Using Program Dependence Graphs to Detect Misunderstandings of Ansible's Variable Precedence and Expression Evaluation Semantics

Ruben Opdebeeck, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{Ruben.Denzel.Opdebeeck,Coen.De.Roover}@vub.be

I. Presentation Abstract

Our daily lives increasingly depend on large-scale digital computing infrastructures. Many of these infrastructures are managed using Infrastructure as Code (IaC), which enables rapid and automated deployments using domain-specific languages. The reliability of such digital infrastructures and their deployment, and the correctness of their configuration, is of the upmost importance. One would therefore expect the configuration management languages involved in cloud infrastructure deployments to render it difficult to write incorrect configurations. Nonetheless, Ansible, one of the most popular IaC languages [1], employs semantics that can easily lead to defects which can be difficult to debug [2]. This is caused by the combination of two peculiar properties:

1) Variables are names for expressions rather than values, and these expressions are re-evaluated every time a variable is dereferenced;
2) Variable precedence is convoluted and allows for global variables to take precedence over local variables[1].

These properties can lead to situations where unexpected values are used in the configuration of infrastructure machines. Moreover, tracing the usage of a variable to its concrete definition can be difficult, since the result of an expression can be impacted by any previously executed configuration step.

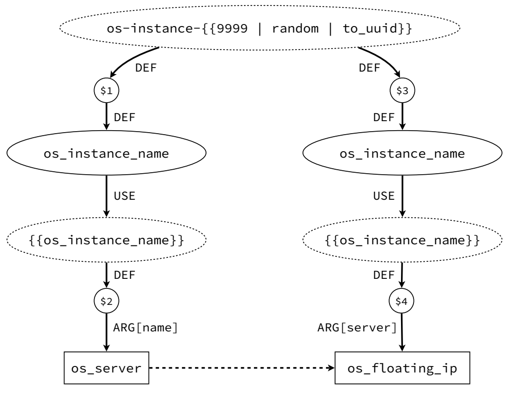
In this presentation, we share the preliminary results of an approach to automatically detect potential misunderstandings of these unconventional semantics. We describe program dependence graphs which fully capture the intricacies of Ansible's variable and expression semantics, and which facilitate reasoning about the variables, expressions, and data values used in Ansible code. We also introduce a catalogue of code smells related to variable precedence and expression semantics, show how the graphs can be used to detect these smells, and report on the preliminary results of running this smell detector on nearly 25.000 open source Ansible projects in the Andromeda dataset [3].

A. Illustrative Example

Figure 1 depicts an example of a defect which may be caused by a misunderstanding of Ansible's semantics. This example was automatically detected by a defect detector implemented on top of our program dependence graphs. In Figure 1a, we depict a simplification of the code of the open-source `RedHatGov.openstack_instance` role. This role provides reusable functionality to set up an OpenStack instance. The code describes two tasks, *i.e.*, steps that Ansible executes to configure a machine. The first task (lines 7–9) launches an OpenStack instance. Here, the name of the instance is the result of an expression (between double braces) which dereferences the `os_instance_name` variable. The second task (lines 11–13) is intended to assign a floating IP to the instance that was launched by the preceding task, using the same variable. However, by default, the variable is

```
1  ---
2  # Simplified content of "defaults/main.yml"
3  os_instance_name: "os-instance-{{ 9999 | random | to_uuid }}"
4
5  ---
6  # Simplified content of "tasks/main.yml"
7  - name: Launch instance {{ os_instance_name }}
8    os_server:
9      name: "{{ os_instance_name }}"
10
11 - name: Associate floating IP
12   os_floating_ip:
13     server: "{{ os_instance_name }}"
```

(a) Simplified code of the `RedHatGov.openstack_instance` role.



(b) Control-data flow graph of the above example.

Fig. 1: Simplified example of potential defect in the `RedHatGov.openstack_instance` role.

[1]https://docs.ansible.com/ansible/latest/user_guide/playbooks_variables.html#understanding-variable-precedence

initialised with an expression that is not idempotent (line 1). This expression uses a random number to generate the instance name, and because variables are names for expressions rather than values, the expression will be re-evaluated in the second task. Consequently, the second task will assign a floating IP to an instance with a different name.

Figure 1b depicts our graph representation of this simplified example. Here, rectangles depict tasks, dashed ellipses represent expressions, solid ellipses depict variables, and circles represent values. Solid edges denote data flow relations between nodes, whereas dashed edges depict control flow relations. Notice that the `os_instance_name` variable, although defined only once in the code, occurs twice in the graph because the value that will be produced by dereferencing the variable depends on an expression which is not idempotent. This graph therefore succinctly states that the two tasks will receive different values even though they use the same variable, because the variable evaluates to a different value produced by a non-idempotent expression. We can thus use graph queries to detect issues related to misunderstandings of Ansible's semantics, such as this reuse of a non-idempotent expression.

References

[1] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, "Adoption, support, and challenges of infrastructure-as-code: Insights from industry," in *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME19), Industrial Track*, 2019.

[2] R. Opdebeeck and C. De Roover, "The pitfalls of ansible's variable and template expression semantics," 2021, unpublished abstract in 1st Workshop on Configuration Languages (CONFLANG21).

[3] R. Opdebeeck, A. Zerouali, and C. De Roover, "Andromeda: A dataset of Ansible Galaxy roles and their evolution," in *Proceedings of the 2021 International Conference on Mining Software Repositories (MSR21)*, 2021, pp. 580–584.