to corresponding numerical constraints (indicating the minimum number of necessary connections);*

- $D_w = (\mathcal{I} \nrightarrow \mathbb{N})$ *are the weak required interfaces (defined in the same way as the strong ones, with the difference that the value 0 can also be used, to indicate that it is not strictly necessary to connect microservices);*

- $C \subseteq \mathcal{I}$ *are the conflicting interfaces. A conflict implies that the interfaces cannot be active in other components' provided ports with the same interface;*

- $R = (\mathcal{R} \rightarrow \mathbb{N})$ *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.* [1]
Microservice types are grouped in the set Γ*, ranged over by* $\mathcal{T}_1, \mathcal{T}_2, \ldots$

A Microservice type $\mathcal{T}$ can be accessed as a record using abbreviation as $\mathtt{.prov}$, $\mathtt{.req}_\mathtt{S}$, $\mathtt{.req}_\mathtt{W}$, $\mathtt{.conf}$ and $\mathtt{.res}$.
To better understand how to use the different fields of a microservice type, an example is provided. $\mathcal{T}\mathtt{.res}$ returns the partial function used to describe resources consumption. So this function takes a resource type as input and returns the corresponding amount required. Examples can be made with reference to the configuration showed in Figure 2.3:

- $\textsf{Message Receiver}\mathtt{.res}(\textsf{RAM}) = 4$,

- $\textsf{Message Analyzer}\mathtt{.req}_\mathtt{S}(\textsf{AA}) = 1$,

- $\textsf{Attachment Analyzer}\mathtt{.prov}(\textsf{AA}) = 2$.

The default values for numerical constraints on provided/required ports, when no numbers are specified, are: ∞ for provided interfaces and 1 for required interfaces. These values mean that if no value is explicitly assigned to a port then:

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

- a provided port can be connected with an unlimited amount of instances, and
- a single connection is enough to satisfy a required port.

As in Aeolus, this model offers conflict interfaces. They can be used to avoid the presence at the same time in the system of two instances with a conflict. They can be two instances of the same microservice or two instances of different microservices that cannot be active together.

To manage the deployment of configuration with circular dependency in Aeolus, the *Multiple state change* action was initially introduced. In this model, configurations with strongly circular dependency are not allowed. This means that at least one weak port must be involved in the cycle. To formalize this constraint, a well-formedness condition on microservice types is introduced.

Definition 2 (**Well-formed Universe**). *Given a finite set of microservice types U (that can be called universe), the strong dependency graph of U is as follows: $G(U) = (U, V)$ with $V = \{(\mathcal{T}, \mathcal{T}') | \mathcal{T}, \mathcal{T}' \in U \wedge \exists p \in \mathcal{I}.p \in \mathtt{dom}(\mathcal{T}.\mathtt{reqs}_S) \cap \mathtt{dom}(\mathcal{T}'.\mathtt{prov})\}$. The universe U is well-formed if $G(U)$ is acyclic.*

The *Well-formed Universe* is a necessary pre-condition and it is always assumed. It is not a limitation because circular dependencies are possible by inserting at least one weak interface in the cycle.

Definition 3 (**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 $R = (\mathcal{R} \rightarrow \mathbb{N})$ that specifies node resource availability: $o.\mathtt{res}$ is used to denote such a function.*
- *A value in $\mathbb{N}$ that specifies node cost: $o.\mathtt{cost}$ is used to denote such a value.*

As example, in Figure 2.3, there are 4 nodes: Node1_large, Node2_xlarge, Node3_xlarge, Node4_large. The nodes of large type can be described as:

- Node1_large.$\texttt{res}$(RAM)$=4$,
- Node1_large.$\texttt{res}$(CPU)$=2$,
- Node1_large.$\texttt{cost} = 100$,

while nodes of xlarge type are described as:

- Node2_xlarge.$\texttt{res}$(RAM)$=8$,
- Node2_xlarge.$\texttt{res}$(CPU)$=4$,
- Node2_xlarge.$\texttt{cost} = 199$.

So new nodes can be defined providing their names, the amounts of each type of resource offered and the corresponding costs.

A deployed system can be described through a configuration. This provides information about the possible microservice types, the set of deployed instances, their location on computation nodes and the connections between them. Formally:

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

- *$Z \subseteq \mathcal{Z}$ is the set of the currently deployed microservice instances;*
- *$T = (Z \to \mathcal{T})$ are the microservice types, defined as a function from deployed instances to corresponding microservice types;*
- *$N = (Z \to \mathcal{N})$ are the microservice nodes, defined as a function from deployed instances 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 instance that requires that interface, and the microservice instance that provides it; we assume that, for $(p, z_1, z_2) \in B$, the two instances z_1 and z_2 are distinct and $p \in (\mathtt{dom}(T(z_1).\mathtt{req_S}) \cup \mathtt{dom}(T(z_1).\mathtt{req_W})) \cap \mathtt{dom}(T(z_2).\mathtt{prov})$.*

Using again the environment showed in Figure 2.3, instances location and bindings can be better understood through some examples:

- **Bindings**: using mr1 to refer solely to the instance of Message Receiver, and ma1 for the first deployed instance of Message Analyzer; the connection between them can be represented through (MA,mr1,ma1). In a similar way, referring to the first deployed instance of Attachment Analyzer through aa1 another element of B is (AA,ma1,aa1).

- **Location**: using the names introduced in the previous point to refer to instances, the function N can be used to individuate the instance locations:

  - N(mr1)= Node1_large,
  - N(ma1)= Node2_xlarge,
  - N(aa1)= Node2_xlarge.

The configuration definition does not consider the correctness aspects.

To introduce the latter, two definitions of correct configuration will be given. The first one considers only strong interfaces. It must always be respected, including during the internal steps of a reconfiguration process. The second one considers the weak interfaces, and it must be satisfied when a final configuration is obtained.

Definition 5 (Provisionally correct configuration)**.** *A configuration $\mathcal{C} =$*

$\langle Z, T, N, B\rangle$ is provisionally correct if, for each node $o \in \texttt{ran}(N)$[2]*, it holds*

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

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

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

The first constraint ensures that each node has enough resources of each considered type to satisfy the requests of all the hosted instances. The second one guarantees that there are enough connections from different instances to satisfy the numerical constraint of all strong required interfaces. The third one in a dual mode ensures that numerical constraint on each provided port is respected, so that each provided port is not connected more than the allowed number of times.

Definition 6 (Correct configuration). *A configuration $\mathcal{C} = \langle Z, T, N, B\rangle$ is correct if $\mathcal{C}$ is provisionally correct and, for each microservice $z \in Z$, both following conditions hold:*

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

To have a correct configuration, it firstly has to be provisionally correct and in addition to this two more conditions are considered. The first one ensures that the numerical constraints on weak interfaces are satisfied with

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

at least the minimum number of bindings requested. The second one introduces the conflicts, ensuring that if a conflict is specified for an interface p, this interface is not provided by other deployed instances.
Analysing the correctness of example in Figure 2.3, it is possible to observe that the initial configuration (in continuous lines) is only provisionally correct because the constraints on resources, strong required interfaces and provided interfaces are respected, but it is not completely correct because the constraint ≥ 3 on weak required interface MA of the Message Receiver instance is not satisfied (there is only one binding). Considering the final configuration including the components in dotted lines, the completely correctness is reached. The resource constraints remain satisfied, the constraints on strong required, provided and weak required interfaces are satisfied and there are no conflicts.

To pass from a configuration to a new one, it is necessary to formalize possible actions. The model provides four possible atomic actions that can be combined to pass from an initial to a final configuration.

Definition 7 (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 port p (which is supposed to be a weak-required port of z_1 and a provided port 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 provided port of z_2);*

- *new$(z, \mathcal{T}, o, B_s)$ where $z \in \mathcal{Z}$, $\mathcal{T} \in \Gamma$, $o \in \mathcal{N}$ and $B_s = (\mathit{dom}(\mathcal{T}.\mathtt{req_S}) \rightarrow 2^{\mathcal{Z}-\{z\}})$; with B_s (representing bindings from strong required interfaces in $\mathcal{T}$ to sets of microservices) being such that, for each $p \in \mathit{dom}(\mathcal{T}.\mathtt{req_S})$, it holds $|B_s(p)| \geq \mathcal{T}.\mathtt{req_S}(p)$: add a new microservice instance z of type $\mathcal{T}$ hosted in node o and bind each of its strong required interfaces to a* *set of microservices as described by B_s;* [3]

- *$del(z)$ where $z \in \mathcal{Z}$: remove the microservice instance z from the configuration and all bindings involving it.*

The most difficult action to understand is certainly the $new(z, \mathcal{T}, o, B_s)$ one. It is possible to provide an usage example again through the system showed in Figure 2.3 and the abbreviations already discussed for it. In particular to deploy the first Message Analyzer instances ma1, it is necessary to insert the instruction new(ma1,Message Analyzer, Node2_xlarge, (AA $\mapsto$ {aa1})) in the deployment plan. The connection between ma1 and aa1 must be immediately established because it involves a strong interface.

The effects of an action on a configuration can be formalized using a labelled transition system on configurations, which uses actions as labels.

Definition 8 (Reconfigurations)**.** *Reconfigurations are denoted by transitions $\mathcal{C} \xrightarrow{\alpha} \mathcal{C}'$ meaning that the execution of $\alpha \in \mathcal{A}$ on the configuration $\mathcal{C}$ produces a new configuration $\mathcal{C}'$. The possible transitions from a configuration $\mathcal{C} = \langle Z, T, N, B \rangle$ are defined as follows:*

$$\begin{array}{l}
\mathcal{C} \xrightarrow{bind(p,z_1,z_2)} \langle Z, T, N, B \cup \langle p, z_1, z_2 \rangle \rangle \\
\quad \textit{if } \langle p, z_1, z_2 \rangle \notin B \textit{ and} \\
\quad p \in \mathtt{dom}(T(z_1).\mathtt{req_w}) \cap \mathtt{dom}(T(z_2).\mathtt{prov})
\end{array}$$

$$\begin{array}{l}
\mathcal{C} \xrightarrow{unbind(p,z_1,z_2)} \langle Z, T, N, B \backslash \langle p, z_1, z_2 \rangle \rangle \\
\quad \textit{if } \langle p, z_1, z_2 \rangle \in B \textit{ and} \\
\quad p \in \mathtt{dom}(T(z_1).\mathtt{req_w}) \cap \mathtt{dom}(T(z_2).\mathtt{prov})
\end{array}$$

$$\begin{array}{l}
\mathcal{C} \xrightarrow{new(z,\mathcal{T},o,B_s)} \langle Z \cup \{z\}, T', N', B' \rangle \\
\quad \textit{if } z \notin Z \textit{ and} \\
\quad \forall p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_s}).\ \forall z' \in B_s(p). \\
\qquad p \in \mathtt{dom}(T(z').\mathtt{prov}) \textit{ and} \\
\quad T' = T \cup \{(z \mapsto \mathcal{T})\} \textit{ and} \\
\quad N' = N \cup \{(z \mapsto o)\} \textit{ and} \\
\quad B' = B \cup \{\langle p, z, z' \rangle \mid z' \in B_s(p)\}
\end{array}$$

$$\begin{array}{l}
\mathcal{C} \xrightarrow{del(z)} \langle Z \backslash \{z\}, T', N', B' \rangle \\
\quad \textit{if } T' = \{(z' \mapsto \mathcal{T}) \in T \mid z \neq z'\} \textit{ and} \\
\quad N' = \{(z' \mapsto o) \in N \mid z \neq z'\} \textit{ and} \\
\quad B' = \{\langle p, z_1, z_2 \rangle \in B \mid z \notin \{z_1, z_2\}\}
\end{array}$$

Formalizing the possible actions and their individual effects on a configuration, it is possible to define a *deployment plan*. Intuitively, this is a

[3] 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.

sequence of actions that allow to pass from an initial configuration to a final target configuration. The plan ensures that the correctness is respected, or rather, all the intermediate configurations are always provisionally correct and the final one is completely correct.

Definition 9 (Deployment plan). *A deployment plan* P *from a provisionally correct configuration* $\mathcal{C}_0$ *is a sequence of actions* $\alpha_1, \ldots, \alpha_m$ *such that:*

- *there exist* $\mathcal{C}_1, \ldots, \mathcal{C}_m$ *provisionally correct configurations, with* $\mathcal{C}_{i-1} \xrightarrow{\alpha_i} \mathcal{C}_i$ *for* $1 \leq i \leq m$*, and*
- $\mathcal{C}_m$ *is a correct configuration.*

Deployment plans are also denoted with $\mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$*.*

A Deployment Plan example can be provided by describing the one necessary to pass from the initial provisionally correct configuration represented with continuous lines in Figure 2.3 to the final correct configuration where the components in dotted lines are deployed.

new(aa2, Attachment Analyzer, Node3_xlarge, ())
new(ma2, Message Analyzer, Node4_large, (AA $\mapsto$ {aa1}))
new(ma3, Message Analyzer, Node3_xlarge, (AA $\mapsto$ {aa2}))
bind (MA, mr1, ma2)
bind (MA, mr1, ma3)

Using the previous definitions it is now possible to formally define the *optimal deployment problem.* Given:

- the description of all possible microservices, called *universe of microservice types*,
- a set of available computation nodes that can be used to host new instances and
- an initial configuration,

the problem consists in finding whether it is possible to reach a correct configuration with at least one instance of a given microservice type $\mathcal{T}$ and, in that case, how this can be done optimizing the overall cost of the used nodes.

Definition 10 (Optimal deployment problem). *The optimal deployment problem has as inputs a finite well-formed universe U of microservice types, a finite set of available nodes O, an initial provisionally correct configuration $\mathcal{C}_0$ and a microservice type $\mathcal{T}_t \in U$. The output is:*

- *If there exists one, a **deployment plan** $\mathsf{P} = \mathcal{C}_0 \xrightarrow{\alpha_1} \mathcal{C}_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_m} \mathcal{C}_m$ such that*
  - *for all $\mathcal{C}_i = \langle Z_i, T_i, N_i, B_i \rangle$, 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*
  - *$\mathcal{C}_m = \langle Z_m, T_m, N_m, B_m \rangle$ satisfies $\exists z \in Z_m : T_i(z) = \mathcal{T}_t$.*

  In particular, among all deployment plans satisfying the constraints above, one that minimizes $\sum_{o \in O.(\exists z.N_m(z)=o)} o.\texttt{cost}$ (i.e., the overall cost of nodes in the last configuration $\mathcal{C}_m$), is outputted.

- *otherwise, **no** (stating that no such plan exists).*

The problem has the inputs previously described and returns a deployment plan as output, if this exists, otherwise the message ”no”. For the deployment plan it is guaranteed that:

- in all configurations passed through, all microservice types and all computation nodes used are allowed,
- the final configuration contains at least an instance of the requested microservice type, and
- the total cost, calculated as the sum of the costs of all nodes used (that host at least an instance), is the minimum possible.

2.5 Decidability Proof

In this section the proof that shows the decidability of the analysed problem for microservices will be described. As mentioned in the corresponding section, microservices are loosely coupled services that can be **independently** developed and deployed. These characteristics allow to transform the undecidable problem showed in Chapter 1 in the decidable problem described in this Chapter. The decidability is proved providing an algorithm, based on constraint programming, organized in three phases:

1. the first one defines and solves a set of constraints whose solution indicates which microservices must be deployed and in which nodes they have to be hosted,

2. the second one defines and solves a set of constraints whose solution indicates how to connect the different instances,

3. and the last one uses the information obtained by the previous phases to synthesize the necessary deployment plan to obtain the defined configuration.

The defined constraints also manage the optimization process to ensure that the solution is the optimal one. In particular, it uses one or more optimization metrics to minimize the overall final configuration costs.
So, it is possible to formalize a theorem that captures the described goal and after that the corresponding proof will be presented.

Theorem 1. *The optimal deployment problem is decidable.*

Proof. As already mentioned, the proof follows a constructive approach providing an algorithm that solves the optimal deployment problem. The input of this algorithm can be summarised as:

- U, the set of microservice types,

- O, the set of available computation nodes,

- $\mathcal{C}_0$, the initial configuration that is ensured to be provisionally correct, and

- $\mathcal{T}_t \in U$, the target microservice type for which the user wants at least an instance in the final configuration.

In addition to these pieces of information, it is possible to pre-calculate the set of interfaces required or provided by microservices,
$\mathcal{I}(U) = \bigcup_{\mathcal{T} \in U} \texttt{dom}(\mathcal{T}\texttt{.req}_\texttt{S}) \cup \texttt{dom}(\mathcal{T}\texttt{.req}_\texttt{W}) \cup \texttt{dom}(\mathcal{T}\texttt{.prov}) \cup \mathcal{T}\texttt{.conf}.$
Starting from these data, the algorithm is structured in three consecutive phases.

Phase 1 As briefly described above, the first phase uses a set of constraints to check if a possible solution to the provided optimal deployment problem exists and, if that is the case, the solution reports:

1. the number of instances that have to be created during the deployment plan for each microservice type $\mathcal{T}$ (denoted with $\texttt{inst}(\mathcal{T})$),

2. the number of instances of each microservice type that have to be deployed on node o (denoted with $\texttt{inst}(\mathcal{T}, o)$),

3. the number of bindings that have to be established for each weak or strong interface p from instances of type $\mathcal{T}$ and instances of type $\mathcal{T}'$ (denoted with $\texttt{bind}(p, \mathcal{T}, \mathcal{T}')$).

At the end, in addition to the involved constraints, a minimizing optimization function is also introduced to guarantee that the solution proposed is the optimal one. Usually, this is the configuration's total cost if it is requested to minimize the costs of computation nodes used to obtain the final configuration.
The cited constraints will be presented in three different logical group to help the reader understand them. The first group concerns the number of

bindings:

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U,\, p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_S})} \mathcal{T}.\mathtt{req_S}(p) \cdot \mathtt{inst}(\mathcal{T}) \leq \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}, \mathcal{T}') \tag{2.1a}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U,\, p \in \mathtt{dom}(\mathcal{T}.\mathtt{req_W})} \mathcal{T}.\mathtt{req_W}(p) \cdot \mathtt{inst}(\mathcal{T}) \leq \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}, \mathcal{T}') \tag{2.1b}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U,\, \mathcal{T}.\mathtt{prov}(p) < \infty} \mathcal{T}.\mathtt{prov}(p) \cdot \mathtt{inst}(\mathcal{T}) \geq \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}', \mathcal{T}) \tag{2.1c}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U,\, \mathcal{T}.\mathtt{prov}(p) = \infty} \mathtt{inst}(\mathcal{T}) = 0 \;\; \Rightarrow \;\; \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}', \mathcal{T}) = 0 \tag{2.1d}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U,\, p \notin \mathtt{dom}(\mathcal{T}.\mathtt{prov})} \sum_{\mathcal{T}' \in U} \mathtt{bind}(p, \mathcal{T}', \mathcal{T}) = 0 \tag{2.1e}$$

The first three expressions guarantee that numerical constraints on interfaces are respected. In particular, 2.1a and 2.1b ensure that there are enough bindings to satisfy all the strong and weak required interfaces. Symmetrically, constraint 2.1c guarantees that the provided ports, with bounded capacities, are not overused. This is obtained requesting that the capacity of instances of a type on an interface, calculated as the sum of the single capacities, is greater than the number of bindings where the instances of that type are used as provider. The next one, 2.1d, instead takes into consideration the unbound provided ports (with the default value ∞). It specifies that, if there is no instance of a microservice type, it can not be used as provider in a binding, but otherwise if an instance is available, this is enough to satisfy any possible requirement on the corresponding interface. The last one, constraint 2.1e guarantees that if a microservice type is used as provider, it actually has a provided port for the corresponding interface.

The second group concerns the number of microservice instances that have to be deployed:

$$\mathtt{inst}(\mathcal{T}_t) \geq 1 \tag{2.2a}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\substack{\mathcal{T} \in U, \\ p \in \mathcal{T}.\mathtt{conf}}} \quad \bigwedge_{\substack{\mathcal{T}' \in U - \{\mathcal{T}\}, \\ p \in \mathtt{dom}(\mathcal{T}'.\mathtt{prov})}} \mathtt{inst}(\mathcal{T}) > 0 \;\; \Rightarrow \;\; \mathtt{inst}(\mathcal{T}') = 0 \tag{2.2b}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\substack{\mathcal{T} \in U,\, p \in \mathcal{T}.\mathtt{conf} \,\wedge \\ p \in \mathtt{dom}(\mathcal{T}.\mathtt{prov})}} \mathtt{inst}(\mathcal{T}) \leq 1 \tag{2.2c}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U} \quad \bigwedge_{\mathcal{T}' \in U - \{\mathcal{T}\}} \mathtt{bind}(p, \mathcal{T}, \mathcal{T}') \leq \mathtt{inst}(\mathcal{T}) \cdot \mathtt{inst}(\mathcal{T}') \tag{2.2d}$$

$$\bigwedge_{p \in \mathcal{I}(U)} \quad \bigwedge_{\mathcal{T} \in U} \mathtt{bind}(p, \mathcal{T}, \mathcal{T}) \leq \mathtt{inst}(\mathcal{T}) \cdot (\mathtt{inst}(\mathcal{T}) - 1) \tag{2.2e}$$

The first expression, 2.2a, guarantees that the final configuration contains at least an instance of the requested microservice type as defined in the optimal deployment problem. The second constraint, 2.2b, ensures that if there is an instance of a microservice type with a conflict with a particular interface, no instances of different types that provide that interface can be deployed at the same time. The third one addresses again the conflict aspects but considering instances of a same type. In particular, the situation where an instance has a conflict with, and at the same time provides, an interface. In this case at most an instance of that type can be deployed at the same time, as requested by the cited constraint. The last two constraints, 2.2d and 2.2e control that there are enough pairs of distinct instances to establish all the necessary bindings identified by the first group of constraint. 2.2d considers bindings between instances of different types while 2.2e between instances of the same type.

The last group concerns the distribution of microservice instances over the available computation nodes O:

$$\texttt{inst}(\mathcal{T}) = \sum_{o \in O} \texttt{inst}(\mathcal{T}, o) \tag{2.3a}$$

$$\bigwedge_{r \in \mathcal{R}} \bigwedge_{o \in O} \sum_{\mathcal{T} \in U} \texttt{inst}(\mathcal{T}, o) \cdot \mathcal{T}.\texttt{res}(r) \leq o.\texttt{res}(r) \tag{2.3b}$$

$$\bigwedge_{o \in O} \Big(\sum_{\mathcal{T} \in U} \texttt{inst}(\mathcal{T}, o) > 0 \Big) \Leftrightarrow \texttt{used}(o) \tag{2.3c}$$

$$\min \sum_{o \in O,\, \texttt{used}(o)} o.\texttt{cost} \tag{2.3d}$$

The first expression, 2.3a, defines the value of $\texttt{inst}(\mathcal{T})$ as the sum of all the instances of type $\mathcal{T}$ deployed on the used computation nodes. The second constraint, 2.3b, ensures the correctness of resource usage. In particular, it guarantees that each node has enough resources (of each type) to satisfy the requirements of all the hosted instances. The last two constraints define the optimization function used to obtain the optimal solution. The last expression, 2.3d, minimizes the sum of the costs of the computation nodes used, that is the main user's goal considered. A computation node is used if it hosts at least a microservice instance. Formally, to understand if a node is used, a boolean variable called used(o) is defined by constraint 2.3c. It is true if and only if the corresponding node hosts at least an instance of any kind within the allowed types.

The formalization through constraints allows to use a constraint/optimization solver to search a solution. All the constraints described in Phase 1 become the input of the adopted solver. If it returns a solution, the algorithm proposed can proceed with the next phase, otherwise the required microservice system, described with the set of constraints, cannot be deployed.

Phase 2 If Phase 1 admits a solution, it means that the algorithm knows how many instances of each microservice type have to be deployed, where they have to be deployed and how many bindings between instances of a type and instances of another have to be created. In this second phase the connections between the various instances computed in the previous phase are specifically decided. This operation is done again using a set of constraints. To individually identify the instances of each type a new notation is introduced: $s_i^{\mathcal{T}}$, with $1 \leq i \leq \texttt{inst}(\mathcal{T})$. $s_i^{\mathcal{T}}$ indicates the i-th instance of type $\mathcal{T}$. Clearly, it can be used only for microservice types with at least an instance computed in Phase 1, or rather, such that $\texttt{inst}(\mathcal{T}) > 0$.
Boolean variables are introduced to identify the presence of a binding between two particular instances. These variables are formalized as $\texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'})$, considering that if $\mathcal{T} = \mathcal{T}'$, or rather, a connection between two instances of a same type has to be created, then is necessary that $i \neq j$, because an instance cannot be connected with itself.

- $\texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) = 1$ means that there is a connection between the required port p of $s_i^{\mathcal{T}}$ and the provided port p of $s_j^{\mathcal{T}'}$,
- $\texttt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) = 0$, otherwise

The values n and m are used to denote $\texttt{inst}(\mathcal{T})$ and $\texttt{inst}(\mathcal{T}')$, respectively, or rather, the total number of instances of types $\mathcal{T}$ and $\mathcal{T}$'. Also an auxiliary total function *limProv*$(\mathcal{T}', p)$ that extends $\mathcal{T}'$.prov associating 0 to interfaces outside its domain, is introduced. In this way, *limProv*$(\mathcal{T}', p)$ allows to call $\mathcal{T}'$.prov on all the possible interfaces and not only on those provided by that microservice types. If the specified interfaces are not effectively provided the value returned is 0 that means that it cannot be used in practice.

The constraints formalized in this second phase are:

$$\bigwedge_{\mathcal{T}\in U} \bigwedge_{p\in\mathcal{I}(U)} \bigwedge_{i\in 1\dots n} \sum_{j\in(1\dots m)\setminus\{i|\mathcal{T}=\mathcal{T}'\}} \mathtt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \leq \mathit{limProv}(\mathcal{T}', p) \tag{2.4a}$$

$$\bigwedge_{\mathcal{T}\in U} \bigwedge_{p\in\mathtt{dom}(\mathcal{T}.\mathtt{req_S})} \bigwedge_{i\in 1\dots n} \sum_{j\in(1\dots m)\setminus\{i|\mathcal{T}=\mathcal{T}'\}} \mathtt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \geq \mathcal{T}.\mathtt{req_S}(p) \tag{2.4b}$$

$$\bigwedge_{\mathcal{T}\in U} \bigwedge_{p\in\mathtt{dom}(\mathcal{T}.\mathtt{req_W})} \bigwedge_{i\in 1\dots n} \sum_{j\in(1\dots m)\setminus\{i|\mathcal{T}=\mathcal{T}'\}} \mathtt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) \geq \mathcal{T}.\mathtt{req_W}(p) \tag{2.4c}$$

$$\bigwedge_{\mathcal{T}\in U} \bigwedge_{p\notin\mathtt{dom}(\mathcal{T}.\mathtt{req_S})\cup\mathtt{dom}(\mathcal{T}.\mathtt{req_W})} \bigwedge_{i\in 1\dots n} \sum_{j\in(1\dots m)\setminus\{i|\mathcal{T}=\mathcal{T}'\}} \mathtt{b}(p, s_i^{\mathcal{T}}, s_j^{\mathcal{T}'}) = 0 \tag{2.4d}$$

The first constraint fixes an upper bound to the bindings that can be created with each provided port using the numerical value associated by the corresponding type to the selected provided interface. The second and the third expressions fixes, in a dual mode, lower bounds for the strong (2.4b) and weak (2.4c) required interfaces, guaranteeing that they will be connected with a sufficient amount of provided ports. Finally, the last constraint ensures that no unnecessary bindings will be established, or rather, connections on interfaces that are not required will be avoided.

If a solution for Phase 1 exists, then also a solution for the constraints described just now in Phase 2 exists. Because the constraints solved in Phase 1 guarantees that in the configuration obtained there are enough provided ports, with enough capacities, to satisfy all the strong and weak required ports. This result is formally shown in [45].

Phase 3 In this last phase of the resolution algorithm, the information about instances, their locations and bindings obtained in the previous phases are used to synthetize a deployment plan. The deployment plan is the solution searched and, in particular, it can be applied to the initial configuration provided to obtain a new configuration that satisfies the user needs.

To simplify the proof and without loss of generality, in this formal demonstration, the searched deployment plan is obtained un-deploying all the instances in the initial configuration, and after that, deploying from scratch the target configuration. However, in practice it is possible to re-use the components

already deployed in the initial configuration to implement an incremental solution that is more efficient.
So in the proof the deployment plan is obtained through two phases.

1. The first step requires to reach an empty configuration. This goal can be easily obtained starting with a sequence of *unbind* actions for all connections on weak required interfaces. Then, it is possible to begin the instances deletion in a safe way. The process first removes the instances that have not bindings on provided port, because consequently they can be removed without disconnecting strong required interfaces on other components losing the configuration correctness (weak required ports have already been disconnected by previously *unbind* operations). Then, the process repeats these operations until all the instances have been deleted. This approach is possible thanks to the well-formedness assumption (Definition 2). Indeed, it is possible to topologically sort the configuration, ordering the instances to be removed so as to guarantee that no strong required interfaces will be violated. Intuitively, the un-deploy order is the opposite of the deploy order used to obtain the initial configuration.

2. The second step requires to reach the target configuration from scratch. The deployment operations for the instances computed and distributed over the nodes in *Phase 1* and connected in *Phase 2* can be executed following a topological sort considering the microservices dependencies following from the strong required interfaces. Intuitively, the process begins deploying instances without strong required interfaces (that can be correctly deployed) and, only when all the necessary provided ports are available, deploying components with strong requirements. At the end, when all the instances are deployed (it means that all the strong required interfaces are correctly satisfied) a sequence of *binding* operations are inserted to connect weak required ports in any possible order.

□

Possible Extensions It is possible to extend the set of constraints defined in the showed proof to obtain additional advantages.
A first example can be obtained considering that usually it is better to connect instances hosted in a same computation node to allow them to communicate locally without using the network. To do that, it is enough to add an optimization metric to maximize the number of local bindings. In the following expression used to formally represents this goal, a function N that returns the location of a microservice instance is used.

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

Another additional constraint that can be inserted is a metric to maximize the number of bindings. For example, it is useful if the load balancers are modelled as components where the back-ends are connected through a weak required interface. Maximizing the bindings means that in the synthesized configuration all the possible services that can satisfy the interface required by the load balancer, will be connected with it to obtain the wanted behaviour.

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

2.6 Complexity Analysis

The algorithm proposed in the previous section proves the decidability of the analysed problem. In this section its complexity will be studied. It is possible to show that *Phase 1* and *Phase 2* are in NP, while *Phase 3* can be solved in polynomial time. Observing that numeric constraints can be represented in log space, the output of *Phase 2* could be exponential in the size of output of *Phase 1*. This is because the output of *Phase 1* indicates only the total number of instances for each type while the output of *Phase 2* requires the enumeration of all the instances that have to be deployed. In other words, the length of the deployment plan, that is the algorithm output, could be exponential with respect to the input size. For this reason in the worst case the algorithm has a NEXPTIME complexity.

But considering that the number of microservices hostable in a single node in a real situation is clearly strongly limited by the amount of its resources, it makes sense to assume that a node can host only a polynomial amount of microservices. Using this assumption the worst case can be considered unfeasible in practice. Nevertheless, the decision version of the problem remains NP-complete and consequently to obtain the optimal plan it is necessary to solve an NP-optimization problem.

This can be better faced thank to the choice of develop the algorithm using a constraint programming paradigm. In fact, it allows to exploit state-of-the-art constraint solvers [41, 42, 43] that are frequently used to solve NP-hard problems.

A formal proof of the algorithm complexity discussed in this section can be found on [15].

Chapter 3

Case-Study: Model of an Email Processing Pipeline Microservice Architecture

In addition to the theoretical outcomes described in Chapter 2, a small model of a real-world microservices application has been realized to evaluate the applicability of the proposed approach in spite of the algorithm complexity (NP under the assumption of polynomial size of the target configuration). It simulates an email processing pipeline inspired by Iron.io and described in [2].
It is developed using the Abstract Behavioral Specification (ABS) language, a high-level object-oriented language that supports deployment modelling. ABS is agnostic with respect to deployment platforms (e.g., Amazon AWS, Microsoft Azure) and technologies (e.g., Docker or Kubernetes) and offers high-level deployment primitives to create *deployment components* and to instantiate objects inside them. The deployment and scaling plans are obtained through a tool called SmartDepl. It is an ABS extension that offers the possibility to enrich the developed code with the required information to specify the current instance of the optimal deployment problem. In particular, this information includes: the computing resources offered by computation

nodes and required by each software component, the costs of those nodes, the dependencies between components and the deployment constraints – which usually drive the decisions taken by the operations' professionals – provided through declarative expressions. Inside, SmartDepl uses Zephyurs2, which is a configuration optimizer. It takes the user requirements and a universe of components, and computes the optimal configuration guaranteeing that user needs are satisfied. The Zephyurs2 result is parsed by SmartDepl to create an ABS module that codifies the computed optimal configuration plan. In particular, the generated ABS code reports the allocation of components and how to connect them.

For the proposed case-study, a simple deployment plan with one instance for each microservice is considered, and three horizontally scaling plans with different dimensions have been realized. The scaling plans differ for the amount of inbound requests expected (small, medium or large increments in the number of received emails). The experimental results reached in this case-study are encouraging. The tool is able to compute deployment or scaling plans that allow to OPTIMALLY deploy more than 30 new instances considering hundreds of available machines of 3 different types.

Then, additional auxiliary ABS classes are developed and a specific ABS feature called Timed-ABS are used to implement a simple simulation. This allows to observe the behaviour of the system while the calculated deployment and scaling plans are applied on variations of the number of emails received.

3.1 Email Processing Pipeline Architecture

The system modelled is an Email Processing tool. The architecture contains 12 different microservices and the same amount of load balancers. It is organized as a pipeline composed by distinct parallel sub-pipelines, each of which analyses a different email component. In the realized model, four sub-pipelines are considered dividing an email in the same amount of com-

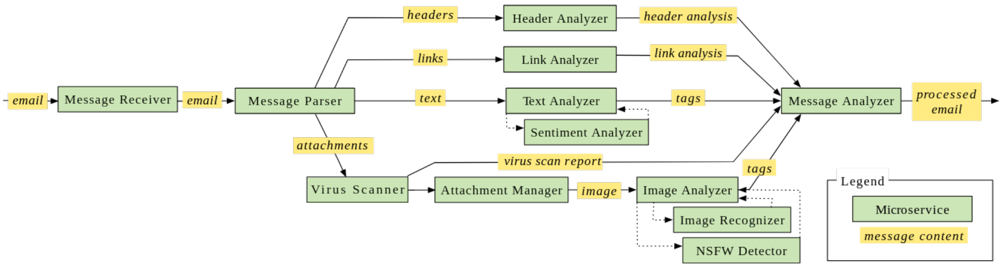

Figure 3.1: Microservice architecture for email processing

ponents:

1. headers,
2. set of links,
3. text, and
4. set of attachments.

Other branches can be easily introduced if necessary. For the attachments only a path for images has been considered, but alternative ways can be easily added in this case as well to specifically analyse different types of documents. A simple representation of the realized model is showed in Figure 3.1. This representation only contains the main microservices, while load balancers, a DB service and other auxiliary components are expressly excluded to favour the clarity. The processing flow can be easily understood observing the image.

1. The entry-point of the system is the Message Receiver, it is the only component without a load balancer in front of and it simply receives emails and forwards them to the Message Parser.

2. Message Parser extracts data from the incoming mail, parsing the different planned fields. It also generates a unique message id. Before starting the different analysis, it stores some information about that message in the DB (using the specific microservice) to know that it is

being analysed. After that, it sends the various mail components to the corresponding microservices.

(a) Header Analyser receives the mail headers, analyses them and sends the obtained result to Message Analyser.

(b) Link Analyser receives the set of links, analyses each of them and sends the obtained result to Message Analyser.

(c) Text Analyser receives the mail text, divided in message header and message body, analyses them and sends the obtained result to Message Analyser. The message body analysis, in addition to the normal operations expected, exploits the Sentiment Analyser functionalities. It divides the text into blocks and does a sequence of request-response calls to obtain a sentiment analysis.

(d) Virus Scanner receives an attachment, controls if it contains a virus: in this case it sends directly a warn to Message Analyser, otherwise the attachment proceeds on the expected path.
An attachment without virus is sent to Attachment Manager that decides its type and sends it in the correct analysis process. In our example only images are considered but other possible paths can be introduced.
An image is received by Image Analyser that analyses it and sends the obtained result to Message Analyser. The image analysis, in addition to the normal operations expected, does two request-response calls to NSFW Detector and Image Recognizer. The first one is a ”Not Safe for Work Detector” and returns a boolean value with an opinion about the image, while the second returns a string that describes what the selected image represents.

3. All the single sub-analysis results are sent to the Message Analyser that inserts them on the DB and checks if all the expected information have been loaded. In this case, it recovers all the data for the corresponding message from the DB using the message id and produces a final total

result, which is the email processing pipeline's output. Otherwise, if a sub-pipeline is still analysing an email component, it moves to analyses the next call.

- DB is a microservice that operates as an interface with a database. It allows to insert results of partial analyses linked with a message id or to recover all the data stored for a particular message when its analysis is completed. In this last case, all the returned information is also deleted from the database because the analysis of that message is finished. The DB microservice further offers auxiliary services to monitor the system. In particular, it counts the messages reached and the average analysis duration. A monitor component can exploit these additional services recovering the data at the end of a monitoring window and resetting them for the next one.

In addition to the code specification of each microservice, an estimation on the resources necessary to host an instance of each of them has been performed. It has been realized comparing the intuitive computation load of each microservice. These resource data are inserted with information about dependencies through specific annotations that will be introduced later. The cited annotations are also used to introduce the specification of the available computation nodes. In particular in our case-study, three types of computation nodes, inspired by Amazon EC2 instances, are coded: c4_large, c4_xlarge, and c4_2xlarge.

3.2 Language and Tool

To realize the model, the Abstract Behavioral Specification (ABS) language [5] has been used. The ABS extension SmartDepl [9] is instead used to specify all the additional information and obtain the deployment plans. Indeed, it offers the possibility to annotate the code with information about resources and costs of computation nodes, resources and dependencies of microservices and a declarative description of deployment plans required. Zephyrus2 is the core of SmartDepl and it is an engine used to solve the optimal deployment problem presented in Chapter 2. In this section, these three tools are briefly presented.

3.2.1 Abstract Behavioral Specification (ABS) Language

As mentioned above the ABS language is a high-level, actor-based, object-oriented language. It is designed to create models and execute the corresponding code. ABS offers good features for a programming language as algebraic user-defined data types, side effect-free functions and immutable data. Considering also its formal semantics and compositional proof theory, it is easy to understand that ABS allows to formally rationalise the modelled system and its proprieties. For this purpose and to enable static analysis of the code developed, a variety of tools (deadlock checker, resource analysis, formal verification) have been realized. ABS expressions are divided in pure expressions (side effect-free) or expressions with well-defined side effects. The latter cannot be combined with other expressions to simplify the static analysis of the cited tools and, in general, other theoretical investigation on the code.

The high-level syntax adopted, strongly inspired by Java, is very intuitive. This choice allows not ABS-skilled programmers to understand the majority of the codes and to easily learn how to develop ABS programs. The code is organized in modules that exports and imports definitions.

ABS is based on asynchronous method calls. Futures are used to synchronize and read the returned results. The objects are organized into COGs (Concurrent Object Groups). Each COG runs one process at a time and a cooperative context-switch/scheduling is adopted. Multiple COGs can be executed in parallel. The image showed in Figure 3.2, presented in the ABS manual [4], clarifies this idea. Two possible *new* instructions (`new` and `new local`) are available to instantiate new objects in the current COG or in a new one.

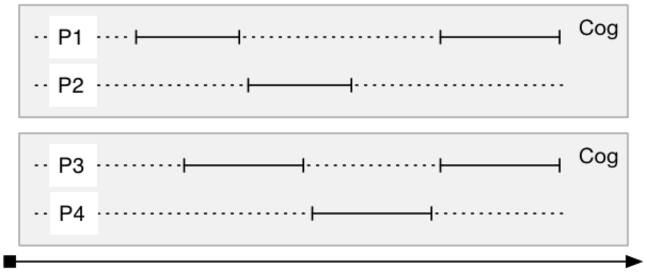

Figure 3.2: ABS objects organization in COGs

Annotations can be written in front of statements and definitions using square brackets ([Annotation Text]) to specify additional information or information used by tools. The SmartDepl annotations presented in the following section and used to provide the necessary information to solve the Optimal Deployment Problem follow this specification.

In the development of the case-study model, an extension to the ABS core, called Timed ABS, has been used. It introduces a notion of *abstract time* that allows to execute simulations studying the timing-related behaviour of the modelled system. In the ABS environment the time does not advance by itself but as a response to specific language instructions. In particular, time only advances when all processes are blocked and no process is ready to be executed. A process can be suspended by classical situations as it waits that a condition becomes true or a result is returned through a Future but also when there are not enough resources to be executed. Indeed the concepts of code deployment on different possible machines, resources and their use can also be introduced in ABS. To do that, three aspects have to be introduced in the model:

- *Locations* where software components are hosted, implemented by **Deployment Components**. They can host COGs and they have a limited amount of resources that can be used by objects contained in the hosted COGs.

- *Costs* to execute operations. They can be specified for each type of resources considered and they are implemented through **Resource Annotations**. Executing an annotated operation implies paying the associated costs.

- *Time* used to reason about advantages or disadvantages obtainable by varying locations and costs. It is implemented with the previously mentioned **Timed ABS**

These elements work together to simulate the Resource Consumption. In particular, each deployment component offers an amount of resources. Processes hosted in that location can execute operations that consume resources until these are available. When the remaining amount of resources is not enough to continue the execution, the hosted processes are stopped until a time unit passes and the deployment component's resources are refilled.
The deployment components are defined, created and managed through the *CloudProvider API.* A provider of machines (in ABS they correspond to deployment components) is modelled through a *CloudProvider* object. Using this object it is possible to specify the offered machines descriptions, providing a name and the offered resources (if no value is provided for a resource type, it is infinite). Then, DeploymentComponent objects can be created requiring an instance of one of the possible machines using its name. This can be done using the *preLaunchInstanceNamed* method offered by CloudProvider objects. Consider an example inspired by the generated code for the analysed case-study:

```
CloudProvider cloudProvider = new CloudProvider("CloudProvider");

cloudProvider.addInstanceDescription( Pair("c4_xlarge",
          map[ Pair(Cores, 4), Pair(Memory, 750) ]));
cloudProvider.addInstanceDescription( Pair("c4_large",
          map[ Pair(Cores, 2), Pair(Memory, 375) ]));
cloudProvider.addInstanceDescription( Pair("c4_2xlarge",
          map[ Pair(Cores, 8), Pair(Memory, 1500) ]));

DeploymentComponent c4_large_0 = cloudProvider.prelaunchInstanceNamed("c4_large");
DeploymentComponent c4_large_1 = cloudProvider.prelaunchInstanceNamed("c4_large");
DeploymentComponent c4_xlarge_0 = cloudProvider.prelaunchInstanceNamed("c4_xlarge");
DeploymentComponent c4_2xlarge_0 = cloudProvider.prelaunchInstanceNamed("c4_2xlarge");
```

As defined above, Deployment Components host COGs. Processes inside COGs deployed in the same Deployment Component share the resources provided by that Deployment Component. To deploy a new COG in a specific Deployment Component, the corresponding definition instruction has to be annotated with a *DC* annotation. Examples inspired again by the case-study generated code are:

```
[DC: c4_large_0] MessageParser_LoadBalancerInterface MessageParser_LoadBalancer_0 =
                              new MessageParser_LoadBalancer();
[DC: c4_large_1] MessageReceiverInterface MessageReceiver_0 =
                              new MessageReceiver(MessageParser_LoadBalancer_0);
[DC: c4_large_2] MessageAnalyser_LoadBalancerInterface MessageAnalyser_LoadBalancer_0 =
                              new MessageAnalyser_LoadBalancer();
[DC: c4_xlarge_0] HeaderAnalyserInterface HeaderAnalyser_0 =
                              new HeaderAnalyser(MessageAnalyser_LoadBalancer_0);
[DC: c4_2xlarge_0] DBInterface DB_0 = new DB();
```

In ABS resources are countable, measurable properties of a deployment component. ABS supports four resource types: Cores, Memory, Bandwidth and Speed. Cores and Memory are static while Bandwidth and Speed are consumed and refilled during the program execution. The first three types are intuitive. The latter type models the execution speed. Speed resources are consumed by execution of instruction annotated with the label Cost. If the amount of Speed available is lower than the annotated cost, the execution takes an observable amount of time. This time is necessary to refill the Speed resource of the Deployment Component reaching a sufficient quantity.

Different back-ends with different characteristics and purposes are available to execute ABS code. In this project the Erlang back-end has been used because it supports the resource models and the time simulation (Real-Time ABS).

Additional information about ABS and the tools offered can be found in the ABS web site [3], in its documentation [4] or in the corresponding paper [5].

3.2.2 SmartDepl

SmartDepl is an ABS extension. It gives the possibility to declaratively specify deployment requirements, abstracting from concrete deployment decision, and by processing them it provides an ABS class that implements the requested plan. The user can use the method *deploy* and *undeploy* offered by generated classes to trigger the deployment execution or undo it if a downscale is required. To specify the necessary information, SmartDepl uses annotations. These can be divided in: annotations employed to describe available computation nodes, annotations linked with ABS classes, where necessary resources and dependencies for the corresponding components are described, and annotations used to present user desiderata. Deployment needs can be generated from both business decisions or technical reasons, and SmartDepl attempts to allow users to capture all of them. Common examples of deployment requirements are:

- A minimum number of instances of each software component to ensure a Service Level Agreement (SLA).

- Services that work with sensitive data cannot be deployed on shared machines.

- For world-wide applications, the used virtual machines should be distributed over different geographical locations to increase the fault-tolerance. Cloud service providers as Amazon or Google easily allow to specify the region and the availability zone wanted during virtual machines initialization. The correct distribution over different geographical areas should be maintained also after scale-up or scale-down operations, requiring particular attentions.

- To improve the network throughput and latency, it could be necessary to specify preferences on bindings between components. For example, local (i.e. inside a single machine or multiple machines in the same zone) connections are usually preferable.

- Co-location requests, when to install an instance of a component, an instance of another is strictly necessary in the same machine.

The annotations values are specified using the JSON format. In the ABS code, they can be inserted with the following tags:

- *[SmartDeployCloudProvider : JSON_String]*: to describe possible computation nodes,
- *[SmartDeployCost : JSON_String]*: to specifies annotation linked with an ABS class, and
- *[SmartDeploy : JSON_String]*: to insert the declarative description of user deployment desiderata.

Annotations used to describe possible nodes types are very simple and intuitive. An example is:

```
{
   "c4_large" : {
      "cost" : 119,
      "payment_interval" : 1,
      "resources" : {
         "Cores" : 2,
         "Memory" : 375
      }
   },
   "c4_xlarge" : {
      "cost" : 237,
      "payment_interval" : 1,
      "resources" : {
         "Cores" : 4,
         "Memory" : 750
      }
   },
   "c4_2xlarge" : {
      "cost" : 476,
      "payment_interval" : 1,
      "resources" : {
        "Cores" : 8,
        "Memory" : 1500
      }
    }
}
```

SmartDepl requires that all the ABS classes, that can be involved in an automatically generated plan, have to be annotated. The annotations should be inserted immediately before the related class. These annotations, linked with ABS classes, contains:

- **Resources** consumed/required by an instance of the corresponding class. SmartDepl works with resources exploiting the ABS Cloud API [6]. Usually the main resources taken into account are the memory and the number of cores.

- Required dependencies and eventual numeric constraints on it (for instance, at least two services should be present in the initialization list of a load balancer). This information corresponds to **Required Ports** described in the formal model.

- A list of functionalities provided and eventual numeric constraints. These information correspond to **Provided Ports** described in the formal model.

An example can be taken from the code developed for the case-study model. In this particular case, the annotation associated with the Message Receiver microservice is used.

```
{
  "class" : "MessageReceiver",
  "scenarios" : [
    {
      "name" : "default",
      "provide" : -1,
      "cost" : {
            "Cores" : 2,
            "Memory" : 200
         },
         "sig" : [
            {
              "kind" : "require",
              "type" : "MessageParser_LoadBalancerInterface"
            }
         ],
         "methods" : []
      }
   ]
}
```

The previous JSON object specifies a single scenario, called *default*, where the microservice provides its interface to an unlimited amount of other components (”provide” value = -1). As resources, it requires 2 core and 200MB of RAM and as dependencies a reference to a component that provides the MessageParser_LoadBalancerInterface. This reference is passed through a parameter of the class constructor and formally it represents a strong required port. Additional scenarios can be inserted to specify different annotations for the same class, if it can be used in different modes.

The load balancer microservices have a slightly different annotation. An example is:

```
{
   "class" : "MessageParser_LoadBalancer",
   "scenarios" : [
      {
         "name" : "default",
         "provide" : -1,
         "cost" : {
            "Cores" : 2,
            "Memory" : 200
         },
         "sig" : [],
         "methods" : [
            {
               "add" : {
                  "name" : "connectInstance",
                  "param_type" : "MessageParserInterface"
               },
               "remove" : {
                  "name" : "disconnectInstance",
                  "return_type": "MessageParserInterface"
               }
            }
         ]
      }
   ]
}
```

They do not have strong required ports specified through the ”sig” field, but they have a weak required port represented through the ”methods” field. *connectInstance* and *disconnectInstance* are methods of the annotated class that both require a parameter of type MessageParserInterface. They implement the necessary operation to connect or disconnect an instance from the corresponding port during the component life-cycle.

The last type of possible annotations, as cited above, allows users to define and characterize the required deployment plans. These annotations can be specified everywhere in the ABS code. An example can be given, presenting the code that has been used in the case-study to generate the initial deployment plan.

```
{
   "id":"MainSmartDeployer",
   "specification":"
      DB = 1 and
      MessageReceiver = 1 and
      MessageParser = 1 and
      ...

      forall ?x in DC: (
        (?x.MessageParser_LoadBalancer +
         ?x.HeaderAnalyser_LoadBalancer +
         ...
         ?x.MessageReceiver +
         ?x.DB)
         > 0 impl (sum ?y in obj: ?x.?y) = 1
      )",
   "DC":[],
   "obj":[],
   "cloud_provider_DC_availability":{
      "c4_large":40,
      "c4_xlarge":40,
      "c4_2xlarge":40
   },
   "bind preferences":[
      "local",
      "sum ?x of type MessageParser in ’.*’ :
        forall ?y of type MessageParser_LoadBalancer in ’.*’:
         ?x used by ?y",
      "sum ?x of type HeaderAnalyser in ’.*’ :
       forall ?y of type HeaderAnalyser_LoadBalancer in ’.*’:
        ?x used by ?y",
      "..."
   ],
   "add_method_priorities":[],
   "remove_method_priorities":[]
}
```

The first field is the name of the deployment plan described. It will become the name of the corresponding generated ABS class. So, a class with that name and the expected behaviour (it offers the *deploy*/*undeploy* methods) can be used into the ABS code and it will become available after the SmartDepl execution. The field ”specification” contains the main deployment requirements. In this case the user requires an instance for all the main microservice (the corresponding load balancers are automatically added to satisfy the components dependencies). In addition to this request, the user requires that instances of load balancers, DB and MessageReceiver have to be deployed on a dedicated machine (if a machine hosts an instance of the specified types it can host only a single component, or in other words only its own). The "DC" field can be used to provide existing deployment components with free resources, if there are some, that can be used to deploy new objects without renting new machines, therefore saving money. The next field, "obj", instead can be used to specify already deployed components and their provided functionalities still available. They can be used if necessary without having to deploy them again. The "cloud_provider_DC_availability" allows to provide the available computation node types and the related multiplicity. The purpose of the last field used, "bind preferences", is very intuitive. In this case, it is required to prefer local bindings if possible and to maximize the number of connections between microservices and related load balancers. These last constraints guarantee that each instance is always connected with the corresponding load balancer. Some fields are not described here. More complex values can additionally be specified for the showed fields. If necessary, more information can be obtained in the SmartDepl documentation, publicly available [8].

Once all the necessary information has been provided and SmartDepl has been launched, it parses the ABS code, extracting all the presented annotations. In particular, it understands which virtual machines can be used, which components are coded, how many resources they require, what are their required and provided ports and the end the deployment plan characteristics. Using these inputs, it computes a plan to reach an optimal configuration that satisfy all the user requests. As cited above, the calculated plan can be triggered or undo using *deploy*/*undeploy* methods on an object of the generated ABS class. The code of the *deploy* method is intuitively, usually it instantiates new deployment components (that represent computation nodes), it creates new objects on them (that represent new instances) providing all the parameters required by the corresponding constructor (that represent the strong required ports satisfaction) and at the end it connects different deployed components using methods specified in the annotations

(that represent the weak required ports satisfaction). During these operations it updates data structures to store useful information to undo the executed operations when the corresponding *undeploy* method is called. The two methods can be used to deploy or undeploy the initial configuration or inside a monitor to scale-up or scale-down the system when necessary. The generated class interface offers also getter methods to obtain information or references of objects deployed or computation nodes used.
SmartDepl can also provide a graphical representation of the configuration computed, as shown in Figure 3.3.

3.2.3 Zephyurs2

As mentioned in the introduction of this section, Zephyurs2 is the engine used by SmartDepl to find the optimal configuration required. It is presented in [10], and here only a briefly introduction is provided. Zephyurs2 is a tool that solves the deployment optimization problem, or in other words, it searches a way to correctly deploy and configure an application minimizing the costs. These operations are usually done manually, requiring times and highly skilled specialists (DevOps team). Despite the usage of specifically prepared experts, the considered phases remain one of the major source of errors. The following document [54] reports that problems during deployment and configuration are the second cause of errors in Google data centres.
The Zephyurs2 inputs are:

- a description of the available virtual machines where components can be hosted, providing a name, the amount of each type of resource offered and the associated cost,

- a declarative specification of software components, containing information about required resources and dependencies, and

- a declarative representation, using constraints, of target configuration requirements and characteristics. For example the requested number of instances of a component or conflicts in co-location between instances of different components, and so on.

These information are provided using a specifically defined language based on JSON format. Clearly there are strong similarities between the following specifications and the previous described SmartDepl annotations. Indeed SmartDepl translates its annotations into Zephyrus2 accepted inputs, to use it to solve the provided problem. Example of virtual machine description is very easily understandable:

```
"c4_large": {
   "num": 40,
   "resources": { "Cores": 2, "Memory": 375 },
   "cost": 119
}
```

The first field specifies the number of virtual machines of the corresponding type available and then the other fields describe the resources offered and the cost required. An example of a component description is:

```
"MessageReceiver": {
  "resources": { "Cores": 2, "Memory": 200 },
  "requires": { "MessageParser_LoadBalancerInterface": 1 },
  "provides": [
      {
          "ports": [ "MessageReceiverInterface" ],
          "num": -1
      }
   ]
}
```

The fields are very intuitively: the first one specifies the resources consumed, the second one describes the strong-required ports with corresponding numerical constraint and in a similar way the last one the provided ports. The value -1 stays for ∞, in other words, it is used with ports that can be connected with an unbounded number of components.

The last type of input is specified using a specific language for deployment constraints. This language allows to provide simple constraints, but also more complex cloud- and application-specific constraints. To present some examples will be used the expressions presented in [10]:

```
HTTP_Load_Balancer > 0 and c3_large[1].WordPress = 1
```

This constraint requires at least an instance of HTTP_Load_Balancer component and exactly an instance of WordPress hosted in the second virtual machine of type *c3_large*. The considered virtual machine is the second one because the indexs start from 0.

```
forall ?x in locations: ( ?x.WordPress > 0 impl ?x.MySQL > 0)
```

Identifiers prefixed with a question mark are variables used to codify quantification and sum expressions. Quantification and sum building can range over components, locations, or over components/locations whose names match a given regular expression. The previous constraint specifies that in each location (virtual machine) where at least an instance of WordPress is deployed, also at least an instance of MySQL has to be deployed too. So this expression allows to capture a co-location requirement. In a similar way is possible to require that a component has to be installed alone, or in other words, that if a virtual machine hosts an instance of this component, the sum of other instances hosted has to be 0. So it hosts exactly one component. This idea is codify has:

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

Using more complex constraints is also possible to express preferences on bindings. The simplest possibility is to use only the keyword *local* that specifies the request of maximize the number of connections between components deployed in the same node. Arithmetic expressions can be used to capture more complex preferences. To better understand these advanced possibilities and to obtain more examples, the Zephyurs2 repository [11] can be used.

The language allows also to customizes the optimization criteria. The tool minimizes the expressions provided in the given priority order. *Cost* is the keyword used to represent the total cost of the application. Consequently the following expression, that is the default value used, provides the objective function to minimize first the total cost and then the total number of components:

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

Providing all the described information the tool returns the optimal deployment plan to reach a configuration that:

- is correct, considering components dependencies and resource usage,
- satisfies the user specifications given, and
- is the optimal one considering the objective function specified.

So, the main advantage obtained using the tool is that the application architects no longer have to take care of components distribution or component dependencies. In fact, they only describe the most important aspects of the application, exploiting the constraint expressiveness, and they leaves to the tool the responsibility of prepare the correct, and optimal, configuration that satisfies the characteristics provided. For example, if the application description specifies only the presence of an instance of a specific component, it is the tool that in an automatic way considers also all the components necessary to deploy the instance required, exploiting the information about dependencies provided in input.

Zephyurs2 is an evolution of Zephyurs. Respect to the previous version, Zephyurs2 can handle in a better way a larger number of components and virtual machines and it is faster. In addition to the performance, Zephyurs2 provides simplified input format using a better declarative language that allows the users to specify the deployment scenarios in a more direct and concise way.

The computation is based on modern SMT and CP technologies to better manage large and complex scenarios. In fact, Zephyurs2 translates the input provided into a *Constraint Optimization Problem* encoded in MiniZinc [55]. At this point the problem is solved successively minimizing the objective function components. Zephyrus2 supports also some extensions to give the possibility to find the better search method for the current problem. It is possible:

- to use MiniSearch [56], a meta-search language for MiniZinc, to give the possibility of use (heuristic) meta-searches solution approaches, such as large neighborhood search (LNS), lexicographic branch-and-bound, and And/Or search.
- the use of *satisfiability-modulo-theories (SMT) solvers.*

Zephyurs2 is developed in Python and it is open source. The Zephyurs2 implementation code can be found in [11] and it can be easily used and tried through a Docker container. In [10] can be found an experimental analysis of Zephyurs2 performances considering different settings and solving engines.

3.3 Deployment Plans

In this section will be introduced the deployment and scaling plans prepared for the realized model. The first step to prepare a Deployment Plan is the estimation of a maximum load, in term of inbound requests, supported by a single instance of each microservice. Considering that the developed model is used only to show the applicability of the discussed approach, these values have been calculated intuitively, sorting the different computations respect to a supposed workload. The list of adopted maximum input capacities (MICs) is showed in Table 3.1. It is possible to observe how light microservices, as Header Analyser that processes short and uniform data, have a bigger MIC than components with an heavier computations, as Image Recognizer.
For Sentiment Analyser and Message Analyser, the number of simultaneous manageable requests, 15K and 70K respectively, are transformed in number of messages. In fact, these two microservices are called more times for each message. In particular, an average of 2,5 blocks for message is considered for Sentiment Analyser (15/2,5=6K) and an average of 5 partial results received

for Message Analyser (70/5=14K). The two values depend in the first case on the length of the mail text and in the second case on the number of attachments contained in the received email.
The Table 3.1 contains also the cost of each computation used to implement the simulation of the system in ABS. In particular, for each microservice: a unit of time is divided by the corresponding MIC and the result obtained is multiplied by 1000 to obtain an integer value. For example, the Message Parser, Header Analyser and Link Analyser computations have a cost of $\frac{1}{40} \cdot 1000 = 25$, while Virus Scanner, NSFW Detector and Image Recognizer computations a cost of $\frac{1}{13} \cdot 1000 = 76$. The computation costs for Sentiment Analyser and Message Analyser have been calculated considering the number of simultaneous manageable requests (15K and 70K respectively) and not the number of simultaneous manageable messages. Because clearly the cost is payed at each call, so at each request. The costs are inserted in the code through the

```
[Cost: n] skip;
```

instruction that it is used to model the behaviour of each component without explicitly coded its. The cost annotations are necessary to exploit the *Deployment and Resource Modelling* features of ABS to implement a simulation. Costs are not introduced on components that cannot be scaled, as MessageReceiver, load balancers and DB. But these additional annotations can be easily introduced if they are considered useful.

Using the previous estimations, it has been prepared an initial deployment plan with only one instance for each microservice and three incremental scaling plans. Monitoring the inbound requests, or in other words, the number of emails received in the chose time windows, is possible to incrementally scale-up or scale-down the system using the more adapted plan. The calculated possibilities are showed in Table 3.2. It is important to observe that the scaling plans are incrementally. So, if the biggest forecast increments of inbound requests is registered all the three scaling plans have to be consequently applied and not only the last one. All the plans are computed

Microservice	Manageable Messages	Computation Cost
Message Receiver	∞	0
Message Parser	40K	25
Header Analyser	40K	25
Link Analyser	40K	25
Text Analyser	15K	66
Sentiment Analyser	6K (15K requests)	66
Virus Scanner	13K	76
Attachment Manager	30K	33
Image Analyser	30K	33
NSFW Detector	13K	76
Image Recognizer	13K	76
Message Analyser	14K (70K requests)	14

Table 3.1: Maximal number of simultaneous requests that each microservice can manage

considering 40 available computation nodes of each one of the three types, for a total of 120 available nodes.

The scaling factor of each microservice, or rather how many new instances have to be deployed for that microservice considering the computation load, clearly depends on the relative MIC. The number of instances added, it is more or less the amount of additional inbound requests (number of requests received minus the number of requests that the system can already correctly manage) divided by the corresponding MIC. For example, specifically analysing the Image Recognizer case: it has a MIC of 13K, so if the system received an increment of the inbound request equal to:

- +20K → $20/13 \cong 2$, this means that at least two more instances for this microservice have to be added to correctly manage the new computation load. Considering that the Image Recognizer computation is very heavy, it was decided to add directly 3 instances.

- +50K → $50/13 \cong 4$, this means that at least four more instances for this microservice have to be added to correctly manage the new computation load.
- +80K → $80/13 \cong 6$, this means that at least six more instances for this microservice have to be added to correctly manage the new computation load.

The values computed as described are used as references, but the scaling plans are then defined considering also other factors, changing and adapting these values where necessary.

The load balancers are not considered in the plans because they are not "main" microservices, so the user has not to focus on them. They are automatically singly deployed in the initial deployment plan because they are necessary to satisfy the microservice dependencies. A single instance is enough because their provided ports are unbounded so the single instance can satisfy all the corresponding required ports. All the microservice instances that are deployed (or un-deployed) during the scaling plans, are automatically connected (or disconnected) with the corresponding load balancers using the specifically prepared methods *connectInstance* and *disconnectInstance* (specified through the annotations) and provided by load balancer classes.

The algorithm used in practice to compute the previous described deployment and scaling plans is a little be different respect to that one presented in the decidability proof showed in Chapter 2. In particular, the *Phase 3* does not remove all the deployed instances and re-deploy the new configuration from scratch, but it simple adds the new instances and connects them with the corresponding load balancers (or it executes the inverse behaviour during scale-down operations). Considering the constraints definition, the maximization of bindings, showed as a possible extension at the end of the proof in Chapter 2, guarantees the described behaviour, where each new instance are connected with the corresponding load balancer.

The four presented plans are computed using SmartDepl that returns for each one an ABS class with a set of methods where the most important are:

Microservice (MIC)	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(6K)	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(14K)	1	+1	+2	+2

Table 3.2: Description of different deployment/scaling plans calculated

Deploy and *Undeploy.* These classes and the cited methods can be used in a system monitor to adapt the current configuration with the computation load, guaranteeing that each change adopted is optimal in term of computation node costs. SmartDepl is be able to generate the code to deploy the optimal configurations presented with a timeout of 30 minutes for each scenario. This time has been considered reasonable because the different deployment plans are computed in advance, predicting different system loads. To obtain on-the-fly deployment plans that would allow to immediately respond to unpredictable system loads, the tool should be sped. An ambitious goal could be the possibility to compute a plan in few minutes, around the average start-up time of a virtual machine in a public Cloud.

SmartDepl can also provide a graphical representation of the configuration computed. An example for the initial deployment plan previous described can be seen in Figure 3.3. It represents:

- Computation Nodes: with external boxes with the name on the top and smaller boxes inside for hosted microservice instances.

- Microservice Instances: with boxes, inside a node, with three types of information: the first field is the instance name, after that there are a sequence of green fields that represent provided ports and red fields that represent required ports.
- Bindings: with arrows from green fields (provided ports) to red fields (required ports).

3.4 Additional ABS Classes

In this section a more specific presentation of the code developed will be giveb. The ABS deployment components have been used to codify the computation nodes. Instead, each microservice has been modelled through an ABS class. Consequently each microservice instance is represented through an instantiation of the corresponding ABS class, or in other words, through an ABS object. The code developed consequently contains an ABS class for each microservice described in the previous sections and another for the corresponding load balancer. The connections between instances are managed through object references. If an instance has a reference to another, it can call/communicate with its. This situation correctly model the presence of a binding between them. The bindings required to create a new microservice instance, that in our model are all the connections to strong required interfaces, are inserted as parameters of the class constructor. In this way, they have to be available to be able to create a new instance of the class, or rather, to simulate the deployment of a new instance of the analysed microservice. These parameters are specified in the SmartDepl annotations as mandatory information to deploy a new instance of the microservice type captured by the corresponding class. SmartDepl provides them in the plans, deploying new instances if necessary or using previous deployed instances of the necessary types if available. Instead, to model weak required interfaces, it is

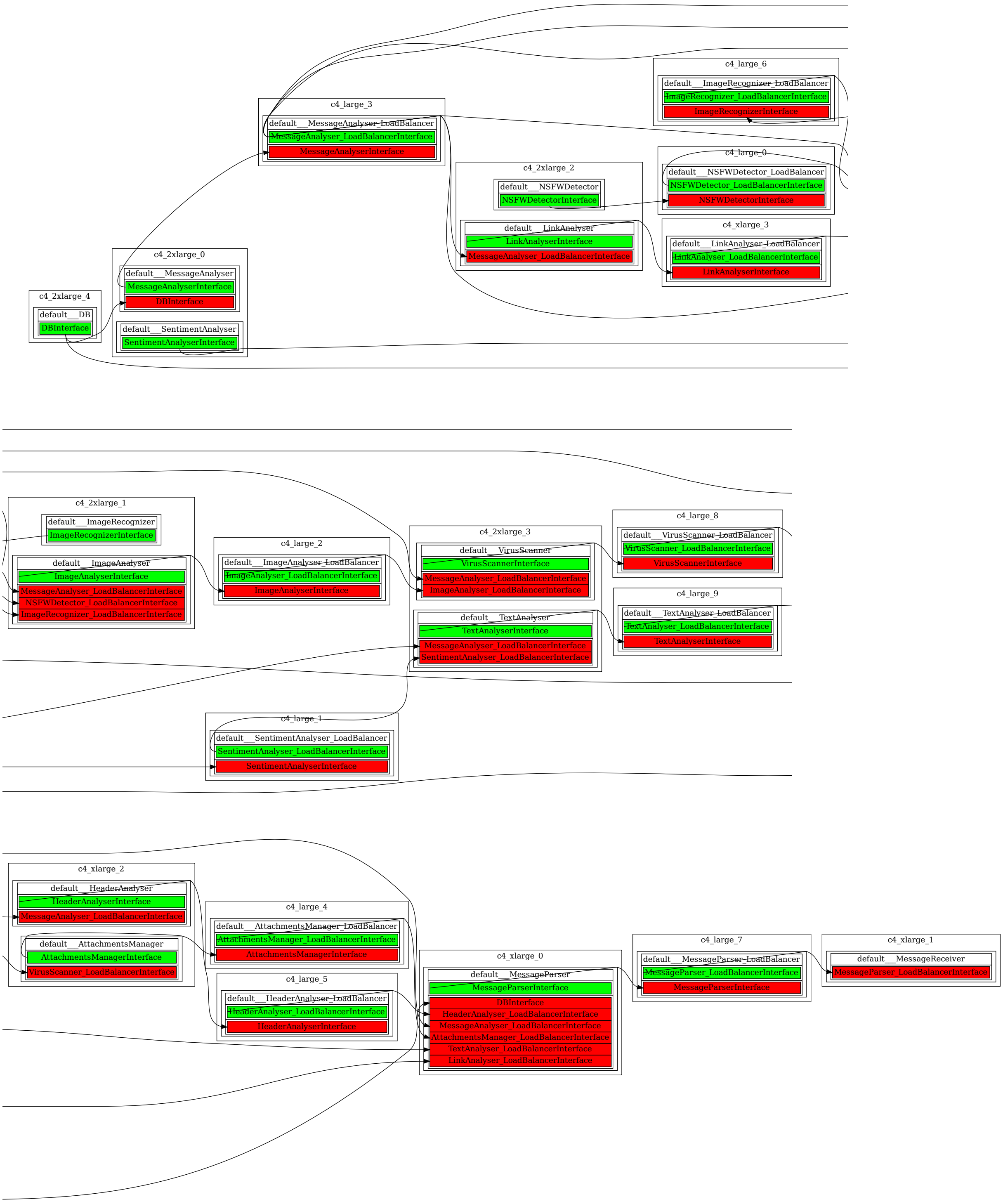
c4_large_6
default__ImageRecognizer_LoadBalancer
ImageRecognizer_LoadBalancerInterface
ImageRecognizerInterface
c4_large_3
default__MessageAnalyser_LoadBalancer
MessageAnalyser_LoadBalancerInterface
MessageAnalyserInterface
c4_2xlarge_2
default__NSFWDetector
NSFWDetectorInterface
default__LinkAnalyser
LinkAnalyserInterface
MessageAnalyser_LoadBalancerInterface
c4_large_0
default__NSFWDetector_LoadBalancer
NSFWDetector_LoadBalancerInterface
NSFWDetectorInterface
c4_xlarge_3
default__LinkAnalyser_LoadBalancer
LinkAnalyser_LoadBalancerInterface
LinkAnalyserInterface
c4_2xlarge_0
default__MessageAnalyser
MessageAnalyserInterface
DBInterface
default__SentimentAnalyser
SentimentAnalyserInterface
c4_2xlarge_4
default__DB
DBInterface
c4_2xlarge_1
default__ImageRecognizer
ImageRecognizerInterface
default__ImageAnalyser
ImageAnalyserInterface
MessageAnalyser_LoadBalancerInterface
NSFWDetector_LoadBalancerInterface
ImageRecognizer_LoadBalancerInterface
c4_large_2
default__ImageAnalyser_LoadBalancer
ImageAnalyser_LoadBalancerInterface
ImageAnalyserInterface
c4_2xlarge_3
default__VirusScanner
VirusScannerInterface
MessageAnalyser_LoadBalancerInterface
ImageAnalyser_LoadBalancerInterface
default__TextAnalyser
TextAnalyserInterface
MessageAnalyser_LoadBalancerInterface
SentimentAnalyser_LoadBalancerInterface
c4_large_8
default__VirusScanner_LoadBalancer
VirusScanner_LoadBalancerInterface
VirusScannerInterface
c4_large_9
default__TextAnalyser_LoadBalancer
TextAnalyser_LoadBalancerInterface
TextAnalyserInterface
c4_large_1
default__SentimentAnalyser_LoadBalancer
SentimentAnalyser_LoadBalancerInterface
SentimentAnalyserInterface
c4_xlarge_2
default__HeaderAnalyser
HeaderAnalyserInterface
MessageAnalyser_LoadBalancerInterface
default__AttachmentsManager
AttachmentsManagerInterface
VirusScanner_LoadBalancerInterface
c4_large_4
default__AttachmentsManager_LoadBalancer
AttachmentsManager_LoadBalancerInterface
AttachmentsManagerInterface
c4_large_5
default__HeaderAnalyser_LoadBalancer
HeaderAnalyser_LoadBalancerInterface
HeaderAnalyserInterface
c4_xlarge_0
default__MessageParser
MessageParserInterface
DBInterface
HeaderAnalyser_LoadBalancerInterface
MessageAnalyser_LoadBalancerInterface
AttachmentsManager_LoadBalancerInterface
TextAnalyser_LoadBalancerInterface
LinkAnalyser_LoadBalancerInterface
c4_large_7
default__MessageParser_LoadBalancer
MessageParser_LoadBalancerInterface
MessageParserInterface
c4_xlarge_1
default__MessageReceiver
MessageParser_LoadBalancerInterface

Figure 3.3: Graphical representation of the initial configuration automatic generated by SmartDepl

necessary to specify methods to add or remove references to microservices when the object is already instantiated. These methods are called with an object reference to add a binding to the corresponding weak required port or to disconnect its. The SmartDepl annotations contains a description of prototypes of these methods and in the analysed case-study they are used to model load balancers. The names chose for them are: *connectInstance* and *disconnectInstance.*

In addition to the already introduced ABS classes used to model the presented case-study, others have been developed to implement a simply simulation. They can be schematically presented:

- EntryPoint: simply forwards messages to MessageReceiver. It is introduced to obtain an external callable entrypoint. Indeed the Erlang backend, chosen to execute the ABS code, supports external calls to a running model through HTTP requests. This feature can be found in the ABS documentation in the Model API section [7]. The calls through HTTP requests can be executed only on methods annotated as *HTTPCallable* provided by an instance annotated with *[HTTPName: ”name”].* To correctly annotate the corresponding creation instruction, it can not be inserted in a deployment plan. So, this component is manually created in the *initializeSystem* method of the SetUpSystem class that will be presented later. Thank to this new component the following Python code could be used to interact with the system.

  ```
  import requests
  import json

  url = ”http://localhost:8080/v2/call/entryPoint/newMessage”
  payload = {’mailData’: ’Message1’}
  r = requests.post(url, data=json.dumps(payload))
  ```

- WrapperScale: is used to level out the different scale classes interfaces and to correctly manage the fact that the scaling plans are incrementally. Indeed when a scaling plan is requested through this class, also all the smaller plans are applied. For example, if the monitor requires the application of the biggest scaling plan (Scale3), the *deploy* methods are called on objects representing the Scale1, the Scale2 and the Scale3 plans at the same time. The same behaviour is clearly offered also for the *undeploy* operations.

- Monitor: is used to periodically monitor the system exploiting methods offered by DB to obtain information on: number of received messages in the current monitoring window and average analysis time for each message. It uses the first value to calculate the difference respect to the number of messages actually supported and consequently decides which scaling plan should be applied. The possible scaling plans have to be inserted through an apposite method called *insertScalingPolicy* providing the scaling plan name, the WrapperScale object and the value used to decide when that plan should be used. At the moment the average analysis time obtained by DB is used only to print a useful value to observe the system adaptability to work load changes.

- MailGenerator: is used to generates mail with different speeds to periodically change the work load received by the system.

- SetUpSystem: contains only a method to initialize the system. It simply instantiates:

  - the CloudProvider,
  - the initial deployment plan and applies it to create the system,
  - the monitor,
  - the three WrapperScale objects for the corresponding three scaling plans and inserts them in the monitor, and
  - the EntryPoint object, specifying its HTTPName through the specific annotation.

At this point, the main function simply instantiates the SetUpSystem object and calls on it the *initializeSystem* method obtaining the EntryPoint object. This one is used to instantiate the MailGenerator and the simulation is consequently launched.

To improve the possibilities given by simulation results, the costs of each computation reported in Table 3.1 are increased by 60% to obtain bigger analysis average times. Bigger values are useful to execute studies and different tests on them. Observing the simulation prints, it is possible to note that the system, using the scaling plans, tries to maintain the average analysis time (printed as AAT) constant. In the reported results, around 10 time units. Clearly, when there is a big change in the amount of inbound requests the system requires the necessary scaling policy but some times is necessary to apply its allowing the new configuration to be completely operational again. During that time, the average analysis time grows before returns to the fixed value.

```
Initial Deployment Configuration set up
Message Arrived = 10 - AAT = 15
Message Arrived = 10 - AAT = 14
Message Arrived = 30 - AAT = 55   Scale1-up
Message Arrived = 30 - AAT = 67
Message Arrived = 30 - AAT = 11
Message Arrived = 30 - AAT = 11
Message Arrived = 30 - AAT = 10
Message Arrived = 80 - AAT = 66   Scale2-up
Message Arrived = 80 - AAT = 89
Message Arrived = 80 - AAT = 65
Message Arrived = 80 - AAT = 17
Message Arrived = 80 - AAT = 9
Message Arrived = 80 - AAT = 9
Message Arrived = 80 - AAT = 10
Message Arrived = 160 - AAT = 23   Scale3-up
```

```
17 Message Arrived = 160 − AAT = 47
18 Message Arrived = 160 − AAT = 41
19 Message Arrived = 160 − AAT = 41
20 Message Arrived = 160 − AAT = 14
21 Message Arrived = 160 − AAT = 9
22 Message Arrived = 160 − AAT = 9
```

The previously described behaviour can be observed in lines [4-6], [9-13] and [16-20]. With the increase of messages received, the average analysis time increases too, but when the required scaling policy became operative the monitored parameter returns to the fixed value (≅ 10).

The ABS code realized, the annotations and the generated classes are publicly available at [18]. The following simple code of HeaderAnalyser class is used to provide an example.

```
data HeadersAnalysis = HeadersAnalysis(
   String haResults,
   String haMessageId
);
interface HeaderAnalyserInterface {
   Unit analyzeHeaders (String headers, String messageId);
}
/* {
        "class" : "HeaderAnalyser",
        "scenarios" : [
           {
              "name" : "default",
              "provide" : −1,
              "cost" : {
                 "Cores" : 2,
                 "Memory" : 200
              },
              "sig" : [
                 {
```

```
            "kind" : "require",
            "type" : "MessageAnalyser_LoadBalancerInterface"
          }
        ],
        "methods" : []
      }
    ]
  } */
[SmartDeployCost : "..."]
class HeaderAnalyser(MessageAnalyser_LoadBalancerInterface messageAnalyserLoadBalancer) implements HeaderAnaly
  Unit analyzeHeaders (String headers, String messageId){
    //analyze headers to extract useful high-level information
    [Cost: 25] skip;
    HeadersAnalysis res = HeadersAnalysis("Results of HeaderAnalysis (" + headers + ") by HeaderAnalyser in " + t
    //send analysis results (HeadersAnalysis object) to MessageAnalyser
    messageAnalyserLoadBalancer!insertHeadersAnalysisResults(res);
  }
}
```

The first lines contain the declaration of a user-defined algebraic data type used to store headers analysis results. Then, the interface of the analysed component is specified. It contains only an operation. After that, the SmartDepl annotation is previously presented in JSON format and later correctly inserted with the corresponding instruction. The last part is the real code of the analysed microservices. It contains the implementation of the single method declared in the corresponding interface. The method simply simulates the microservice behaviour through a *skip* instruction and then creates a dummy result structure that is sent to the MessageAnalyser LoadBalancer.

Conclusion

This research project addressed the Optimal Deployment Problem for Microservice Architectures. It started from the Aeolus model [1] that has been presented in Chapter 1, and from contributions of Zephyrus [10] and ConfSolve [24]. Inspired by the approaches followed and container-technologies such as Docker [36] and Kubernetes [39], a new model specifically thought for microservices has been proposed. The project began with the formal definition of the new model and a formal proof to demonstrate the decidability and complexity properties of that model. These are described in Chapter 2. Then, the theory part is followed by a practice section, which analyses a case-study showing the applicability of the approach and ideas proposed. The example model realized, presented in Chapter 3, shows that the generation of a deployment plan for an architecture of microservices is fully automatable; in particular a tool can compute an optimal configuration and prepare the deployment actions necessary to reach it.

To support specification of deployment plans, different specification languages [46, 47], reconfiguration protocols [48, 49] and system management tools [50, 51, 52, 53] already exist, but they do not consider the computation nodes and the distribution of the system components over these. The proposal presented in this dissertation has been specifically realized to solve the cited problem providing all the necessary tools and considering the deployment optimality with respect to other possibilities available. The approach proposed tries also to go beyond single-component horizontal scaling policies that are the classical behaviour of very widespread autoscaling systems,

proposing a new alternative that works at an higher-level and exploits architecture information.
Some possible evolutions have been identified. As already mentioned in Chapter 3, the main future work is to investigate approaches, like local search, to speed-up the resolution of the optimization problem, which is the heaviest part of the computation. This advancement is necessary to allow users to dynamically prepare and compute deployment plans, giving them the possibility to use custom response to unpredictable loads instead of pre-prepared plans. The realization of a graphical tool that allows to easily specify all the necessary information and obtains a description of optimal deployment plans searched following a driven path, could open the use of this approach outside the academic environment. Starting from this tool, a direct interfacing with cloud platform or other real deployment languages could be imagined. This would give the chance to automatize the plan application allowing the user to comfortably specify in a declarative way all the required information (through a graphic UI) and to obtain in an automatic way the system deployed with the optimality guaranteed. In spite of these possible evolutions, the current version can already be used and it could prove to be useful in multiple situations. For example to deploy a Cloud application. In this case, the optimality guaranteed by the tool allows users to immediately save money paid to rent virtual machines from cloud provider.

This experience has been amazing for me. I had the chance to work for about a year on a real research project that has provided several reasons for gratification. It has been recognized as interesting from the scientific community, being accepted at two conferences after passing different reviews. I have personally grown clearly on the technical/scientific side but also on the human side, collaborating with fantastic and very present professors and overcoming my fears when I presented this work at the Microservices 2019 conference in Dortmund in front of a large audience, discussing and answering questions not only on my part of the project but on the entire work. In addition, during experiences linked with this project I met many professors and students – for example during my visit period at SIRIUS research centre in Oslo in February 2018 where I began to study ABS and Kubernetes – with whom I shared ideas, solutions and possible evolutions of this work but also other topics.

Bibliography

[1] Aeolus: A component model for the cloud. Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro

[2] Thinking Serverless! How New Approaches Address Modern Data Processing Needs. Ken Fromm (https://read.acloud.guru/thinking-serverless-how-new-approaches-address-modern-data-processing-needs-part-1-af6a158a3af1, accessed on May, 2019)

[3] https://abs-models.org/ (accessed on May, 2019)

[4] https://abs-models.org/manual/ (accessed on May, 2019)

[5] ABS: A Core Language for Abstract Behavioral Specification. Einar Broch Johnsen, Reiner Hähnle, Jan Schäfer, Rudolf Schlatte, Martin Steffen

[6] https://abs-models.org/manual/#sec:deployment (accessed on May, 2019)

[7] https://abs-models.org/manual/#-the-model-api (accessed on May, 2019)

[8] SmartDepl. Jacopo Mauro (https://abs-models.org/tutorial_pdfs/SmartDepl.pdf, accessed on May, 2019)

[9] Declarative Elasticity in ABS. Stijn Gouw, Jacopo Mauro, Behrooz Nobakht, Gianluigi Zavattaro

[10] Zephyrus2: On the Fly Deployment Optimization Using SMT and CP Technologies. Erika Ábrahám, Florian Corzilius, Einar Broch Johnsen, Gereon Kremer, and Jacopo Mauro

[11] Zephyrus2. Jacopo Mauro
(https://bitbucket.org/jacopomauro/zephyrus2/src, accessed on May, 2019)

[12] Beyond auto-scaling: application-aware optimal elasticity. Jacopo Mauro, Iacopo Talevi, and Gianluigi Zavattaro

[13] https://www.conf-micro.services/2019/ (accessed on May, 2019)

[14] Optimal and Automated Deployment for Microservices. Jacopo Mauro, Saverio Giallorenzo, Mario Bravetti, Iacopo Talevi, and Gianluigi Zavattaro

[15] Technical Report: Optimal and Automated Deployment for Microservices. Jacopo Mauro, Saverio Giallorenzo, Mario Bravetti, Iacopo Talevi, and Gianluigi Zavattaro
(https://arxiv.org/abs/1901.09782, accessed on 2019)

[16] https://conf.researchr.org/track/etaps-2019/fase-2019-papers (accessed on May, 2019)

[17] A Formal Approach to Microservice Architecture Deployment. Jacopo Mauro, Saverio Giallorenzo, Mario Bravetti, Iacopo Talevi, and Gianluigi Zavattaro (accepted for publication in ”Microservices Science and Engineering” book)

[18] Code repository for the email processing example. Jacopo Mauro, Saverio Giallorenzo, Mario Bravetti, Iacopo Talevi, and Gianluigi Zavattaro

(https://github.com/IacopoTalevi/ SmartDeploy-ABS-ExampleCode, accessed on June, 2019)

[19] https://aws.amazon.com (accessed on May, 2019)

[20] https://cloud.google.com (accessed on May, 2019)

[21] https://azure.microsoft.com (accessed on May, 2019)

[22] The FRACTAL component model and its support in Java. Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, Jean-Bernard Stefani

[23] Reconfigurable SCA Applications with the FraSCAti Platform. Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, Jean-Bernard Stefani

[24] A declarative approach to automated configuration. John Hewson, Paul Anderson, Andrew Gordon

[25] https://www.chef.io/products/chef-infra/ (accessed on May, 2019)

[26] https://www.cloudfoundry.org/ (accessed on May, 2019)

[27] https://jaas.ai/ (accessed on May, 2019)

[28] Computation: Finite and Infinite Machines. Marvin Lee Minsky

[29] Well-structured transition systems everywhere!. Alain Finkel, Philippe Schnoebelen

[30] Revisiting Ackermann-Hardness for Lossy Counter Machines and Reset Petri Nets. Philippe Schnoebelen

[31] On the Expressiveness of Synchronization in Component Deployment. Jacopo Mauro, Gianluigi Zavattaro

[32] https://blog.newrelic.com/technology/microservices-what-they-are-why-to-use-them/ (accessed on May, 2019)

[33] Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Jez Humble, David Farley

[34] A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. Tania Lorido-Botran, Jose Miguel-Alonso, Jose A. Lozano

[35] Amazon: AWS auto scaling.
(https://aws.amazon.com/autoscaling/, accessed on May, 2019)

[36] Docker: lightweight Linux containers for consistent development and deployment. Dirk Merkel

[37] https://www.docker.com/ (accessed on May, 2019)

[38] Docker compose documentation.
https://docs.docker.com/compose/ (accessed on May, 2019)

[39] Kubernetes: Up and Running Dive into the Future of Infrastructure. Brendan Burns, Kelsey Hightower, Joe Beda

[40] https://kubernetes.io/ (accessed on May, 2019)

[41] The CP solver. Chuffed Team
(https://github.com/geoffchu/chuffed, accessed on May, 2019)

[42] GECODE: An open, free, efficient constraint solving toolkit.
(http://www.gecode.org, accessed on May, 2019)

[43] Google: Optimization tools.
(https://developers.google.com/optimization/, accessed on May, 2019)

[44] https://aws.amazon.com/ec2/pricing/on-demand/ (accessed on May, 2019)

[45] Automatic application deployment in the cloud: From practice to theory and back. Roberto Di Cosmo, Michael Lienhardt, Jacopo Mauro, Stefano Zacchiroli, Gianluigi Zavattaro, Jakub Zwolakowski

[46] A systematic review of cloud modeling languages. Alexander Bergmayr, Uwe Breitenbücher, Nicolas Ferry, Alessandro Rossini, Arnor Solberg, Manuel Wimmer, Gerti Kappel, Frank Leymann

[47] Madeus: A formal deployment model. Maverick Chardet, Hélène Coullon, Dimitri Pertin, Christian Pérez

[48] Robust reconfigurations of component assemblies. Fabienne Boyer, Olivier Gruber, Damien Pous

[49] Robust and reliable reconfiguration of cloud applications. Francisco Durán, Gwen Salaün

[50] Ansible. Red Hat
(https://www.ansible.com/, accessed on May, 2019)

[51] Puppet: Next-generation configuration management. Luke Kanies

[52] Opscode. Chef
(https://www.chef.io/chef/, accessed on May, 2019)

[53] Marionette collective. Puppet Labs
(http://docs.puppetlabs.com/mcollective/, accessed on May, 2019)

[54] The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. Luiz André Barroso, Jimmy Clidaras, Urs Hölzle

[55] MiniZinc: Towards a Standard CP Modelling Language. Nicholas NethercotePeter J. StuckeyRalph BecketSebastian BrandGregory J. DuckGuido Tack

[56] MiniSearch: A Solver-Independent Meta-Search Language for MiniZinc. Andrea Rendl, Tias Guns, Peter J. Stuckey, Guido Tack

Acknowledgement

I would like to express my gratitude and my affection to Professor Gianluigi Zavattaro, not only for the great opportunity to collaborate in this project, but also to have been a shepherd for me during my University years. Thanks to him I had the possibilities to pass a month period as guest at SIRIUS research centre in Oslo and to follow my first research conference in Dortmund. I am also indebted to Professor Jacopo Mauro. He helps me a lot during this project answering hundreds questions asked through mail or Whatapp/Skype messages. In addition to this, he also received and guided me during my period in Oslo and he gave me a lot of advices before my first conference presentation in Dortmund when I was very nervous. Their helps and their continuous support allows me to arrive here.

During this research project I also met Professor Mario Bravetti and post-doctoral researcher Saverio Giallorenzo. I would like to thank them and in particular Saverio, who more collaborated with me during the case-study model design and realization, solving multiple problems linked with my code.

My heart-felt thanks to my sister for her helps during the writing of this dissertation, in spite of her several appointments around the world, and to my family, in particular my parent to allows me to focus only on University during this five years without worries.

I want to mention also my girlfriend Federica, who always believes in me and encourages me to give my best, putting up with me in these years. I cannot cite my roommate: Federico, Lorenzo and Luca that put up with all my flaws. I am very happy to share my flat with them for at least another year. Finally, my old friends in Falconara because our relationships do not change despite of the distance and my new friends in Bologna, who welcome me with open arms, in spite of I am the last arrived.