Automatic Deployment of Services in the Cloud with Aeolus Blender*

Roberto Di Cosmo[1], Antoine Eiche[2], Jacopo Mauro[3], Stefano Zacchiroli[1], Gianluigi Zavattaro[3], and Jakub Zwolakowski[1]

[1] Univ. Paris Diderot, Sorbonne Paris Cité, PPS, CNRS (France)
[2] Mandriva S.A.(France)
[3] Dep. of Computer Science and Engineering - Univ. Bologna / INRIA FoCUS (Italy)

Abstract. We present *Aeolus Blender* (Blender in the following), a software product for the automatic deployment and configuration of complex service-based, distributed software systems in the "cloud". By relying on a configuration optimiser and a deployment planner, Blender fully automates the deployment of real-life applications on OpenStack cloud deployments, by exploiting a knowledge base of software services provided by the Mandriva Armonic tool suite. The final deployment is guaranteed to satisfy not only user requirements and relevant software dependencies, but also to be optimal with respect to the number of used virtual machines.

1 Introduction

The cloud market is now a reality able to modify companies behaviour. The needs for solutions or efficient tools to support development activities for the profitability of the company is becoming more and more important. The new perspectives of IT usage (mobility, social networks, Web 2.0, Big Data) has brought the world into a new digital revolution. The first consequence is an explosion in the needs for computing and storage. According to an IDC study [23], digital data created in the world will increase to 40 Zettabytes in 2020. Faced with this, CIOs need to change, becoming more and more service-based and evolve their IT towards virtualized platforms and cloud (IAAS, PAAS, and SAAS) to address issues related to infrastructure growth, the need for power and provision computing resources on demand from internal or third party.

The CloudIndex study [34], conducted in partnership with Capgemini and Orange Business Services, indicates that 30% of respondents (300 companies)

* This work was supported by the EU projects FP7-610582 *Envisage: Engineering Virtualized Services* (`http://www.envisage-project.eu`), FP7-644298 *HyVar: Scalable Hybrid Variability for Distributed, Evolving Software Systems* (`http://www.hyvar-project.eu`), and the ANR project ANR-2010-SEGI-013-01 Aeolus. It was partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org.

have a cloud strategy and three quarters of them are planning to dedicate resources to this strategy. Two main directions emerge: homogenise the application portfolio and facilitate the deployment of applications.

Driven by this business need, several tools have been developed and used routinely to help system architects and administrators to automate at least some of the deployment and configuration phases of the complex service application in the cloud. For instance, configuration managers like Puppet [35] and Chef [33] are largely used by the "DevOps" community [13] to automate the configuration of package-based applications. Domain specific languages like ConfSolve [22] or Zephyrus [10] can be used to compute—starting from a high-level partial description of the application to be realised—an (optimal) allocation of the needed software components to computing resources. Tools like Engage [15] or Metis [26] synthesise the precise order in which low-level deployment actions should be executed to realise the desired application.

Despite the availability of such tools, the mainstream approach for deploying cloud applications is still to exploit pre-configured virtual machines images, which contain all the needed software packages and services, and that just need to be run on the target cloud system (e.g., Bento Boxes [16], Cloud Blueprints [8], and AWS CloudFormation [2]). However, the choices of the services to use (e.g., WordPress installed with Apache or Nginx, with NFS or GlusterFS support) lead to an explosion of configurations that can hardly be matched by the offered set of pre-configured images. Moreover, pre-configured images often force the user to run her application on specific cloud providers, inducing an undesirable vendor lock-in effect.

Arguably, the adoption of pre-configured images is still the most popular approach due to the lack of *integrated solutions* that support system designers and administrators throughout the entire process, ranging from the high-level declarative description of the application to the low-level deployment and configuration actions. In this paper we describe Blender, a software product maintained by Mandriva, which is based on the approach taken by the Aeolus project [7] that strives to overcome the limitations of using pre-configured images.

More precisely, Blender integrates three independent tools:

Zephyrus [10] A tool that automatically generates, starting from a partial and abstract description of the target application, a fully detailed service oriented architecture indicating which components are needed to realise such application, how to distribute them on virtual machines, and how to bind them together. Zephyrus is also capable of producing *optimal* architectures, minimising 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.

Metis [25,26] A planner that generates a fully detailed sequence of deployment actions to be executed to bring an application to a desired configuration (e.g., as the one produced by Zephyrus). Plans are made of individual deployment actions like installing a software artefact, changing its state according to its internal life-cycle, provisioning virtual machines, etc. Metis relies on an

ad hoc planning algorithm that exploits service dependencies to prune the search space and produce the needed deployment steps very efficiently (i.e., provably in polynomial time). Metis could produce plans involving hundreds of components in less than one minute.

Armonic [27] A collection of scripts and libraries that, starting from a knowledge base of information about available software artefacts, allows for the deployment of software applications and services on several Linux distributions. Each software artefact has a list of states, and each state performs actions to deploy and configure the associated software component on the target distribution.

By exploiting the above tools, Blender realises a framework that supports system architects and administrators all the way from the design phase down to the deployment on a cloud infrastructure. The present paper extends [10, 25–27] by offering a tighter integration among the three tools and by adding an actual user interface that turns Blender into a real, production-ready solution.

A declarative approach is adopted throughout Blender, according to which only a minimal amount of information needs to be initially given. For instance, it is sufficient to indicate the main services the application should expose to application users, plus non-functional requirements like the desired level of replication (for load balancing and/or fault tolerance) for critical service instances. From this initial information, Blender computes the complete architecture of the application and supports the administrator in the deployment phases, during which only context-dependent configuration variables needs to be manually instantiated.

Paper structure Section 2 presents Blender from the point of view of the users, by showing how to realise a real-life, moderately complex service-based cloud application: a replicated, load-balanced deployment of the WordPress blogging platform. Section 3 enters into more details, by showing what happens behind the scenes when Blender is used to realise the case study of Section 2. Section 4 points to the open source implementation of Blender. Before concluding, Section 5 reviews related literature and tools.

2 Deploying a WordPress farm with Blender

We consider the deployment of a so-called "WordPress farm", i.e., a load balanced, replicated blogging service based on WordPress.[4] A typical approach to deploy this kind of application is to rely on specific services listed in Table 1.

Instead, of adopting pre-configured virtual machines, on which instances of these software artifacts have been installed, our approach starts from reusable, abstract *descriptions* of these services, collected in the *Armonic* knowledge base. Only a limited amount of case-by-case information must be provided by the user. From these elements (the abstract description of the available software artifacts

[4] https://wordpress.com/

Table 1: Software components used to deploy a WordPress farm

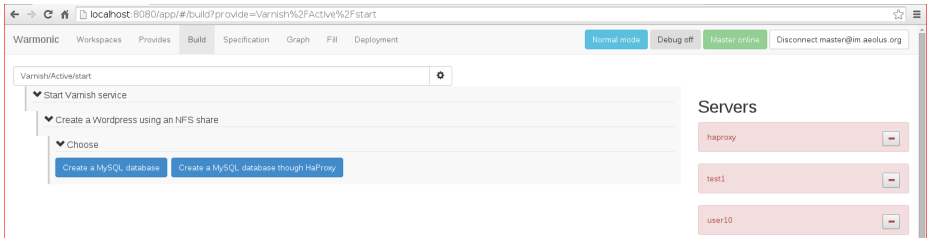
WordPress a blogging tool based on PHP;
Galera Cluster for MySQL: a multi-master cluster of MySQL databases synchronously replicated;
HAProxy a load balancer for TCP and HTTP-based applications spreading requests across multiple servers;
Varnish an HTTP accelerator designed for content-heavy dynamic web sites supporting dynamic load balancing;
HTTP Server a software component serving web server requests;
NFS client/server an application implementing a distributed file system.

and the additional specific user-defined information), Blender synthesises and then deploys the entire application.

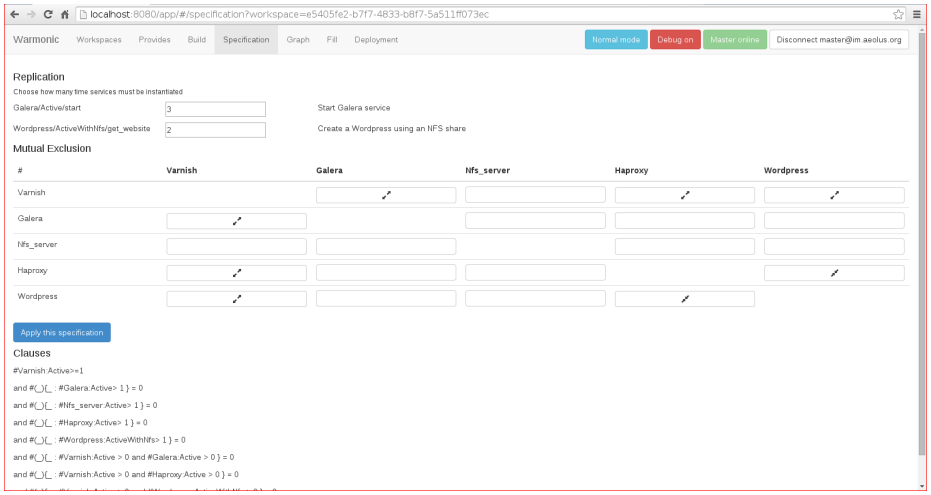
When executing Blender, the first piece of information the user will need to provide is an indication of the desired front-end service to be deployed, in this case the *Varnish* load balancer. Based on this initial piece of information, Blender will guide the user through an interactive question/answer phase, during which the main services needed to complete the application are chosen, and service-dependent additional information are asked to the user. The kind of information requested to the user in this second phase typically deals with desired installation policies, which usually vary on a case-by-case basis. For instance, as shown in Figure 1a, once *Varnish* and *WordPress with NFS* is chosen, two different solutions for the database are proposed (i.e., single shared installation or multi-master replication based on *Galera*). As shown in Figure 1b the user can then also specify that specific service pairs cannot be co-installed on the same virtual machine (e.g., WordPress cannot be installed with Galera for performance reasons) or that two services have to be co-installed (e.g., WordPress and HAProxy are installed on the same machine for fault tolerance reasons). This information cannot be automatically inferred, as it depends on specific properties like the expected workload, so user guidance is required.

Once these pieces of information are entered, Blender translates the description of the Armonic services into the Aeolus component-based model representations used by Zephyrus and Metis. In particular, Zephyrus synthesises the full architecture of the installation, indicates how many and which kind of virtual machines are needed, and distributes the services onto such machines. Subsequently, Metis computes the sequence of deployment actions needed to reach the final configuration produced by Zephyrus.

The computed plan is not ready to be executed yet, because some system-level configuration parameters are still missing (e.g., administrative passwords, credentials, etc) and should be provided by the user. Blender asks the user for these information and, once all the configuration data is available, it proceeds to create the virtual machines computed by Zephyrus on the target OpenStack



(a) Selection choice: MySQL or a MySQLs replicated via HaProxy?



(b) Deciding and forbidding co-installation.

Fig. 1: User inputs for WordPress installation

infrastructure. Then, Blender uses Armonic to deploy and configure components by executing state changes, according to the Metis deployment plan.

In our example, during the interactive Q/A phase we have chosen Varnish to balance the traffic between 2 WordPress servers, NFS support, and 3 Galera instances. Moreover, we chose to inhibit co-installation of WordPress with Galera or the NFS Server, and to install HAProxy on every machine where WordPress is installed. The only additional piece of information asked by Blender as configuration data were the admin passwords for the DBs and HAProxy services.

The final architecture produced by Blender is depicted in Figure 2. The installation requires 6 machines, 3 running Debian and 3 MBS (Mandriva Business Server). It took approximately 7 minutes to deploy such architecture on a simple OpenStack infrastructure deployed on an Intel Xeon server with 4 cores. The computation of the final configuration and the deployment plan was almost instantaneous: the execution of Zephyrus and Metis required less than a second while the most time consuming task was the deployment of the Galera Cluster

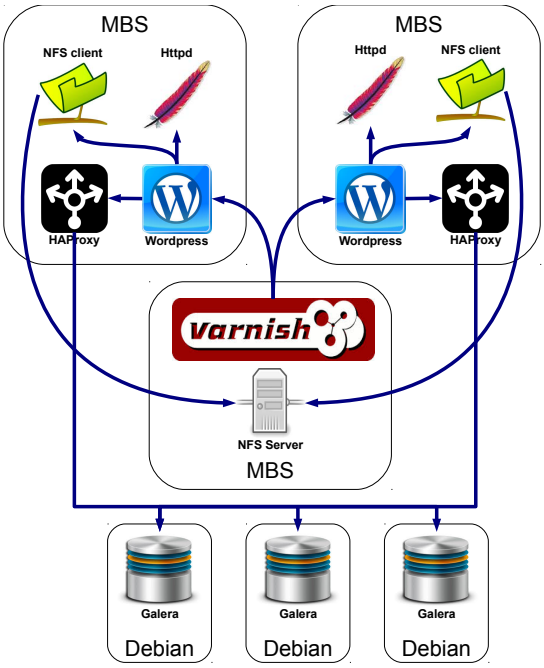


Fig. 2: Deployed WordPress farm architecture

that required 3 minutes and half (1 minute and 10 seconds for every instance). The other services were deployed instead in less than a minute.

3 Blender Internals

As depicted in Figure 3, Blender is intended to be used in combination with an XMPP server and an OpenStack cloud installation. Blender is realised as an XMPP client that wraps and combines the tools Zephyrus, Metis, and Armonic and exposes its functionalities via *ad hoc* commands.[5] Basically, such commands are used to launch Zephyrus, view the graph representing the computed final configuration, fill the configuration variables, and perform the deployment actions according to the plan produced by Metis. It is possible to interact with Blender via a Web user interface or the command line. An advantage of this architecture is that new elements can be added by wrapping them as simple XMPP clients. For instance, other IaaS offers can be easily added in addition to the currently supported OpenStack.

Blender relies on scripts that integrate Zephyrus, Metis, and Armonic following the execution flow depicted in Figure 4. Such workflow requires two distinct

[5] http://xmpp.org/extensions/xep-0050.html

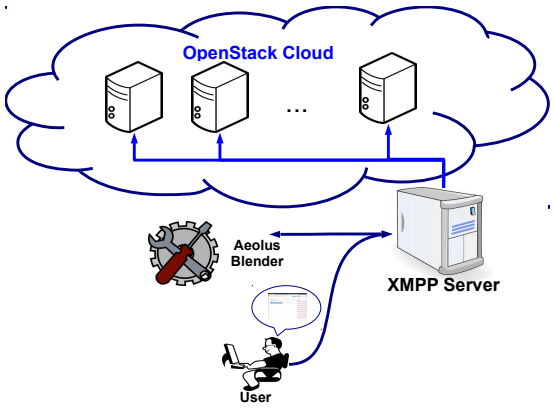


Fig. 3: Blender environment

inputs: an Armonic service repository, and a high-level description of the desired application to be deployed.

Armonic associates to every service a life-cycle that can be conceptually viewed as a state machine representing the different steps that need to be performed in order to deploy the service. For example, a service could have an associated state machine with 4 states: *not installed*, *installed*, *configured*, and *active*. Each state is usually associated to a collection of actions that need to be performed to enter into or exit each state, and actions that can be invoked on the service when a state has been entered. Technically speaking, states are implemented as Python classes, and actions are class methods. Each state has at least *enter* and *leave* hooks that are invoked when a state is entered and exited. Actions to be performed require the instantiation of a group of variables capturing information such as the required services, or the needed configuration values (with their default or optional values). In some cases, the required functionalities should be local when they must be provided in the same host where the component is deployed. For instance, in our running example, the NFS client is a local dependency of WordPress because an active WordPress needs an NFS client to be installed on the same machine.[6]

The first step of the Blender execution flow is querying the user to gather her desiderata. This task is performed by the *Builder* that asks the user for the services she wants to install, their desired replication constraints, and information about the need or impossibility to co-install onto the same host specific pairs of services.

When the user has entered all this information, the *Builder* queries the Armonic service repository and generates:

[6] For more information related to Armonic services we refer the interested reader to [28]

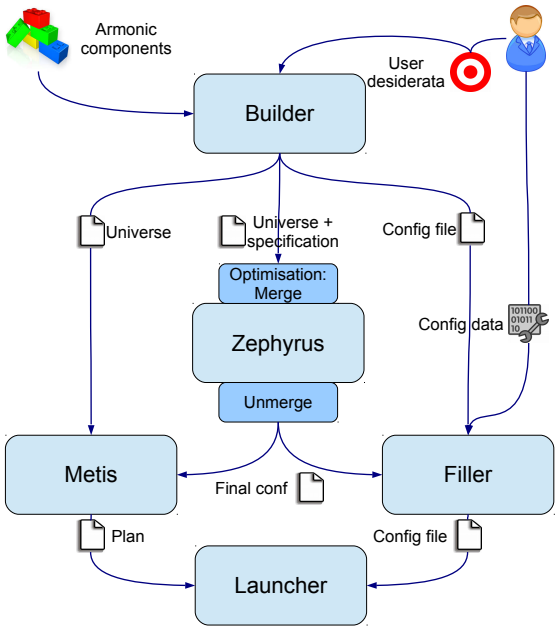


Fig. 4: Blender execution flow

specification file containing the encoding of the constraints that should be satisfied in the final configuration expressed in the specification language used by Zephyrus;

universe file containing the Aeolus component representations [11] of available services, in the JSON format used by both Zephyrus and Metis;

configuration data file containing indications about the system-level configured data needed to configure Armonic services. Some of them, if not already provided, will have to be entered by the user later on (e.g., credentials). Other data may be inferred from the configuration parameters of other components (e.g., WordPress can suggests a database name to its database dependency).

An excerpt of the specification file generated from user input for the running example is as follows:

```
Varnish:Active >= 1
and #(_){_ : #Galera:Active > 0 and
             #Wordpress:ActiveWithNfs > 0 } = 0
```

The first line requires a final configuration to have at least one Varnish service in the Active state. The second and third lines forbid the co-installation of Galera with WordPress. This is obtained requiring that the number of virtual machines having at least one Galera and one WordPress is 0.

The *universe* file is generated by encoding Armonic services into Aeolus components, which faithfully capture states and transitions. In Aeolus terminology,

methods exposed by states become provide ports. These methods and special state methods (e.g., *enter* and *leave*) can expose dependencies which become require ports. As an example, a graphical representation of the Aeolus model for the WordPress service of our example is given in Figure 5. WordPress is depicted as a 5 state automaton, requiring the *add_database* functionality from the HAProxy to be configured, the *start* and *get_document_root* functionalities to be active, and the *mount* functionality from the NFS client to support the NFS. When active with NFS support, WordPress will provide the *get_website* functionality to other services.

Since the Aeolus model abstracts away from configuration data, these are stored in the *configuration data* file, which will be later used to perform deployment.

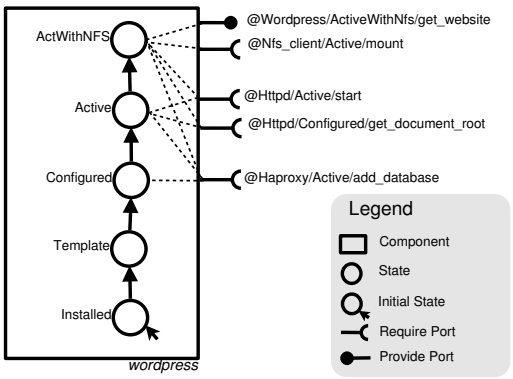


Fig. 5: Aeolus representation of the WordPress component

The universe file generated by the *Builder* is subsequently post-processed in order to merge together services that must be installed on the same machine. For instance, in our example, the WordPress services needs an NFS client to be installed on the same machine. These two services are therefore merged together obtaining a new service that consumes the sum of the resources. This simplifies the input of Zephyrus, reducing the number of services to be managed, thus speeding up the computation of the final optimal configuration, i.e., the one that uses the smallest number of virtual machines.

The solution computed by Zephyrus is then processed to decouple the services that were previously merged together. Indeed, while Zephyrus abstracts away from the internal life-cycles of the service, Metis needs to consider individual automata to compute the needed deployment actions. Metis then takes the post-processed output of Zephyrus and the original Universe file to compute a deployment plan to reach the final configuration.

At this point the user is asked to provide the missing configuration data for the final deployment. The configuration data file generated by the *Builder* is

processed together with the output of Zephyrus to detect which services should be installed and then fill the missing data querying the user if needed. This task is performed by a component dubbed *Filler* that uses several *Armonic* libraries to deduce configuration variables from default values when possible.

Once all the configuration information are filled in, the plan produced by Metis and the configuration data file are passed to the *Launcher*, a Python tool that acquires and bootstraps the virtual machines indicated in the output of Zephyrus using the OpenStack API, and transforms the abstract deployment actions generated by Metis into concrete actions that are sent to Armonic agents running on individual virtual machines.

4 Implementation

The complete toolchain presented in this paper is publicly available and released as free software, under the GPL license. Blender consists of approximately 5k lines of Python and is available from `https://github.com/aeolus-project/blender`. As Blender is an integrator, it has as software dependencies the tools it integrates:

- Zephyrus that amounts to about 10k lines of OCaml and is available from `https://github.com/aeolus-project/zephyrus`;
- Metis that amounts to about 3.5k lines of OCaml and is available from `https://github.com/aeolus-project/metis`;
- Armonic that amounts to about 5k lines of Python, plus glue code for service life-cycles written in shell script or Augeas and is available from `https://github.com/armonic/armonic`.

Screencasts showing the use of Blender to deploy different WordPress installations are available at `http://blog.aeolus-project.org/aeolus-blender/`.

5 Related work

Currently, developing an application for the cloud is 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. At the PaaS level (e.g., [5, 18]) 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.

To deploy distributed applications at the IaaS level, different languages with their deployment engines have been proposed. In this context, one prominent work is represented by the TOSCA (Topology and Orchestration Specification for Cloud Applications) standard [32], promoted by the OASIS consortium [31] for open standards. TOSCA proposes an XML-like rich language to describe an application. Deployment plans are usually manually specified using the BPMN or BPEL notations, workflow languages defined in the context of business process modelling. Other similar deployment languages approaches are CloudML [17], the Fractal-based language extending the OVF standard [14], and approaches supporting the OASIS CAMP standard [30] such as Apache Brooklyn [3]. All these approaches allow a form of abstraction of the configuration to deploy. However, contrary to what can be done in Blender, the configuration to deploy have to be fully specified with all its configuration parameters and service dependencies. Moreover, due to their lack of a production-ready tool support, these approaches have seen a limited practical adoption so far. For this reason, as previously mentioned, the most common solution for the deployment of a cloud application is still to rely on pre-configured virtual machines (e.g., Bento Boxes [16], Cloud Blueprints [8], and AWS CloudFormation [2]).

Another common, but more knowledge-intensive solution, is to use configuration management tools which allows application managers to avoid some of the drawbacks of pre-configured images (e.g., lack of flexibility, lock-in mechanism) at the price of requiring a deep knowledge and expertise of the management tool and the configuration to realise.

One of the most similar approach to Blender is *Engage* [15], a tool that automatically generates the right order in which deployment actions should be performed to deploy some services. Engage avoids circular service dependencies and therefore the deployment plan can be generated by a simple topological sort of the graph representing the service dependencies. This is a significant limitation w.r.t. Blender because circular dependencies can arise in practice when, for instance, configuration information flow between services in both directions (consider, e.g., a master database that first requires the slave authentication and subsequently provides the slave with a dump of the database). Moreover, Engage does not provide a production-ready tool support.

Other commercial configuration management tools are instead Terraform [19], Juju [24], Cloudify [9], Rudder [29], and Scalr [37]. Terraform [19] is a configuration tool to describe both resources and services used to remotely execute a sequence of low-level deployment actions. However, it lacks a mechanism to describe the relationships between software services. Juju [24] is a tool and approach by Canonical, dedicated to the management of Ubuntu-based cloud environments. It is more a software orchestration framework than a proper configuration tool as it focuses on services and their relationships, to the detriment of many low-level aspects. Cloudify [9] is a software suite for the orchestration of the deployment and the life cycle of applications in the cloud. It is based on a meta language to describe a deployment plan and a monitoring software used to follow the application behaviour and to trigger a set of tasks to perform. Rud-

der [29] is an open source web solution dedicated to the production, automation and configuration of application deployment using CFEngine [6] and providing real-time monitoring of the application trying to ensure its compliance using a rule base mechanism. Scalr [37] is an open source web application for cloud services management. Scalr uses the API of major cloud providers to deploy templates containing a Scalr agent that allows for fine grained interaction with supported services such as MySQL, Apache, etc.

All these commercial configuration management tools are used to declare the services to be installed on each machine and their configuration. However, contrary to Blender, the burden of deciding where services should be deployed, and how to interconnect them is left to the operator. Furthermore, no offering computes the final and optimal configuration starting from a partial specification, nor can devise the order in which deployment actions must be performed.

Other related works are ConfSolve [22] and Saloon [36]. ConfSolv is an academic approach that relies on a constraint solver to propose an optimal allocation of virtual machines to servers, and of application services to virtual machines. Saloon instead computes a final configuration by describing a cloud application using a feature model extended with feature cardinalities. Unfortunately, ConfSolve does not compute the actions needed to reach the computed configuration while Saloon automatically detects inconsistencies but, differently from Blender, it does not offer the ability to minimise the number of resources and virtual machines to be used.

Another relevant research direction leverages on traditional planning techniques and tools coming from artificial intelligence. In [4, 20, 21] off-the-shelf planning solvers are exploited to automatically generate (re-)configuration actions. To use these tools, however, all the deployment actions with their preconditions and effects need to be properly specified in a formalism similar to the Planning Domain Definition Language (the *de facto* standard language for planners). The Blender approach, on the other hand, relies on simpler and natural service descriptions (i.e., state machines describing the temporal order of the service configuration actions).

Finally, we would like to underline that Blender integrates various tools, some of which have been detailed elsewhere. Zephyrus has been presented in [10]. The present paper extends [10] in several ways: it integrates Metis to drive deployment on the basis of an actual deployment plan; it adds an actual user interface turning Blender into a real, production-ready solution; and it offers tighter integration among the three tools. Thanks to Metis, which supports the synthesis of infrastructure-independent plans, Blender could also be used with other deployment engines, while deployment as described in [10] relied on hard-coded internal mechanisms of Armonic. The new GUI supports the user in lively step-by-step visualisation of the effect of each deployment action. This functionality is effective if actions are executed in sequence. For this reason the current version of Blender serialises the actions synthesised by Metis (which, a priori, are parallelisable); future versions of Blender will consider parallel deployments by further improving its GUI.

Metis has been presented in [25]. The tool validation in that paper was done by using automatically generated descriptions of components. The integration of Metis in Blender described in this paper, on the other hand, represents the validation of Metis on real use-cases.

6 Conclusions

We have presented Blender, a tool exploiting a configurator optimiser, an *ad hoc* planner, and a deployment engine to automate the installation and deployment of complex service-based cloud applications. Blender does not rely on predefined recipes, but on reusable service descriptions that are used as building blocks to synthesise a fully functional configuration satisfying the user desiderata. Blender is easy to use, comes with a web graphical interface, and requires as input just those specific configuration parameters that cannot be deduced from the service descriptions. It is an open source project started by Mandriva S.A., a software company specialised in Linux and open-source software providing innovative and easy to use offerings and business services to professionals. Mandriva with its research unit Innova is planning to exploit Blender offering a new server management solutions to speed up the current trend of migration of physical servers to cloud environments.

Since there is no standard benchmark for application deployment, as a future work we plan to define qualitative and quantitative evaluation mechanisms by first describing a series of deployment tasks that can later be used to evaluate both the improvements of future Blender versions and for comparison with possible future competitors. Moreover, as done in [12], we would like to compare the quality of the automatically generated deployment plans against those (manually) devised by DevOps. Since Blender always produces an optimal final configuration, its solution can be used to prove that an existing handmade solution is optimal. If instead the solutions differ due to the fact that some constraints were not specified or forgotten by the user, we may capture the missing requirements and then use them to ease and standardize the deployment of future similar deployment tasks.

Furthermore, we would like to reduce the deployment time of Blender by following the maximal parallelisable plan suggested by Metis. In this way, the deployment actions that are found to be independent may be executed in parallel. Moreover, noticing that replicated servers (e.g., the Debian machines containing the replicated database in our *WordPress* example) share part of their deployment plan, we would like to use live virtual machine cloning instead of re-creating instances that will end up being similar from scratch.

Further optimizing service deployment actions is outside the scopes of Blender. These actions are indeed intrinsic to the nature of the services to be deploy and depend just on their Armonic definition. However, we would like to tackle instead the time required by Zephyrus to compute the final optimal configuration. Indeed, as shown in [38] for some complex and large *WordPress* deployment scenarios, the computation of the optimal configuration may become the most

computational intensive task of the toolchain. Even though for scenarios of reasonable size (for a typical professional *WordPress* installation) less than one minute of computation is needed, in our biggest stress tests (i.e., in one case we required Zephyrus to compute a solution for a system having 103 software components distributed over 86 machines) we experienced computation times of more than 20 minutes. To reduce the time required by Zephyrus in these cases we therefore plan to adopt portfolio solvers (e.g., [1]) or exploit heuristics (e.g., local search techniques) to quickly get good but possibly sub-optimal solutions.

Finally we also plan to integrate other public IaaS solutions (such as Amazon EC2, RackSpace, or Google Compute Engine) directly as well as exploiting and interface with other services libraries and tools such as Juju [24] or Apache Brooklyn [3].

References

1. Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. A Multicore Tool for Constraint Solving. In *IJCAI*, pages 232–238, 2015.
2. Amazon. AWS CloudFormation. http://aws.amazon.com/cloudformation/.
3. Apache Software Foundation. Apache Brooklyn. https://brooklyn.incubator.apache.org/.
4. Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Journal*, 15(3):265–281, 2007.
5. Microsoft Azure. http://azure.microsoft.com.
6. Mark Burgess. A Site Configuration Engine. *Computing Systems*, 8(2):309–337, 1995.
7. Michel Catan, Roberto Di Cosmo, Antoine Eiche, Tudor A. Lascu, Michael Lienhardt, Jacopo Mauro, Ralf Treinen, Stefano Zacchiroli, Gianluigi Zavattaro, and Jakub Zwolakowski. Aeolus: Mastering the Complexity of Cloud Application Deployment. In *ESOCC*, volume 8135 of *LNCS*. Springer, 2013.
8. CenturyLink. Cloud Blueprints. http://www.centurylinkcloud.com/products/management/blueprints.
9. Cloudify. http://getcloudify.org/.
10. Roberto Di Cosmo, Michael Lienhardt, Ralf Treinen, Stefano Zacchiroli, Jakub Zwolakowski, Antoine Eiche, and Alexis Agahi. Automated synthesis and deployment of cloud applications. In *ASE*, pages 211–222. ACM, 2014.
11. Roberto Di Cosmo, Jacopo Mauro, Stefano Zacchiroli, and Gianluigi Zavattaro. Aeolus: A component model for the cloud. *Inf. Comput.*, 239:100–121, 2014.
12. Stijn de Gouw, Michael Lienhardt, Jacopo Mauro, Behrooz Nobakht, and Gianluigi Zavattaro. On the Integration of Automatic Deployment into the ABS Modeling Language? In *ESOCC*, 2015.
13. DevOps. http://devops.com/.
14. Xavier Etchevers, Thierry Coupaye, Fabienne Boyer, and Noel De Palma. Self-Configuration of Distributed Applications in the Cloud. In *CLOUD*, pages 668–675. IEEE, 2011.
15. Jeffrey Fischer, Rupak Majumdar, and Shahram Esmaeilsabzali. Engage: a deployment management system. In *PLDI*, pages 263–274. ACM, 2012.
16. Flexiant. Bento Boxes. http://www.flexiant.com/2012/12/03/application-provisioning/.

17. Glauco Estacio Gonçalves, Patricia Takako Endo, Marcelo Anderson Santos, Djamel Sadok, Judith Kelner, Bob Melander, and Jan-Erik Mångs. CloudML: An Integrated Language for Resource, Service and Request Description for D-Clouds. In *CloudCom*, pages 399–406. IEEE, 2011.
18. Google App Engine. https://developers.google.com/appengine/.
19. HashiCorp. Terraform. https://terraform.io/.
20. Herry Herry and Paul Anderson. Planning with Global Constraints for Computing Infrastructure Reconfiguration. In *CP4PS*, 2012.
21. Herry Herry, Paul Anderson, and Gerhard Wickler. Automated Planning for Configuration Changes. In *LISA*. USENIX Association, 2011.
22. John A. Hewson, Paul Anderson, and Andrew D. Gordon. A Declarative Approach to Automated Configuration. In *LISA*, pages 51–66, 2012.
23. IDC. Executive summary: A universe of opportunities and challenges. http://www.emc.com/collateral/analyst-reports/idc-the-digital-universe-in-2020.pdf, 2012.
24. Juju, devops distilled. https://juju.ubuntu.com/.
25. Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. A Planning Tool Supporting the Deployment of Cloud Applications. In *ICTAI*, pages 213–220. IEEE, 2013.
26. Tudor A. Lascu, Jacopo Mauro, and Gianluigi Zavattaro. Automatic Component Deployment in the Presence of Circular Dependencies. In *FACS*, volume 8348 of *LNCS*, pages 254–272. Springer, 2013.
27. Mandriva. Armonic. http://armonic.readthedocs.org/en/latest/index.html.
28. Mandriva. Armonic, Lifecycle anatomy. http://armonic.readthedocs.org/en/latest/lifecycle.html.
29. Normation. Rudder. http://www.normation.com/en.
30. OASIS. Cloud Application Management for Platforms. http://docs.oasis-open.org/camp/camp-spec/v1.1/camp-spec-v1.1.html.
31. OASIS. Organization for the Advancement of Structured Information Standards (OASIS). https://www.oasis-open.org.
32. OASIS. Topology and Orchestration Specification for Cloud Applications (TOSCA) Version 1.0. http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.html.
33. Opscode. Chef. http://www.opscode.com/chef/.
34. PAC. Cloudindex study. http://www.cloudindex.fr/sites/default/files/PAC%20CloudIndex%20-%20Webinar%20de%CC%81cembre%202014.pdf, 2014.
35. Puppetlabs. Puppet. http://puppetlabs.com/.
36. Clément Quinton, Andreas Pleuss, Daniel Le Berre, Laurence Duchien, and Goetz Botterweck. Consistency checking for the evolution of cardinality-based feature models. In *SPLC*, pages 122–131. ACM, 2014.
37. Scalr Cloud Management. http://www.scalr.com/.
38. Jakub Zwolakowski. *A formal approach to distributed application synthesis and deployment automation*. PhD thesis, Univeristé Paris Diderot – Paris 7, 2015.