CONSOLAS: A Model-Based Tool for Automatic Configuration and Deployment of Cloud Applications

Hui Song, Franck Chauvel, Franck Fleurey, Nicolas Ferry, Arnor Solberg
SINTEF ICT, Oslo, Norway. Email: {first.last}@sintef.no

***Abstract*—This paper demonstrates CONSOLAS, an automatic tool for the configuration and deployment of software applications in cloud. We provide Domain-Specific Modelling Languages for application developers to specify the components in the application and the possible cloud resources to host them, as well as the constraints between them. Based on the specifications, CONSOLAS assists application operators in configuring and deploying the application automatically. Operators only need to provide simple hints on how they want to configure the application, and the tool generates a complete and valid configuration and deploys it. CONSOLAS also supports operators in refining the configuration both before and after the deployment: Operators make small and partial changes on an existing configuration, and the tool automatically completes the changes and performs incremental deployment. We demo the tool with a sample case, and a video can be found at https://youtu.be/V9WWeFi1ZD8 .**

I. INTRODUCTION

Many people choose to run software applications in a cloud environment. They obtain applications from public repositories such as GitHub, *configure* them according to their own requirements, and *deploy* them on the virtual machines (VMs) that they provision from a private or a public cloud. In this way, these *application operators* provide their own services to customers based on the existing applications.

Configuration and deployment of cloud applications is challenging. Unlike traditional desktop applications or the modern mobile Apps that can be downloaded in a monolithic way and launched with simple set-ups, a cloud application usually consists of multiple components that can be hosted by different VMs. Moreover, for performance or security purpose, the same component may have multiple instances scattered in different VMs. Each component may have its own parameters and depend on several other components. To configure a cloud application, operators need to decide how many instances to create for each component type, set the parameters for each instance, link them together, and assign them to proper VMs. These decision points are sophisticatedly related, because of the many constraints existing in the application or the cloud environment, and therefore, it is difficult and tedious to reach a globally valid configuration. A configuration is also hard to refine and adjust - even a small change may require a lot of subsequent revisions on different parts of the configuration to make it globally valid.

In this paper, we present the CONSOLAS (CONstraint SOLving for Architecture Setup) tool for model-based, automatic configuration and deployment of cloud applications. We provide domain specific modelling languages (DSMLs) for application developers to specify the components and their target cloud resources, as well as the constraints and best patterns to configure their applications. Using the specifications as a reference, CONSOLAS helps operators to configure and deploy the application iteratively: In each iteration, the operators give simple hints on what configuration they expect or make small changes on the existing configuration, and CONSOLAS returns a complete and valid configuration for the next iteration. The configurations are in the form of architectural models, which are intuitive for operators to understand and will be deployed fully automatically into the mainstream cloud platforms by CONSOLAS. The technical basis of the tool is our research on multi-cloud modelling, language engineering, and constraint solving on software architectures.

In the rest of this paper, we use a sample cloud application (introduced in Section II) to show how developers use CONSOLAS to specify their applications (in Section IV), and how operators use it to automatically configure and deploy the applications (in Section V). A demonstration video on the same case can be found at https://youtu.be/V9WWeFi1ZD8 .

II. SAMPLE CASE

SMARTGH [1] is a route planning application developed by Trinity College Dublin. It searches routes on a city scale taking into consideration the sensor data such as pollution and noise. Figure 1 summarises its main components: `Sensors` collect data from city sensors or public data sources and populate them into a `Redis` database. `Hoppers` link the sensor readings to OpenStreetMap maps to plan smart routes. `Webs` receive routing requests and display the results from `Hoppers`. An adaptable `Load-Balancer` forwards requests to the user-preferred `Web`. For the sake of diversity, developers provide different `Hoppers` specialized for foot routing, fast car routing (`FCHopper`), etc., and also different types of `Sensors`. Each component is wrapped as a docker image, which can be obtained at hub.docker.com/u/songhui/.

Third-party operators can *configure* and *deploy* SMARTGH on their own IaaS (Infrastructure as a Service) environments, providing a diverse of smart routing services. Figure 2 shows a sample configuration, where three `Hoppers` and their `Webs` are hosted by two virtual machines. One `Hopper` utilizes the data from a `PollutionSensor`, via a `Redis` database. The essential part of this configuration, as is highlighted by greyed boxes in Figure 2, is to decide what types of hopper and sensors to provide, and how many instances to create for each type. This part is interested to the operator, because it defines the features provided by his/her routing service. However, the

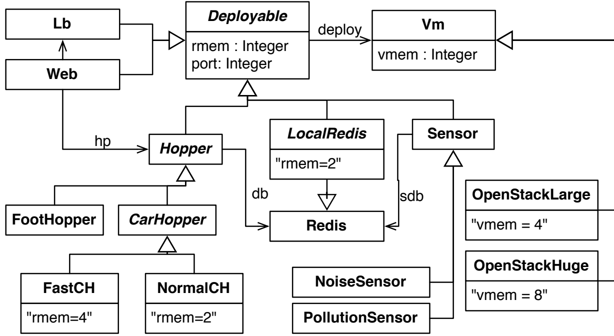

Fig. 1. SMARTGH components

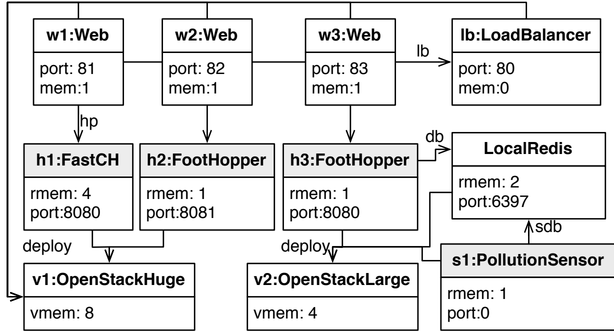

Fig. 2. SMARTGH components

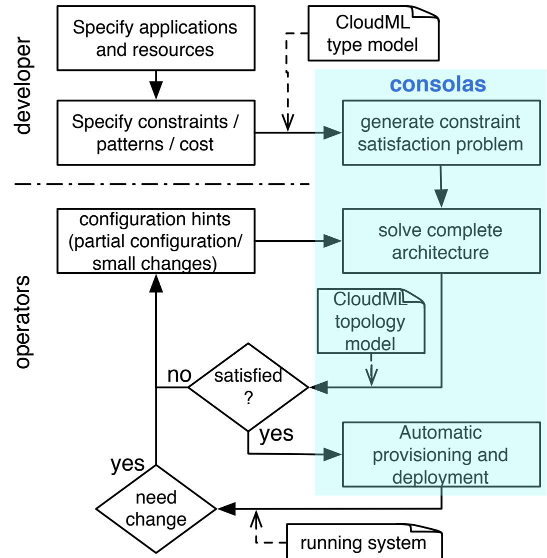

Fig. 3. CONSOLAS process overview

operator has additional tedious jobs to do to reach a proper SMARTGH configuration, such as to arrange the component instances into proper VMs, add auxiliary components (*i.e.*, web, load balancers, database), link the components, assign port numbers, etc. These additional jobs are difficult because there are many constraints that the operator must obey. For example, a VM cannot host so many components that the total memory consumption of these components exceeds the VM's capacity; two components on the same VM should have different port numbers; every `Hopper` needs its own `Web`; a `Hopper` and its database (if there is one) should better be hosted by the same VM, etc. An ideal situation would be that that operators only suggest the essential part of the configuration, and all the tedious part is automatically generated according to the constraints and patterns. This is the *automatic configuration* supported by CONSOLAS. It also deploys the configuration automatically into the IaaS platform.

Application configuration is not a once-for-all task, and operators may need to adjust their SMARTGH configurations either before they are deployed, or after they have been running for a while, to optimise the service or to meet new user requirements. For example, when there are more user requests for walking routes, the operator may want to add a new `FootHopper` into the configuration in Figure 2. For such adjustment, a small change may require complicated co-changes to reach a valid configuration again. For the sample scenario, the co-changes will be to add a `Web` for the new hopper, migrate a component from `v1` to `v2` to save space for the new components, linking it to the database, *etc.*. CONSOLAS also supports such agile and incremental reconfiguration: Operators make the changes that they cares, and the tool completes the required co-changes automatically, and deploys all the changes to the system incrementally.

III. TOOL OVERVIEW

Figure 3 summarises the process of using CONSOLAS to configure and deploy a cloud application. The shadowed part of the flowchart is the internal process of the tool, and the rest is the activities and artefacts required by tool users. We identify two different roles of tool users, *i.e.*, the *developers* and the *operators*, and divide the process into two parts, each for a different user role.

Developers specify the types of components and resources. After that, they specify the constraints that a proper configuration should follow. The artefacts produced by the two activities are a type model specified in CLOUDML (a DSML for cloud specification developed by SINTEF) and a set of First-Order Logic constraints (specified by a python-based DSL, implemented for CONSOLAS). Details and samples of the specifications can be found in Section IV.

The specifications are machine-readable to CONSOLAS, which helps operators configure their own services from the application. The process starts from the operators' providing a partial configuration as a hint. CONSOLAS then automatically generates a valid configuration, and visualize it to the operator. If the operators are not satisfied with the configuration, they can make changes on it and ask CONSOLAS to complete the changes. This loop continues until a satisfying configuration is obtained, and CONSOLAS will automatically deploy the configuration. Finally, after the application is already running, operators can still change the configuration. Their changes, along with the necessary subsequent changes suggested by CONSOLAS, will be deployed into the system in an incremental way. Details about how operators provide partial configuration and modify it can be found in Section V.

```
1  deployment model SmartGHDiv
2  provider openstack_nova : ...
3  types{
4   vm OpenStackLarge{
5    provider : openstack_nova
6    ram : 4096, core : 1,
7    os : "ubuntu", os64,
8    provided host ubuntuPrv
9    ...}
10  internal component FootHopper{
11   required host ubuntuReq
12   provided communication hopperPrv
13   required communication redisReq
14   resource DockerImage{
15    download: "sudo docker pull..."
16    start: "cd ~; sudo docker run..."
17     }
18  communication hp
19   from hopperReq to hopperPrv...
20 }
21 instances{\\left empty for auto config}
22 }
```

Fig. 4. Excerpt of CLOUDML specification for SMARTGH

IV. SPECIFICATION OF RESOURCE AND APPLICATION

CONSOLAS needs application developers to specify the component and resources types, and the constraints on them. The specification is similar to the "INSTALL" manuals of traditional software applications, but are machine-readable to the CONSOLAS tool. We provide two languages to support the specifications.

CONSOLAS integrates an existing language named CLOUDML for the specification of applications and cloud resources. Figure 4 shows an excerpt of the CLOUDML model for SMARTGH. In the `types` section, the SMARTGH developers define the resource and component types, i.e., the boxes in Figure 1. For a resource type, such as the VM `OpenStackLarge`, they specify the cloud provider, some common arguments across all the instances of this type (such as 4096MB of RAM, 64bit Ubuntu OS, etc.), and the hosting service they can provide. The developers also define a component type `FootHopper`: It requires a hosting service (not surprisingly the same as the one provided by `OpenStackLarge`) and a communication service from a database. The component can be downloaded and started using the specified docker commands. Finally, a `communication` relation connects the communication services provided by `Hoppers` to the service required by `Web`. The `instances` section of this model is left empty, which will be filled with the automatically generated configuration.

CONSOLAS provides another domain-specific language for developers to specify constraints on top of the type specifications. We design and implement the language as an embedded one on the basis of Python. Figure 5 shows three sample constraints specified by the CONSOLAS constraint language. The first constraint specifies that for any `Hopper`, there must be a `Web` for it. The second one specifies that if a `Hopper` needs a database, then the two components should better be deployed on the same VM. This is a soft constraint with a priority as 50 (in the middle between 0 and 100). Finally, the last constraint sets up a cost to use any VM instance, which is correlated to the memory size of the VM. This soft constraints prevents CONSOLAS from using up too much cloud resources.

```
1 g_forall((x, Hopper),
2   g_exists((y, Web), hp(y)==x)))
3 g_forall([(x, Hopper)],
4   soft(50, Implies(alive(db(x)),
5     deploy(x)==deploy(db(y)))))
6 g_forall([(x, VM)],
7   soft(mem(x)*10, Not(alive(x))))
```

Fig. 5. Sample constraints for SMARTGH

V. AUTOMATIC CONFIGURATION AND DEPLOYMENT

CONSOLAS helps operators automatically configure and deploy a cloud application in their own ways. In this section, we first use a simple scenario to illustrate the configuration and deployment process, and then briefly introduce the research approaches that enable this automatic process.

A. A configuration and deployment scenario

As a sample scenario, We show how an operator uses CONSOLAS to achieve the SMARTGH configuration as described in Section II and Figure 2, and deploy it on a private cloud.

Before configuration, the operator needs to have the application and resource specification as described in Section IV, and an account from a cloud provider (in this case, it is SINTEF's private cloud on OpenStack).

In the first iteration, the operator gives no input to CONSOLAS (*i.e.*, an empty hint), and CONSOLAS returns an initial configuration, with only a `BasicHopper` and a `Web` hosted by one VM. This is obviously one of the smallest valid configuration of SMARTGH, and CONSOLAS does not add any additional component because it has a cost to do so (*i.e.*, breaking the software constraints like Line 6 of Figure 5).

The initial configuration does not satisfy the operator, who expects two instances of `FootHoppers`, one instance of `FastHopper`, and a `PollutionSensor`. The operator inputs this requirement into CONSOLAS as the following configuration hints.

```
typeof(h1)==FastCH !
typeof(h2)==FootHopper !
typeof(h3)==FootHopper !
And(db(h3)==sdb(s1),typeof(s1)==PollutionSensor)!
```

The hints are essentially simple constraints on the configuration, and are written in the same CONSOLAS constraint language as in Section IV. The first three hints require three hoppers to be of specific types, and "!" means that the hints apply to all the iterations afterwards. The last hint requires that `h3` connects to a `PollutionSensor` via a database. The names of component instances (*e.g.*, `h2`, `s1`) are syntax sugars for operators to avoid the length expressions such as "there exists three Hoppers, they are different, and their types

are..." Receiving these hints, CONSOLAS automatically returns the configuration that has been shown in Figure 2. Only the greyed component instances are directly required by the hints, but all the rest in the configuration is automatically generated by CONSOLAS. For example, it generates two `Webs` for the two new `Hoppers`, according to the constraint at Line 1 of Figure 5, enlarges `v1` into an `OpenStackHuge` VM in order to be able to host all the `Webs` and two `Hoppers`. Then it instantiates a `LoadBalancer` because there are more than one `Webs`. Finally, according to the last hints, it creates a `PollutionSensor s1` and a `LocalRedis` between `s1` and `h1`. According to the constraints in Figure 5 Line 3, `h3` and `r1` are deployed to the same `v2`.

Now the operator is satisfied with the configuration, and she provides CONSOLAS the endpoint and the account credential of her target cloud platform and launches the deploy command. CONSOLAS will automatically create two VMs, and instantiates 9 docker containers and link them together according to the configuration. After the automatic deployment process, the operator can get the IP address of `v1` (where the load balancer is deployed) from the deployment log, and use port 80 to access the routing services.

After the system is running, the operator can still modify the configuration. For example, for performance purpose, she can require `w3` and `h3` to be hosted in the same VM:

```
deploy(w3)==deploy(hp(w3))
```

Due the resource limitation, CONSOLAS will create a new VM in `OpenStackHuge`, migrate all the components on `v2`, as well as `w3` to the new VM, and finally terminate `v2`.

B. Research behind the tool

The automatic configuration and deployment of CONSOLAS are based on our research of constraint-driven architecture configuration and CLOUDML, respectively. CLOUDML is a language for the modelling of cloud resources and topology. CONSOLAS translate the CLOUDML model together with the constraints into a Satisfactory Modulo Theory (SMT) problem: Component instances are represented by enumeration items. Types, attributes, and connection between components are represented by uninterpreted functions. The constraints are translated into First Order Logic assertions on these functions, and can be either hard or soft. During the configuration process, CONSOLAS first transform the operators' hints into constraints and integrate them into the SMT problem. After that, it launches an constraint solving process to search for a solution to the SMT problem, *i.e.*, giving an interpretation to each of the functions. The solution satisfies all the hard constraints and minimize the total costs of violated soft constraints. For incremental configuration, CONSOLAS will also generate a set of additional soft constraints according to the previous configuration, so that the new configuration will have as little deviation as possible from the previous one. Due to the space limitation, we will not go into the details of the SMT problem and the transformation. Interested readers can refer to our previous publication [2]. CONSOLAS transforms the solution back into a CLOUDML instance model for automatic deployment. The deployment is done by the CLOUDML framework [3]. It first instantiates the required VMs, and then logs-in to the VMs to execute the `download`, `configure` and `start` commands specified with each component.

VI. RELATED WORK

Chef [4] and Puppet [5] are the state-of-the-art cloud management tools, and both of them support the configuration of cloud applications based on scripting languages. Scripts are useful when defining command sequences to configure and deploy individual components, and these tools provide mechanisms to reuse scripts for existing components. Yet operators still need to manually design what components and resources to use and how to link them together. From this point of view, CONSOLAS's automatic architecture configuration is an ideal complement to these tools.

The automatic configuration in CONSOLAS shares the similar idea with the research on auto-completion and auto correction of modelling tools, such as Egyed's work for UML [6] and Xiong et al.'s work on Feature Model [7]. However, model editors suggest completions based on general-purpose rules or constraints, whereas CONSOLAS allows developers to define application-specific constraints. Moreover, as a tool for automatic deployment, CONSOLAS always looks for complete configurations rather than scattered completion suggestions.

VII. CONCLUSION

This paper introduces the CONSOLAS tool for automatic configuration and deployment of cloud applications. Based on developers' specification on the application, resource and constraints, CONSOLAS generates configurations of the application and deploys them automatically to the cloud environment. Future work will be focused on improving the usability, such as providing a web-based GUI and a set of template for application and constraint specification. We will also investigate how to infer the user preference from their refinement iterations.

ACKNOWLEDGEMENT

This work is supported by the EU FP7-ICT-2011-9 No. 600654 DIVERSIFY project

REFERENCES

[1] V. Nallur, A. Elgammal, and S. Clarke, "Smart Route Planning Using Open Data and Participatory Sensing," in *Open Source Systems: Adoption and Impact*. Springer, 2015, pp. 91–100. [Online]. Available: https://github.com/DIVERSIFY-project/SMART-GH

[2] H. Song, X. Zhang, N. Ferry, F. Chauvel, A. Solberg, and G. Huang, "Modelling adaptation policies as domain-specific constraints," in *Model-Driven Engineering Languages and Systems*, 2014, pp. 269–285.

[3] N. Ferry, H. Song, A. Rossini, F. Chauvel, and A. Solberg, "Cloud MF: Applying MDE to Tame the Complexity of Managing Multi-cloud Applications," in *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, 2014, pp. 269–277.

[4] *Cloud Management Chef*, https://www.chef.io.

[5] *The Puppet Lab*, https://puppetlabs.com/.

[6] A. Egyed, "Fixing inconsistencies in uml design models," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007, pp. 292–301.

[7] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki, "Generating range fixes for software configuration," in *Software Engineering (ICSE), 2012 34th International Conference on*. IEEE, 2012, pp. 58–68.