Self-Reconfiguring Microservices*

Maurizio Gabbrielli[1,2], Saverio Giallorenzo[1], Claudio Guidi[3], Jacopo Mauro[4], Fabrizio Montesi[5]

[1] University of Bologna, Department of Computer Science and Engineering
[2] INRIA, Focus Team
[3] italianaSoftware, Italy
[4] University of Oslo, Department of Informatics
[5] University of Southern Denmark, Department of Mathematics and Computer Science

Abstract. Microservices is an emerging paradigm for the development of distributed systems that, originating from Service-Oriented Architecture, focuses on the small dimension, the loose coupling, and the dynamic topology of services. Microservices are particularly appropriate for the development of distributed systems in the Cloud. However, their dynamic nature calls for suitable techniques for their automatic deployment. In this paper we address this problem and we propose JRO (Jolie Redeployment Optimiser), a tool for the automatic and optimised deployment of microservices written in the Jolie language. The tool uses Zephyrus, a state of the art tool that automatically generates a fully detailed Service-Oriented Architecture configuration starting from a partial and abstract description of the target application.

Keywords: Microservices, Service-Oriented Architecture, Automatic Deployment, Optimal Component Allocation

1 Introduction

Microservices [16] is an emerging paradigm for the development of distributed systems that evolved from Service-Oriented Architecture [18] (SOA). The key aspect of microservices is that the idea of using services as components is pervasive.

In typical SOAs, services are used as an overlay meant to integrate and coordinate autonomous information systems. This coordination is obtained via communications, which operate using standard protocols. Such information systems can be built following different methodologies; in practice, many of them are legacy systems. Microservices explore a different direction, i.e., that of using services as the inner components of an information system. This allows to apply

* Supported by the EU project FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems* and *CRC (Choreographies for Reliable and efficient Communication software)*, grant no. DFF–4005-00304 from the Danish Council for Independent Research.

to microservices the same principles that apply to component-based software engineering. For example, since microservices should be small (or, better, "micro") it should be natural to follow principles towards cohesion, such as the Single Responsibility Principle[6].

Moreover, in this paradigm even the components of a single software application are all autonomous services that can interact only through message passing. This has the important benefit of obtaining a loosely-coupled implementation of the internals of an application, thus facilitating modularity and scalability. Due to the fact that microservices are already loosely-coupled, operate via message passing, and offer APIs to be invoked by external software it is easier to coordinate information systems based on microservices.

To understand how microservices support scalability, suppose that a service in a system is under heavy load. Since all the other components can interact with this service only through its message interface, we can replace it with a load balancer that offers the same API and forwards requests to a new subsystem running a set of replicated instances of the original service. From the loose coupling property of microservices we obtain that the rest of the system remains unchanged, independently from its implementation details. This feature makes the topology of a microservices architecture (i.e., the number of its components and their interactions) very dynamic.

Due to their properties, one of the main application contexts of microservices is the deployment of distributed systems in the Cloud [36]. Indeed, in the Cloud it is easy to scale the infrastructure of a system by adding or removing instances of virtual machines. However, allocating and deploying services on that machines while the system is running is a complex task. Usually the deployment of services is done either manually or it is handled programmatically with pre-configured deployment schemas that tools like Puppet [43] and Chef [40] automate. In either cases, the developers and DevOps[7] must carefully define where — in which virtual machine — services must be deployed and specify their connections. The planning of the deployment of a system must balance between the cost of its resources and its performances. Even in systems composed of few types of services, devising such a *deployment plan* quickly becomes a cumbersome and complex task due to dependencies between services and availability of different kinds of virtual machines, with different range of resources and costs. When looking for an *optimal plan* the task becomes extremely difficult, also from a theoretical perspective, since very easily one encounter NP-hard [26] and even undecidable problems [8].

In this paper, we address the problem of automatic optimal deployment planning of microservices. We assume the use of reconfigurable microservices, thus abstracting from the preservation, partition, and consistency of their state and data between successive re-deployments. We present Jolie Redeployment Opti-

[6] This is a well know example from the object oriented world, stating that there should never be more than one reason for a class to change.

[7] DevOps are professionals that collaborate in the development of programs by reporting their experiences with tests and deployments scenarios to developers [11].

miser (JRO), a tool for the automatic and optimised deployment of microservices written in the Jolie language [21, 30–32]. Jolie is an open-source programming language for developing distributed applications based on microservices which combines computation and composition primitives in an intuitive and concise syntax. In Jolie each component is a (micro)service that can communicate with other components by sending and receiving messages over a network. The behaviour and deployment of a Jolie service are orthogonal: they can be independently defined and recombined as long as they have compatible typing. In order to support concurrency, a service can run multiple instances of its behaviour, called processes. Processes can direct messages to each other by using arbitrary sets of data, a mechanism commonly called message correlation [38] and borrowed from Service-Oriented Architectures. The semantics of processes and correlation in Jolie is formally defined [29] and used in studies aiming at providing formal properties on service systems, such as those based on choreography languages [4,33]. Jolie also includes useful features for the programming of dynamic service systems such as embedding that allows the supervised execution of sub-services inside of other services [28]. Embedding can be used at runtime to enable service mobility and the runtime adaptation of parts of a running process [23].

The Jolie Redeployment Optimiser tool is based on the following three main components:

Zephyrus [7] A tool that automatically generates, starting from a partial and abstract description of the target application, a fully detailed architecture, indicating which and how many components are needed to realize such application, how to distribute them on virtual machines, and how to bind them together. Zephyrus is also capable of producing *optimal* architectures, minimizing the amount of needed virtual machines while still guaranteeing that each service has its needed share of computing resources (CPU power, memory, bandwidth, etc.) on the machine where it gets deployed.

Jolie Enterprise (JE) A distributed framework for deploying and managing microservices written in the Jolie language. Jolie Enterprise exposes Application Program Interfaces (APIs) *i*) to access all the data related to the platforms and services running in the managed system, *ii*) to deploy, start, stop, and remove services, and *iii*) to monitor their performances and resource consumption.

Jolie Reconfiguration Coordinator (JRC) A tool that, given a desired configuration and a context for the deployment (provided by Jolie Enterprise) interacts with Zephyrus to produce the optimised deployment planning.

We depict in Figure 1 how JRC, JE, and Zephyrus interact in JRO, starting from a desired configuration and its actual deployment. The sequence of interactions in Figure 1 can be described as follows.

1. The User defines the requirements of the deployment, e.g., how many instances of a service must be deployed or that some type of services cannot run in the same machine with others.

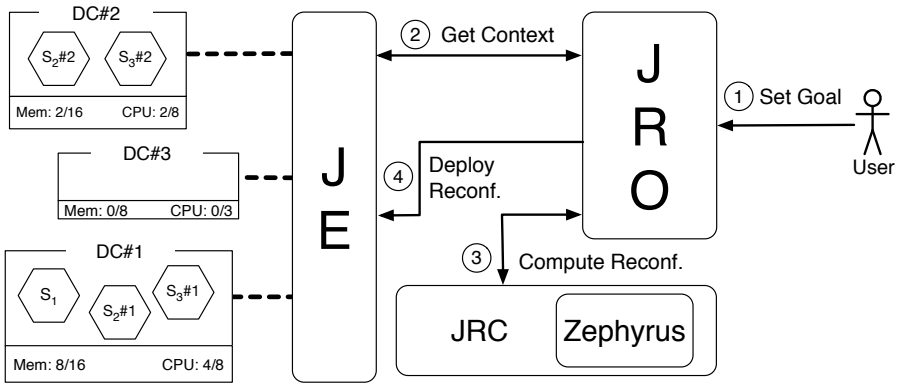

Fig. 1. JRO Workflow

2. JRO retrieves from JE the context of the deployment, i.e., the available virtual machines in the system (in the figure $DC\#1$, $DC\#2$, and $DC\#3$).
3. JRO uses JRC which uses Zephyrus to find the optimal solution.
4. If the User agrees with the solution, JRO proceeds with the orchestration of the deployment, instructing JE on how services should be deployed, linked or removed.

To the best of our knowledge JRO is the first tool allowing to optimally deploy a microservice based application.

Structure of the paper. Section 2 presents a comprehensive, real-world use-case to illustrate how JRO works from the user perspective. In Section 3 we describe the details of JRO and its features. Section 4 contains a discussion on related work and our closing remarks.

2 Example

In this section, we show how JRO can be used to deploy a realistic SOA using as a running example a blog microservices architecture [27]. As depicted in Figure 2, the blog comprises 5 types of microservices for post publication and commenting:

- `Auth` enables the users of the blog to authenticate themselves;
- `Posts` allows an *author* to edit a post. `Posts` needs an instance of `Auth` to authenticate authors;
- `Comments Balancer` dispatches the submission of comments from the *readers* to an instance of the `Comments` service;
- `Comments` receives the submission of a comment and publishes it. `Comments` needs an instance of `Comments Balancer` to receive incoming submissions and an instance of `Auth` to authenticate the reader who sent the comment;

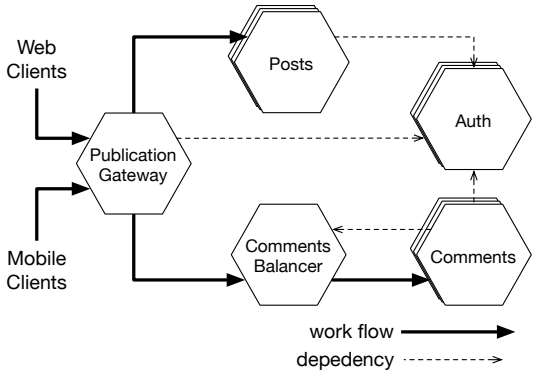

Fig. 2. Blog microservices architecture.

– `Publication Gateway` is the service accessed by clients to read the blog. `Publication Gateway` needs and instance of `Auth` to let *readers* access the contents of the blog.

All these services come with some information related to the resources that they require to be installed. In particular, every service specifies how much RAM and processing power it needs, to how many services it can provide its functionalities (Provision) and the number and the type of services it requires to work (Dependencies)[8]. In the table below we summarize this information for the services of the blog.

Service	Mem	CPU	Dependencies	Provision
`Auth`	50	2	-	5
`Posts`	20	1	Auth: 1	1
`Comments Balancer`	50	4	-	∞
`Comments`	30	1	Auth: 1, Comments Balancer: 1	1
`Gateway`	50	4	Auth: 1	∞

Observe that the profiling of `Comments Balancer` and `Gateway` marks a theoretical infinite provision. This is because these services do no intensive computation and they just dispatch requests towards other services. The `Auth` instead can be used by 5 other services instances, whether they may be `Post`, `Comment`, or `Gateway` services.

The usual way of setting up an instance of the blog to satisfy some expected traffic load requires to reserve some virtual machines and deploy a certain number of `Post`, `Comment`, `Comment Balancer`, and `Gateway` services, which in turn need the deployment of several `Auth` services. Besides the deployment, it would be also

[8] We assume that this information, usually obtained through some profiling of the services, has to be initially entered by the service developer.

necessary to connect all the deployed services — e.g., all `Post` services to their correspondent `Auth` services — in such a way that they can sustain the expected load and do not generate bottlenecks.

With JRO all these concerns are handled automatically and it is guaranteed that the obtained deployment respects the initial desiderata.

For example, let us consider that a DevOps wants to deploy two `Posts` services, two `Comments` services, and a `Gateway`. In JRO she does that by specifying the following string.

```
Post = 2 and Comments = 2 and Gateway = 1
```

These services are usually deployed on a cloud or some (private) cluster of machines. In the context of this work, we use the term of Deployment Containers (DC) to capture the notion of the basic unit where services can be deployed, whether they may be virtual machines, physical machines, or containers a la Docker [13]. A DC is characterised by a cost and some resources that it can provide. For this running example, let us consider the two DCs reported below and characterised by their *Cost*, expressed in dollar/month, *Memory* expressed in MB, and processing power, expressed in processor units (CPU).

DC	**Cost**	**Memory**	**CPU**
Small	4	60	2
Big	6	100	4

When the DevOps enters her desiderata, JRO automatically computes the optimal (i.e., the least expensive) configuration that satisfies her request. In our case, the computed configuration is the one reported in Figure 3, where a `Gateway` service and a `Comments Balancer` service are deployed in two separated Big DC, two `Comments` services are in a Small DC and the remaining services (one `Auth` and two `Post`) are on another Big DC.

The DevOps obtained a correct configuration but she realises that it is not right for fault tolerance and load balancing reasons. Indeed, deploying on the same DC respectively two `Post` services and two `Comments` services can lead to outages in case of high load or crash of one of the DCs. Hence, the DevOps wants to specify that services of the same kind should be deployed on different machines. With JRO it is also possible to express constraints on the co-location and distribution of services. Let us suppose that DevOps requires that every DC contains at most one `Post`, one `Comments`, and one `Auth` service and that the `Auth` service cannot be co-located with a `Post` or `Comments` service. In this case, the configuration computed by JRO is the one depicted in Fig. 4. The DevOps finally accepts the solution and deploys the obtained configuration.

Let us now make the case that, after some usage, the DevOps notices that many users comment the same post, which overloads the `Comments` services and slows down the responsiveness of the blog. To cope with the high load on comments, the DevOps wants to re-deploy the architecture of the blog with a total of four `Comments` services.

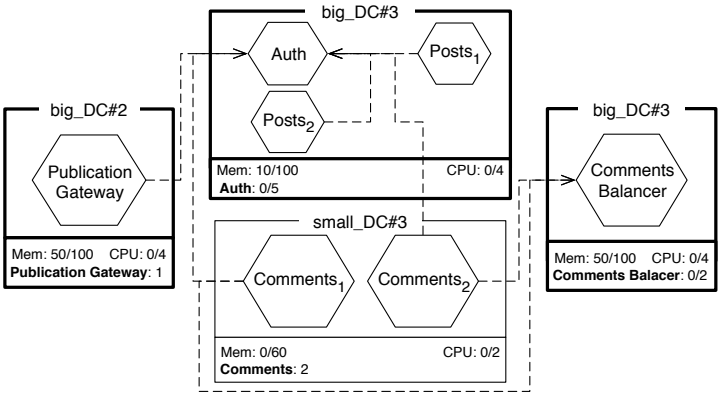

Fig. 3. First Configuration.

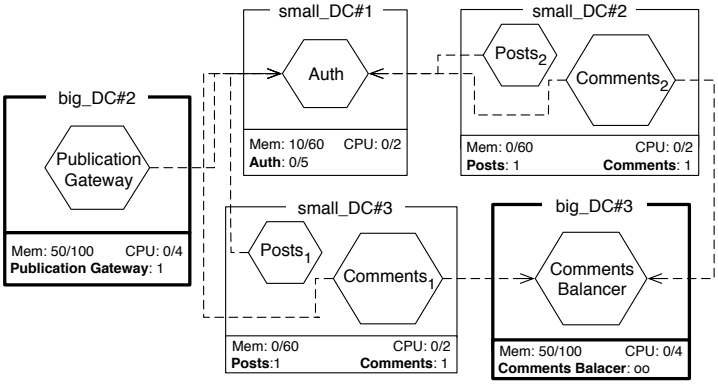

Fig. 4. Fault tolerant configuration.

JRO makes very easy to specify the re-deployment of an architecture. It is sufficient to modify the previous specification by requiring 4, instead of 2, `Comments` services. JRO produces the configuration depicted in Fig. 5. Observe that the increase of 2 `Comments` requires the addition of an additional `Auth` service to handle the increase in authentication requests generated by all the `Comments` services. This is done automatically and the DevOps does not need to handle any dependency between services.

As a final example, let us consider that the profiling of the blog changes. This can be due to a wrong initial profiling or to the introduction of a new version of the services of the blog. In this case, some service of the blog can require more or less resources to work correctly. JRO covers also this case: the DevOps just needs to update the previous profiling and rerun the tool to redistribute the component in the optimal way.

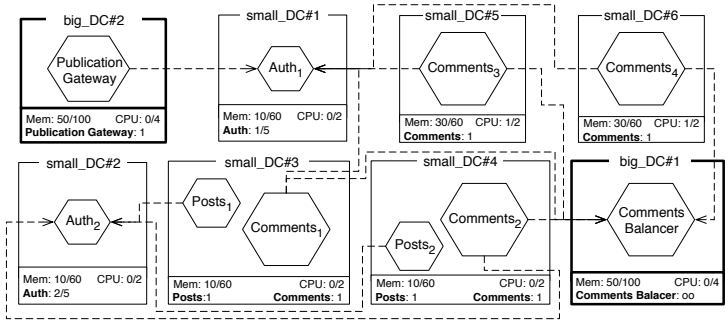

Fig. 5. Configuration with additional 2 Comment service.

3 JRO

In this section, we detail how JRO works. As previously mentioned, the execution phases of JRO are summarized in Figure 1. The deployment of a new configuration or the reconfiguration of an existing one is triggered by the user that enters her desiderata. JRO queries the deployment platform (in our case it is JE) to retrieve the current deployed configuration and the list of the services that could be deployed with their resource needs and dependencies. These data are then encoded and submitted to Zephyrus to obtain a tentative final configuration. This configuration is presented to the user, which may accept it or refuse it by entering a different specification. If a configuration is accepted, it is deployed on the target deployment platform by issuing the commands to install and run the services. The user has only to enter her goals and, if desired, perform the optional step of deciding if accepting or not a given configuration.

JRO can be used in an interactive way to refine the configuration until an acceptable one is obtained. To make this process automatic, JRO requires the services to be annotated with their profiling, i.e., that each service is annotated with its resource consumption, its dependencies, and its capabilities. In JRO annotations are written in a JSON file that, by convention, has the same name and is stored on the same location of the Jolie service. For example, the JSON annotation associated to the `Post` service is the following:

```
1 { "cost":
2     { "Memory": 20, "CPU": 1 } ,
3   "dependencies":
4       { "Auth" : 1 }
5 }
```

At Line 2, we specify that the service requires the use of a 1 CPU and 20 MB of memory[9]. At Line 4, we specify that `Post` depends on the functionali-

[9] This number is given just for illustrative purposes. The real service consumes indeed more resources.

ties provided by the `Auth` service. Hence, to be properly installed, it needs the location of an existing `Auth` to invoke.

In the annotation, it is also possible to quantify the number of other services that can exploit the functionalities provided be the annotated services. This can be done by means of the `provide` property. For example, the `Auth` service is annotated with {`"provide" : 5`}, which indicates that every instance of `Auth` can receive invocation from at most 5 different services.

JRO automatically retrieves the information related to the running SOA by exploiting the JE APIs. In particular, it finds what are the Deployment Components (DC) that are running, their resources (e.g., the number of CPUs and the RAM), and the services deployed on of them. Since the available DCs may not be enough to deploy the desired system, it is possible to specify additional resources to use that may be acquired from a cloud provider and their monetary cost.

The list of deployment components is given as a JSON object with two properties: `DC_description`, which describes the different types of deployment components, and `DC_availability`, which specifies the number of available instances for each of these types. A deployment component type is identified by a name, the list of the resources it provides, and a cost that the user has to pay in order to use it. For instance, the following JSON object defines the possibility of using 5 `c3.large` and 3 `c3.xlarge` Amazon AWS instances as deployment components.

```
1  { "DC_description": [
2    { "name" : "c3.large", "cost" : 105,
3      "provide_resources" : {"CPU" : 2, "Memory" : 375} },
4    { "name" : "c3.xlarge", "cost" : 210
5      "provide_resources" : {"CPU" : 4, "Memory" : 750} } ],
6   "DC_availability": {
7     "c3.large" : 5, "c3.xlarge" : 3 } }
```

The `c3.large` AWS machine is identified as a deployment component type that provides 2 CPUs and 3.75 GB of RAM. When used, this type of deployment component costs 105 dollars per month.

As previously mentioned, the DevOps triggers the execution of JRO by entering the specification of the target configuration. The DevOps does not need to design the final configuration and she rather declares some constraints (e.g., number of services she wants to deploy, co-installation or distribution requirements) of the final configuration. All these goals and desiderata are expressed in a domain specific language called *Service Desiderata Language* (SDA). In the remainder of this section, we first detail this language and then describe the integration of Zephyrus via JRC and how the final configuration, if accepted, is actually deployed in JE.

3.1 Service Desiderata Language (SDA)

The Service Desiderata Language (SDA) is an ad-hoc language created to succinctly state the constraints that the final configuration should entail. As shown

```
1  spec
2    : expr comparisonOP expr | spec boolOP spec | 'true'
3    | 'not' spec | '(' spec ')' ;
4  expr
5    : 'DC[' resourceFilter '|'  simpleExpr ']'
6    | 'DC['  simpleExpr ']'
7    | expr arithmeticOP expr | simpleExpr ;
8  resourceFilter
9    : STRING comparisonOP INT
10   | resourceFilter ';' resourceFilter ;
11 simpleExpr
12   : exprNoDC comparisonOP exprNoDC
13   | simpleExpr boolOP simpleExpr |
14   | 'true' | 'not' spec | '(' spec ')' ;
15 exprNoDC :
16   INT | STRING
17   | exprNoDC arithmeticOP exprNoDC ;
18 comparisonOP : '<=' | '<' | '=' | '>=' | '>' ;
19 arithmeticOP : '+' | '-' | '*' ;
20 boolOP : 'and' | 'or' | 'impl' | 'iff' ;
```

Fig. 6. SDA grammar.

in Fig. 6, which reports the SDA grammar defined using the ANTLR tool[10], a constraint is a specification `spec` of basic constraints `expr comparisonOP expr` (Line 2) combined using the usual logical connectives. These basic constraints specify how many services the user desires to create. An expression `expr` could identify either an integer value or the number of services.

With this expressiveness, it is possible to add constraints that abstract away from the DC. For instance, one might require, as in the running example, the deployment of at least 2 `Post` and 2 `Comments` services as follows.

```
Post >= 2 and Comments >= 2
```

More complex constraints can be stated to restrict the applications installed on the DC. These constraints are expressed (Line 5) with the notation `DC[ resourceFilter | simpleExpr ]` where `resourceFilter` is an optional sequence of constraints on the resources provided by the DC and `simpleExpr` is an expression. `DC[ resourceFilter | simpleExpr ]` denotes the number of deployment components that satisfy the resource constraints of `resourceFilter` and that contain objects satisfying the expression `simpleExpr`. For instance, we can specify that no deployment component having less than 8 CPUs should contain more than one `Post` service as follows.

```
DC[ CPU <= 8 | Post > 1 ] = 0
```

[10] ANTLR (ANother Tool for Language Recognition) - http://www.antlr.org/

It is also possible to express constraints on co-location or distribution. This is an important feature when dealing with performances — e.g., by co-locating services that frequently interact —, or with security or fault handling — e.g., by keeping some kinds of services separated. As an example, consider the case in Section 2 in which we forbid to co-locate the `Post` and the `Comments` services on the same DC. Such requirement is easily stated with the following constraint.

```
DC[ Post > 0 and Comments > 0 ] = 0
```

3.2 JRC

When the specification and the information on the running configuration are retrieved, they must be transformed and encoded in order to exploit the engine of the Zephyrus configurator. This task is performed by JRC, which processes the available information to generate the *universe file* of components required by Zephyrus [7]. Services have to be encoded into Aeolus components since Zephyrus requires as input a representation of the components following the Aeolus model specification [8]. In Aeolus, a component is a grey-box showing relevant internal states and the actions that can be acted on the component to change its state during the deployment process. Each state activates "provide" and "require" ports that represent functionalities that the component offers and needs, respectively.

In this context, a service `S` for JRO can be simply seen as an Aeolus component with two states: an initial state `Init` representing the fact that `S` is not yet deployed, and an `On` state meaning that the service has been deployed. If the service has some initialization parameters (e.g., `Post` requires a service `Auth`) these are seen as require ports.

In Aeolus, it is possible to associate numbers to ports to deal with capacity/replication constraints. The number associated to a *require* port indicates the minimal number of distinct components that should provide resources to satisfy the requirement. The number associated to the *provide* port stands instead for the maximal amount of distinct components that can use the provided functionality. In our setting, the number of service dependencies is therefore the number associated to the *require* port. Dually, the number of services that can use the functionalities of a given service is the number associated to the *provide* of its Aeolus representation.

Zephyrus requires as additional input also the specification file containing the encoding of the constraints that should be satisfied in the final configuration following an ad-hoc specification language, and the location file containing the list of the containers to be used to deploy the components. The generation of these files from the available information is quite straightforward since the Zephyrus specification language is more expressive than SDA because, thanks to the chosen encoding, the notion of component and ports in the Aeolus model collapses into the notion of services (i.e., components and ports share the same domain).

When all the input of Zephyrus is generated, JRC runs the configurator. This is the most computational intensive task of the entire process[11]. We use Zephyrus to compute the cheapest solution satisfying the user desiderata.

3.3 Deployment of the final configuration

When the configuration is returned and it is accepted by the DevOps, JRO removes the services that are deployed but not present in the final configuration and then starts to deploy the new services on the virtual machines defined in the configuration computed by using Zephyrus.

In the final configuration the dependencies between the components are the connection between the services. Since the services are developed in Jolie, satisfying a dependency can be performed simply by changing the configuration of the output port of the dependent service with the appropriate location and the setting of the protocol needed to reach the required service. Services that do not have dependencies are deployed before those requiring these services. In case of a circular dependency (e.g., service A requires service B that requires A), first JRO deploys the services, then it retrieves their inbound connection data, and finally it dynamically rebinds their output ports.

It is important to notice here that, while in principle any suitable platform could be used for the deployment of service, the use of the Jolie Enterprise framework simplify considerably this task.

Jolie Enterprise is structured on two main nodes: the control panel and the cloud node. The Jolie microservices are deployed and run within cloud nodes, while the control panel offers a set of Web APIs for interacting with the cloud nodes by using operations such as `setService`, `startService`, `stopService` and `getServiceList`. Operation `setService` registers a service in the cloud node, `startService` executes it, `stopService` stops its current execution, and `getServiceList` returns the list of all the available services along with its execution status (running, disabled). In our implementation of JRO we have created a service, called `ResourceManager`, which can call Jolie Enterprise APIs in order to get the current configuration of the system, consisting of active services and inactive services. Such a configuration is then passed to JRC to obtain an output containing the new desired configuration for the system. At this stage, the `ResourceManager` calls again the Jolie Enterprise APIs in order to deploy and execute the new configuration.

The Jolie Enterprise is a proprietary solution and therefore is not freely available. Nevertheless, JRC, the core part of the JRO, is open-source and available at https://github.com/jolie/jrc. This tool can be used to support other deployment platforms providing the same functionalities of JE. JRC is provided along with the input and configurations for all the outputs of the running scenario in Section 2.

[11] As formalized in [6], the problem solved by Zephyrus is NP-hard.

4 Related Work and Conclusion

Nowadays, developing applications for the cloud is usually accomplished by relying on the Infrastructure as a Service (IaaS) or the Platform as a Service (PaaS) levels. The IaaS level provides a set of low-level resources forming a "bare" computing environment. Developers pack the whole software stack into virtual machines containing the application and its dependencies and run them on physical machines of the provider's cloud. Exploiting the IaaS directly allows a great flexibility but requires also a great expertise and knowledge of both the cloud infrastructure and the application components involved in the process. The most common solutions for the deployment of a cloud application is still to rely on pre-configured virtual machines (e.g., Bento Boxes [15], Cloud Blueprints [5], and AWS CloudFormation [1]) or to exploit configuration management tools such as Puppet [43] or Chef [40] to better customize the application.

At the PaaS level (e.g., [3, 19]) a full development environment is provided. Applications are directly written in a programming language supported by the framework offered by the provider, and then automatically deployed to the cloud. The high-level of automation comes however at the price of flexibility: the choice of the programming language to use is restricted to the ones supported by the PaaS provider, and the application code must conform to specific APIs. Application in PaaSes are usually scalable and can exploit the elasticity of the cloud to accommodate more requests. However, we are not aware of PaaSes that can guarantee the optimal automatic allocation of services allowing the minimization of the cost of the entire application.

In this work, we combine the flexibility typical of the IaaS level with the high-level automation typical of the PaaS level by allowing the DevOps to specify their SOAs and then automatically deploying the specified SOAs, optimising its costs, its performances, and its resource consumption.

The most similar approach to ours is Aeolus Blender [12] from which we draw inspiration. Blender is a software product for the automatic deployment and configuration of complex distributed software systems in the "cloud". It relies on a configuration optimiser (i.e., Zephyrus as also in our case) and an ad-hoc deployment planner [24] to deploy real-life applications on an OpenStack cloud. However, differently from our tool, Blender requires every service lifecycle to be described with the Armonic formalism [25] which essentially uses state machines to represent the different steps that need to be performed to deploy a service. Due to the fact that Jolie services can be easily deployed and do not need complex iteration patterns to be installed, we were able to simplify the entire deployment process requiring to the user to specify just the resource consumption of the services and thus avoiding the use of a planner to compute the sequence of deployment action to perform. Moreover, differently from Blender, JRO can also deal with configurations where services depend on each other.

Another related work is [10] that relies on Zephyrus to allocate objects to deployment containers starting from a program in modelling language ABS (Abstract Behavioural Specification) where classes are annotated to indicate the resource consumption of their objects.

JRO can be easily extended to handle other services or applications written in different languages. Indeed, we only require that the installation of such components does not involve an interaction with other components and that their dependencies could be configured after their installation. In particular, we can capture and deploy SOA relying on stateless services or application following the best practice of the "immutable server" approach [34, 35]. In any case, our interest lies in how we can suitably change the configuration of a system. This depends on the property that the system supports reconfiguration, which can be achieved in different ways. In this work, we have used Jolie to support the writing and execution of services. As mentioned in Section 1, Jolie services support concurrency by running multiple instances of their behaviour, called processes. Processes in Jolie can be stateful or stateless. In the former case, a popular approach for supporting reconfiguration is using distributed agreement algorithms among the replicated processes [14, 22, 39]. There are other technologies that we could have combined with JRO, e.g., Erlang [2] or other frameworks based upon the actor model [20]. Both Jolie processes and actors are meant as executable instances of a behaviour, to be run inside of a service. The main differences between the two approaches are in what kind of behaviours can be written and in the primitives for communications, e.g., Jolie processes explicitly specify the data used to identify other processes, whereas actors usually leave this duty to another layer and assume that the other actors can be explicitly found via direct references. These differences are orthogonal w.r.t. our work, which focuses on how to change reconfigurations rather than the details of how processes (or actors) are implemented in services.

Apart from this immediate generalization, we see several other directions for future developments in order to obtain a more inclusive and enhanced tool for the automatic and optimised deployment of micro services. First, the human interface part did not receive the due attention so far. We are therefore planning to construct a suitable GUI which allows one to graphically define the desired specification and its modifications, as in the case of the Blender GUI[12].

On a different level, we plan to integrate in our system an existing monitoring functionality of Jolie Enterprise in order to be able to determine the current load of the system and therefore to be able to automatically balance the load, possibly modifying the configuration, in order to maintain some given service level agreements for the deployed services. Suitable extensions of such a monitoring tool could also be used to combine run time checking with static analysis (e.g., based on types) in order to ensure the correctness of the system, and more generally to verify service level agreements along the lines described in [9, 37]. The same techniques can be also exploited to automatize the deployment of system developed by means of choreographic languages [17, 41, 42].

We would also like to address some of the current limitations of JRO due to the use of the Zephyrus configurator. In particular, we would like to extend Zephyrus in order to be able to find the best deployment configuration given

[12] For some example of GUIs we can adopt within JRO we invite the interested reader to see the screen cast at http://www.aeolus-project.org/.

a user-specified maximal cost and a maximal resource consumption. We also intend to add support for annotations with parametric costs that depend on service parameters. Finally, we would also like to tackle the computational aspects involved in the process of finding the optimal configuration allowing the users to exploit heuristics – such as local search techniques – in order to quickly get good but possibly sub-optimal solutions.

A note on Columbus' egg

The idea of integrating Zephyrus with Jolie Enterprise to obtain a tool for the automatic and optimised deployment of microservices is a very simple one, yet it can be the basis of a very effective tool which can have a significant impact on real applications.

In this sense, this idea is in line with one of the most distinguishing features of Frank as a researcher: the strive for simplicity, also when working on very complicate subjects. Indeed Frank has often been looking for "Columbus' eggs", sometime he has found them, and once he actually made public this attitude in front of a distinguished audience. Before the anecdote, a note on these "eggs": "A Columbus egg refers to a brilliant idea or discovery that seems simple or easy after the fact. The expression refers to an apocryphal story in which Christopher Columbus, having been told that discovering the Americas was inevitable and no great accomplishment, challenges his critics to make an egg stand on its tip. After his challengers give up, Columbus does it himself by tapping the egg on the table to flatten its tip" (Wikipedia).

So here is the story. In 1997, at the Thirteenth Annual Conference on Mathematical Foundations of Programming Semantics, Frank was presenting a paper co-authored with one of the authors of this paper. That conference was celebrating also the 65th birthday of Dana Scott, so most of the experts on semantics of programming languages were there. While presenting his paper, Frank mentioned the fact that one of the adopted technical solutions was a kind of Columbus' egg. Having seen that some people in the audience had a strange reaction, Frank asked plainly whether they knew the story. Since many people answered "no", Frank spent almost the rest of his time telling the story of Columbus and his famous egg, rather than presenting the paper.

Thank you Frank, and our best wishes for finding many more Columbus' eggs.

References

1. Amazon. AWS CloudFormation. https://aws.amazon.com/cloudformation/. Last retrieved Jan 2016.
2. J. Armstrong. *Programming Erlang*. Pragmatic Bookshelf, 2013.
3. Microsoft Azure. https://azure.microsoft.com. Last retrieved Jan 2016.
4. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274. ACM, 2013.

5. CenturyLink. Cloud Blueprints. https://www.ctl.io/blueprints/. Last retrieved Jan 2016.
6. R. D. Cosmo, M. Lienhardt, J. Mauro, S. Zacchiroli, G. Zavattaro, and J. Zwolakowski. Automatic Application Deployment in the Cloud: from Practice to Theory and Back. In *CONCUR*, volume 42 of *LIPIcs*, pages 1–16. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015.
7. R. D. Cosmo, M. Lienhardt, R. Treinen, S. Zacchiroli, J. Zwolakowski, A. Eiche, and A. Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, pages 211–222. ACM, 2014.
8. R. D. Cosmo, J. Mauro, S. Zacchiroli, and G. Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239:100–121, 2014.
9. F. S. de Boer and S. de Gouw. Combining Monitoring with Run-Time Assertion Checking. In *SFM*, volume 8483 of *LNCS*, pages 217–262. Springer, 2014.
10. S. de Gouw, M. Lienhardt, J. Mauro, B. Nobakht, and G. Zavattaro. On the Integration of Automatic Deployment into the ABS Modeling Language. In *ESOCC*, volume 9306 of *LNCS*, pages 49–64. Springer, 2015.
11. DevOps. http://devops.com/. Last retrieved Jan 2016.
12. R. Di Cosmo, A. Eiche, J. Mauro, G. Zavattaro, S. Zacchiroli, and J. Zwolakowski. Automatic Deployment of Software Components in the Cloud with the Aeolus Blender. In *ICSOC*, volume 9435 of *LNCS*, pages 397–411. Springer, 2015.
13. Docker Inc. Docker. https://www.docker.com/. Last retrieved Jan 2016.
14. M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.
15. Flexiant. Bento Boxes. https://www.flexiant.com/2012/12/03/application-provisioning/. Last retrieved Jan 2016.
16. M. Fowler and J. Lewis. Microservices. http://martinfowler.com/articles/microservices.html, 2014. Last retrieved Jan 2016.
17. M. Gabbrielli, S. Giallorenzo, and F. Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.
18. D. Georgakopoulos and M. P. Papazoglou. *Service-Oriented Computing*. The MIT Press, 2008.
19. Google App Engine. https://cloud.google.com/appengine/docs. Last retrieved Jan 2016.
20. C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245. William Kaufmann, 1973.
21. Jolie. Programming Language. http://www.jolie-lang.org/. Last retrieved Jan 2016.
22. L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
23. I. Lanese, A. Bucchiarone, and F. Montesi. A framework for rule-based dynamic adaptation. In *TGC*, volume 6084 of *LNCS*, pages 284–300. Springer, 2010.
24. T. A. Lascu, J. Mauro, and G. Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *ICTAI*, pages 213–220. IEEE, 2013.
25. Mandriva. Armonic. https://armonic.readthedocs.org/en/latest/index.html. Last retrieved Jan 2016.
26. J. Mauro and G. Zavattaro. On the Complexity of Reconfiguration in Systems with Legacy Components. In *MFCS*, volume 9234 of *LNCS*, pages 382–393. Springer, 2015.
27. F. Montesi. Hack your way through the microservices revolution. http://www.infoworld.com/article/2903590/application-development/

hack-your-way-through-the-microservices-revolution.html. Last retrieved Jan 2016.
28. F. Montesi. JOLIE: a Service-oriented Programming Language. Master's thesis, University of Bologna, 2010.
29. F. Montesi and M. Carbone. Programming Services with Correlation Sets. In *ICSOC*, volume 7084 of *LNCS*, pages 125–141. Springer, 2011.
30. F. Montesi, C. Guidi, R. Lucchi, and G. Zavattaro. JOLIE: a java orchestration language interpreter engine. *Electr. Notes Theor. Comput. Sci.*, 181:19–33, 2007.
31. F. Montesi, C. Guidi, and G. Zavattaro. Composing Services with JOLIE. In *Proc. of ECOWS*, pages 13–22, 2007.
32. F. Montesi, C. Guidi, and G. Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.
33. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, volume 8052 of *LNCS*, pages 425–439. Springer, 2013.
34. K. Morris. Immutableserver. http://martinfowler.com/bliki/ImmutableServer.html, 2013. Last retrieved Jan 2016.
35. Netflix. Building with legos. http://techblog.netflix.com/2011/08/building-with-legos.html, 2011. Last retrieved Jan 2016.
36. Nginx. Adopting microservices at netflix: Lessons for architectural design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, 2015. Last retrieved Jan 2016.
37. B. Nobakht, S. de Gouw, and F. S. de Boer. Formal Verification of Service Level Agreements Through Distributed Monitoring. In *ESOCC*, volume 9306 of *LNCS*, pages 125–140. Springer, 2015.
38. B. OASIS. Web services business process execution language. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html, 2007.
39. D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *USENIX*, pages 305–320, 2014.
40. Opscode. Chef. http://www.opscode.com/chef/. Last retrieved Jan 2016.
41. M. D. Preda, M. Gabbrielli, S. Giallorenzo, I. Lanese, and J. Mauro. Dynamic Choreographies - Safe Runtime Updates of Distributed Applications. In *COORDINATION*, pages 67–82, 2015.
42. M. D. Preda, S. Giallorenzo, I. Lanese, J. Mauro, and M. Gabbrielli. AIOCJ: A choreographic framework for safe adaptive distributed applications. In *SLE*, pages 161–170, 2014.
43. Puppetlabs. Puppet. http://puppetlabs.com/. Last retrieved Jan 2016.