Formal Verification of Orchestration Templates for Reliable Deployment with OpenStack Heat

Adja Ndeye Sylla, Karine Guillouard, Frédéric Klamm, Meryem Ouzzif, Philippe Merle, Souha Ben Rayana, Jean-Bernard Stefani

Formal Verification of Orchestration Templates for Reliable Deployment with OpenStack Heat*

1st Adja Ndeye Sylla
Orange Labs,
Rennes, France
adjandeye.sylla@orange.com

2nd Karine Guillouard
Orange Labs,
Rennes, France
karine.guillouard@orange.com

3rd Frédéric Klamm
Orange Labs,
Rennes, France
frederic.klamm@orange.com

4th Meryem Ouzzif
Orange Labs,
Rennes, France
meryem.ouzzif@orange.com

5th Philippe Merle
Spirals project-team,
Inria Lille - Nord Europe,
Lille, France
philippe.merle@inria.fr

6th Souha Ben Rayana
Univ. Grenoble-Alpes, INRIA,
CNRS, Grenoble INP, LIG,
Grenoble, France
souha.ben-rayana@inria.fr

7th Jean-Bernard Stefani
Univ. Grenoble-Alpes, INRIA,
CNRS, Grenoble INP, LIG,
Grenoble, France
jean-bernard.stefani@inria.fr

***Abstract*—In the context of Network Functions Virtualization (NFV), telecommunication systems are more and more deployed on the cloud, using orchestration engines such as OpenStack Heat. Heat takes as input templates that describe the components of the target system and automatically performs the deployment. This prevents consumers of cloud services from handling the challenges of manual deployment. However, deploying such systems remains challenging. Indeed, the templates given to Heat may contain errors that can lead to a failed or a partial deployment, thus compromising the systems reliability. To handle this challenge, we propose a formal approach and a tool for the verification of templates consistency prior to launching their deployment. A case study is presented to validate the approach.**

***Index Terms*—Formal verification, OpenStack, Heat, NFV**

I. Introduction

Today's systems, in both IT (e.g., smart buildings and cities) and Telecommunication (e.g., 5G [1], solutions that rely on Network Functions Virtualization (NFV) [2] [3]) domains, consist of many heterogeneous and interacting components. These systems are more and more deployed using cloud technologies, especially to increase their performance, scalability, and accessibility. Examples of cloud management technologies are Microsoft Azure [4], IBM Cloud [5], OpenNebula [6], and OpenStack [7]. OpenStack is among the most used technologies in the NFV context [8] [9]. It is open-source, and has a modular architecture. It also provides an orchestration engine, named Heat [10], for the automatic deployment of systems.

Heat takes as input a template that describes the components of the target system and automatically performs the deployment. This allows consumers of cloud services to not handle the challenges of a manual deployment. Indeed, they just have to provide Heat with the template of the target system. This template can be hand-written or even generated by a tool. However, deploying such a system remains challenging. Indeed, the templates given to Heat may contain errors that, if not detected early, can lead to a failed or partial and useless deployment. This causes waste of resources, and reliability issues because one may wrongly assume that the deployment succeeded. In addition, consumers of cloud services may lose a lot of time when deploying an erroneous template. Indeed, they launch their deployment and notice after a delay, that can be long, that the deployment failed. In this case, they have to manually try to detect the error. This can also take a lot of time, and may be not trivial especially for large templates.

To solve the challenges of reliable deployments with Heat, we propose a formal approach for the verification of templates consistency. This verification is done prior to launching the deployment and allows the detection of different types of errors: syntax, type and consistency. Our contributions are:

1) a review of the HOT specification [11] [12] (language used to design Heat templates) and its formalization;
2) the development of a tool that allows consumers of cloud services to verify their templates before deployment;
3) a case study in the NFV context to validate our approach.

This paper is structured as follows. Section II first discusses related work. Then, Section III gives the background notions. Section IV presents the review of the HOT specification and its formalization. Section V describes the implemented tool. Section VI presents a case study in the NFV context. Finally, Section VII concludes the paper and gives future directions.

II. Related Work

Several approaches, based on formal methods, have been proposed for the reliability of deployments in the cloud [13].

In [14], the authors target the NFV domain and propose a solution to both compute VNFs placement and verify if reachability policies (e.g., isolation, latency) can be met before deployment. The VNFs placement and the reachability problem are formally modeled as clauses resolved by a solver. However, this approach does not enable templates verification.

Heat provides a set of tools for the verification of templates before deployment. These tools consist of the Heat parser, the

* This work is funded by Orange Labs in the context of the <I/O Lab>. The authors thank the IRT b<>com members who provided the source code for the UGW use-case, particularly Emmanuel Gouleau and Philippe Bertin.

heat template-validate command, and the *dry-run* option [15]. These tools perform syntax and type checking on a template. However, using them requires users to connect to a running OpenStack platform even if their aim is to just verify the template without deploying it. Moreover, these tools do not detect errors that are related to the consistency of templates.

III. Background

This section first introduces HOT. Then, it presents Location Graphs and Alloy which are used in the proposed approach.

A. HOT

HOT [11] is a YAML-based language used by Heat. This language allows to describe a system as a set of resources of different types (e.g., virtual machines, physical servers, networks and ports). These resources are defined in a template.

```
1  heat_template_version: stein
   resources:
3    aaa_vnf :
       type: OS::Nova::Server
5      properties:
         networks:
7          - port: { get_resource: aaa_vnf_secure_port }
         # other required properties
9    hss_vnf :
       type: OS::Nova::Server
11     properties:
         networks:
13         - port: { get_resource: hss_vnf_secure_port }
         # other required properties
15   secure:
       type: OS::Neutron::Net
```

Listing 1. Example of *Heat* template

Listing 1 presents an example of *Heat* template. This template is written with the latest version of *HOT* (*stein*). It describes a system that consists of two virtual machines (*aaa_vnf*, *hss_vnf*), and one network (*secure*). Each machine has a port that connects it to the network (cf. lines 7 and 13).

HOT relies on a specification that defines its core concepts [11] and the supported resource types [12]. This specification allows consumers of cloud services to know how to design templates that can be automatically deployed with Heat. However this specification is mostly written in natural language and is therefore, possibly ambiguous, inconsistent, and error-prone. Hence, based on this specification, users may design erroneous and inconsistent Heat templates. To prevent this, we decide to review and formalize the HOT specification.

B. Location Graphs

`Location Graphs` (`LG`) [16] [17] is a formal framework for modeling heterogeneous and distributed component-based systems. It provides us with a reference meta-model for the formal modeling of cloud configuration languages. In this paper, `LG` is used to formally model the *HOT* specification.

A `Location Graph` is a set of `Locations`. Each `Location` is a locus of concurrent computations as in process calculi with localities such as the distributed π-calculus [18] or kell calculi [19]. From a software engineering point of view, a `Location` can be seen as a component or as a connector, as in component-based models [20]. Each `Location` has a `Name` for its identification, a `Sort` and a `Process` respectively for its encapsulation and its behavior, as well as a set of required and provided `Roles`. `Roles` correspond to the points of attachment of a `Location`, in order to enable its interaction with other `Locations`. `Location Names`, `Sorts`, `Processes`, and `Roles` are all instances of `Value`.

A `Location Graph` has four invariants: (1) `Locations` are uniquely named, (2) the required and provided `Roles` of a `Location` are disjoint, (3) a `Role` is provided by a single `Location`, (4) a `Role` is required by a single `Location`. The `Location Graphs` concepts are implemented in Alloy.

C. Alloy

Alloy [21] provides a lightweight formal language and an automated analyzer to specify and verify systems. It is chosen because it is widely known, in the formal methods community, open source, and accessible as its language has few concepts.

The *Alloy* language is based on a first order relational logic [22]. Its concepts include *signature* with fields for describing the components of the target system and their interactions, and *fact* for defining a set of invariants of the system. It also provides *commands* for validating the system.

An example of such *command* is *run*. When executed, this command triggers the *Alloy Analyzer* which finds one or several instances of the system, as a constraint satisfaction problem resolved by a solver, if its model is consistent (i.e. all constraints are satisfiable). When the model is inconsistent, the analyzer highlights the set of constraints that cannot be satisfied. When an instance is found, it can be visualized graphically and/or analyzed in order to verify its properties.

```
module LocationGraphs                                                          2

sig LocationGraph{                                                             4
  locations : set Location }

sig Location{                                                                  6
  name: one Name,
  process: one Process,                                                        8
  sort: one Sort,
  provided: set Role,                                                          10
  required: set Role }
                                                                               12
fact UniquelyNamedLocation {
 all lg: LocationGraph|no disj l1, l2: lg.locations|l1.name = l2.name }        14

// other signatures and facts of Location graphs                               16

run Model {} expect 1                                                          18
```

Listing 2. *Alloy* implementation of `Location graphs`

Listing 2 shows the implementation of `Location Graphs` as a module in *Alloy*. A `LocationGraph` is a *signature* (*sig*) with a set of `Locations` (lines 3-4). A `Location` is a *signature* with one mandatory `Name`, `Process`, and `Sort` (lines 7-9). A `Location` also has provided and required `Roles` (lines 10-11). Lines 13-14 show the `UniquelyNamedLocation` invariant specifying that distinct `Locations` of a `LocationGraph` must have different `Names`. This module has a *run* command to verify `LocationGraph` and find instances, if there is at least one.

Fig. 1 shows an instance of `LocationGraph` found by the *Alloy Analyzer*, after the execution of the *run* command (line 18 in Listing 2). This instance has two `Locations`, two `Names`, one `Process`, and one `Sort`. Note that the `Locations` have distinct `Names` to satisfy the `UniquelyNamedLocation` invariant. `Processes` and `Sorts` could be shared between `Locations`, as in Fig. 1.

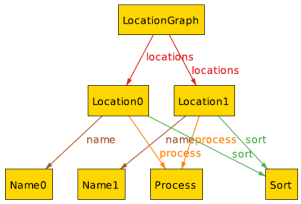

Fig. 1. `LocationGraph` instance found by *Alloy Analyzer*

IV. Review and Formalization of the HOT Specification

This section first describes the *HOT* specification. Then, it shows how this specification is reviewed and then formalized.

A. Description of the HOT Specification

The paper targets the latest HOT specification (*stein*). The HOT core specification states that a template has 7 sections:

- **heat_template_version:** defines the version of HOT that is used to design the template. This section is mandatory;
- **description:** is an optional string to explain the template;
- **parameters:** optionally defines the input parameters that have to be provided at deployment time. This allows to customize the template and reuse it in different contexts.
- **parameter_groups:** specifies how the input parameters are grouped. For example, parameters related to network (e.g., IP addresses, CIDR) can belong to the same group;
- **resources:** defines a set of resource types to be deployed;
- **outputs:** optionally defines the outputs to give a feedback to users after the deployment (e.g. dynamic IP address);
- **conditions:** optionally defines a set of conditions, to be evaluated, based on the values of the input parameters.

The HOT types specification [12] is also written in natural language but its YAML version can be retrieved from a running OpenStack platform. This YAML version is readable by machines and can, therefore, be used for generation purposes. This specification defines the supported resource types. These types are provided by the OpenStack services and are grouped in domains (e.g., computing, networking). For instance, the Nova service provides computing resource types. An example of these types is OS::Nova::Server (virtual machine). The Neutron service provides networking resource types such as OS::Neutron::Net and OS::Neutron::Port. Each resource type has a support status, a set of properties, and a set of attributes.

B. Review and Formalization of the HOT Specification

The aim of reviewing the HOT specification is to verify its consistency and its correctness. To perform the review, we (1) formally model the HOT specification with Location Graphs and Alloy, (2) find instances of the obtained model with the Alloy Analyzer and analyze the instances. This allows us to detect about 63 errors in the HOT specification. These errors are related to `missing invariants that prevent inconsistent behaviors`. In the following, the modeling of the HOT specification is first presented. Then, two examples of errors are given. Finally, the formalization of the HOT specification, to correct the detected errors, is shown.

1) Modeling the HOT specification: the HOT core is first modeled by writing, systematically from the specification, an Alloy module named *HOT.als*. This is done by defining an inheritance relationship and a mapping between the HOT core concepts and those of Location Graphs. A *Template* inherits from `LocationGraph` as both can be seen as a component-based system. *Resource* inherits from `Location`, both can be seen as a system component. *Parameter_group*, *Parameter*, *Output*, and *Condition* inherit from `Value`. A run command is written to find *Template* instances. Then, the model of the HOT types specification is generated by a dedicated Python script. It takes as input the YAML version of the HOT types specification and generates an Alloy module named *HOT_types.als*. This module includes, for each resource type, a set of signatures, and a run command to find its instances.

2) Examples of detected errors: let us consider, in Fig. 2, an instance of HOT template found by the Alloy Analyzer. In this instance, a network has two subnets with the same address (10.0.0.0/24). This behavior is inconsistent. However, this instance is found by the analyzer. The reason is that there is, in the HOT specification, no invariant that prevents this.

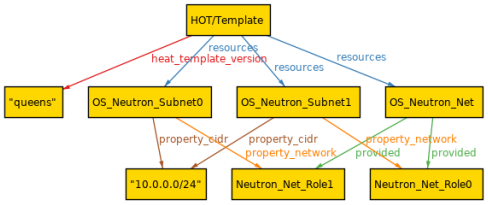

Fig. 2. Two subnets with the same address in the same network

Fig. 3 shows another instance found by the analyzer. This instance consists of 2 virtual machines and 1 port attached to both machines (the port has 2 provided roles and each of them is attached to one of the virtual machines). This behavior is inconsistent and, thus, it must be avoided through an invariant.

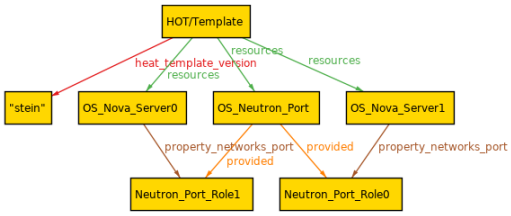

Fig. 3. Two virtual machines with the same port

3) Formalizing the HOT specification: the aim is to correct the errors detected in the HOT specification, by taking into account the missing invariants. This formalization is done by writing in Alloy a fact for each detected error. These facts are added in the *HOT.als* (resp. HOT_types.als) module for the errors related to HOT core (resp. supported resource types).

For instance, Listing 3 shows the Alloy fact written to correct the error presented in Fig. 3. This fact states that distinct resources of type *OS::Nova::Server* must have different ports.

```
fact no_distinct_servers_with_the_same_port {
  all disj s1, s2: OS_Nova_Server|#(s1.property_networks_port.~provided &
                                    s2.property_networks_port.~provided)=0}
```

Listing 3. *Fact* for *OS::Nova::Server* and *OS::Neutron::Port*

V. Implemented Templates Verification Tool

This tool takes as input a Heat template with its environment file, if there is one. This file defines the external resources that are used in the template and gives values to the parameters.

The tool is based on the formalized *HOT* specification and on the *Alloy Analyzer*. It consists of (1) a parser that performs syntax and type checking, and (2) a generator that translates the template and its environment file into a formal Alloy model. This model consists of a *signature*, a *fact*, and a *run* command. The *signature* declares the *parameters*, *parameter_groups*, *resources*, *outputs* and *conditions* sections of the template. The *fact* affects values to the fields of the sections. The *run* command is executed by the *Alloy Analyzer*, to check if the template is compliant with the formalized *HOT* specification (i.e. the template does not contain errors and can be deployed). If the template is deployable, an instance of it is found. Otherwise, its list of errors is returned to the user.

VI. Case Study

To validate our approach, we target a virtual network service developed by the IRT b<>com. This network service (cf. Fig. 4), called Unifier Gateway (UGW) [23], is a convergent access control solution based on NFV technologies. It consists of a user and a control planes. The latter is made up of 6 VNFs:

- **AAA**: Authentication, Authorisation & Accounting;
- **DHCP**: Dynamic Host Control Protocol;
- **HSS**: Home Subscriber Server;
- **MME**: Mobility Management Entity;
- **S/PGW-C**: Serving & PDN Gateway Control plane;
- **GW-U**: SDN Controller (SDN-Ctlr) of the user plane.

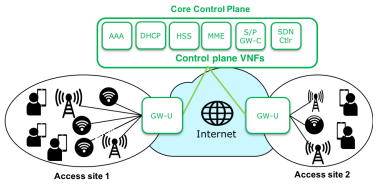

Fig. 4. UGW network service

For the UGW rolling up, b<>com provides a Heat template in order to deploy the control plane on an OpenStack platform.

A. Heat Template of the UGW

This template defines 5 private networks (e.g., secure, lte control) and 1 public network. It also defines 7 Virtual Machines (VM) on which specific services are installed. There is one VM for each VNF and another one called Internal switch for routing the network traffic by applying specific rules. Each VM is connected to a set of private networks through ports. An extract of this template is the example given in Listing 1.

B. Verification of the UGW Template

We give the UGW template, and its environment file, to our tool for its verification. Then, the *Alloy Analyzer* is able to find a UGW instance. This means that there is no error in the template. We expected this result because the template has been used to deploy, on several sites, an operational UGW.

Therefore, to better illustrate our approach, we introduce an error in the UGW template (cf. Listing 1) by changing the port of the *aaa_vnf*, on the *secure* network, to *hss_vnf_secure_port*. After this, *hss_vnf_secure_port* is attached to both machines.

Then, we deploy the new template. Heat does not detect the error and starts the deployment which takes 5 minutes to end. When checking the deployment state, we notice that it is partially done (cf. Fig. 5). Indeed, *hss_vnf_secure_port* cannot be attached to both machines. For a larger system, the deployment may take a longer time without being successful.

```
2019-08-16 16:05:00Z [ugw.hss_vnf]: CREATE_FAILED ResourceInError: resources.hss_vnf: Went
 to status ERROR due to "Message: Build of instance 01a6a927-076d-40e5-bb9f-160b70e66048 ab
orted: Failed to allocate the network(s), not rescheduling., Code: 500"
2019-08-16 16:05:00Z [ugw.hss_vnf]: DELETE_IN_PROGRESS  state changed
2019-08-16 16:05:02Z [ugw.hss_vnf]: DELETE_COMPLETE  state changed
2019-08-16 16:05:05Z [ugw.hss_vnf]: CREATE_IN_PROGRESS  state changed
stack@ami:~/adja$
```

Fig. 5. Deployment of the modified UGW template

Finally, we verify the modified template with our tool to see its behavior. In this case, as shown in Fig. 6, the error is detected, in 876 ms, before starting the deployment. Indeed, no instance of UGW is found and the tool highlights the fact that it cannot satisfy. This fact is the example given in Listing 3.

```
Executing "Run Show_Ugw_template for 0 but 4 int, 0 seq, exactly 1 Ugw_template
   Solver=minisatprover(jni) Bitwidth=4 MaxSeq=0 SkolemDepth=1 Symmetry=20
   307838 vars. 70828 primary vars. 3006854 clauses. 4277ms.
   No instance found Predicate may be inconsistent, contrary to expectation. 745ms.
   Core reduced from 24 to 12 top-level formulas. 876ms.

fact no_distinct_servers_with_the_same_port {
  all disj s1, s2: OS_Nova_Server|#(s1.property_networks_port.~provided &
                                    s2.property_networks_port.~provided) = 0 }
```

Fig. 6. Formal verification the modified UGW template

VII. Conclusion

This paper has proposed a formal approach to improve the reliability of HOT templates deployment through the Heat engine. This is done by (1) reviewing the HOT specification and identifying a set of errors it contains (missing invariants to prevent inconsistent behaviors), (2) formalizing the HOT specification to correct these errors, and (3) implementing a tool that verifies templates consistency, based on the formalized HOT specification, prior to launching their deployment.

A first perspective of this work is to consider more types of resources. Indeed, in the current status, only 36 types are considered. Another perspective is to ensure the evolution of this approach, as well as the implemented verification tool, in order to support the future versions of HOT. Finally, a major perspective is to generalize this approach to other orchestration languages such as Cloudify [24], and Docker Compose [25].

REFERENCES

[1] A. Manzalini, C. Lin, J. Huang, C. Buyukkoc, M. Bursell *et al.*, "Towards 5G Software-Defined Ecosystems: Technical Challenges, Business Sustainability and Policy Issues," *IEEE SDN White Paper*, 2016.

[2] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, "Network Function Virtualization: Challenges and Opportunities for Innovations," *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

[3] R. Mijumbi, J. Serrat, J. L. Gorricho, N. Bouten, F. De Turck, and R. Boutaba, "Network Function Virtualization: State-of-the-art and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 1, pp. 236–262, 2015.

[4] "Microsoft Azure," accessed: 2019-08-14. [Online]. Available: https://azure.microsoft.com/en-us/

[5] "IBM Cloud," accessed: 2019-09-16. [Online]. Available: https://www.ibm.com/cloud

[6] "OpenNebula," accessed: 2019-08-14. [Online]. Available: https://opennebula.org/

[7] "OpenStack," accessed: 2019-08-14. [Online]. Available: https://www.openstack.org/

[8] F. Callegati, W. Cerroni, C. Contoli, and G. Santandrea, "Performance of network virtualization in cloud computing infrastructures: The openstack case," in *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*. IEEE, 2014, pp. 132–137.

[9] F. Foresta, W. Cerroni, L. Foschini, G. Davoli, C. Contoli, A. Corradi, and F. Callegati, "Improving openstack networking: Advantages and performance of native sdn integration," in *2018 IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[10] "Heat: Openstack orchestration engine," accessed: 2019-08-14. [Online]. Available: https://wiki.openstack.org/wiki/Heat

[11] "HOT Core Specification," accessed: 2019-08-14. [Online]. Available: https://docs.openstack.org/heat/stein/template_guide/hot_spec.html

[12] "HOT Types Specification," accessed: 2019-08-14. [Online]. Available: https://docs.openstack.org/heat/stein/template_guide/openstack.html

[13] A. Souri, N. J. Navimipour, and A. M. Rahmani, "Formal Verification Approaches and Standards in the Cloud Computing: A Comprehensive and Systematic Review," *Computer Standards & Interfaces*, vol. 58, pp. 1–22, 2018.

[14] G. Marchetto, R. Sisto, J. Yusupov, and A. Ksentini, "Virtual Network Embedding with Formal Reachability Assurance," in *2018 14th International Conference on Network and Service Management (CNSM)*. IEEE, 2018, pp. 368–372.

[15] "Heat validation tools: information about the dry-run option," accessed: 2019-08-14. [Online]. Available: https://docs.openstack.org/newton/user-guide/cli-create-and-manage-stacks.html

[16] J. B. Stefani, "Components as Location Graphs," in *International Conference on Formal Aspects of Component Software*. Springer, 2014, pp. 3–23.

[17] J. Stefani and M. Vassor, "Encapsulation and Sharing in Dynamic Software Architectures: The Hypercell Framework," in *39th IFIP Int. Conf. Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*, ser. Lecture Notes in Computer Science, vol. 11535. Springer, 2019.

[18] M. Hennessy, J. Rathke, and N. Yoshida, "SafeDPi: A Language for Controlling Mobile Code," in *International Conference on Foundations of Software Science and Computation Structures*. Springer, 2004, pp. 241–256.

[19] A. Schmitt and J. B. Stefani, "The Kell Calculus: A Family of Higher-Order Distributed Process Calculi," in *International Workshop on Global Computing*. Springer, 2004, pp. 146–178.

[20] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron, "A Classification Framework for Software Component Models," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 593–615, 2010.

[21] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2012.

[22] ——, "Automating first-order relational logic," in *ACM SIGSOFT Software Engineering Notes*, vol. 25, no. 6. ACM, 2000, pp. 130–139.

[23] "b<>com, Wireless Edge Factory," https://b-com.com/en/bcom-wireless-edge-factory, 2019.

[24] "Cloudify," accessed: 2019-08-23. [Online]. Available: https://cloudify.co/

[25] "Docker Compose," accessed: 2019-08-16. [Online]. Available: https://docs.docker.com/compose/