Scopeannon: A Static Analyzer of Ansible's Scope Ambiguities

Olivia Proust
olivia.proust@imt-atlantique.fr
IMT Atlantique, INRIA, LS2N, UMR CNRS 6004
Nantes, France

Jolan Philippe
jolan.philippe1@univ-orleans.fr
Université d'Orléans, INSA CVL, LIFO UR 4022
Orléans, France

Hélène Coullon
helene.coullon@imt-atlantique.fr
IMT Atlantique, INRIA, LS2N, UMR CNRS 6004
Nantes, France

Frédéric Loulergue
frederic.loulergue@univ-orleans.fr
Université d'Orléans, INSA CVL, LIFO UR 4022
Orléans, France

Abstract

Nowadays, Infrastructure-as-Code (IaC) is widely used to manage large distributed software systems through scripts. However, script errors could lead to cascading failures rendering systems and applications unavailable. For this reason, the reliability and code quality of IaC systems have been extensively studied, highlighting different types of smells across tools. Ansible, a prominent configuration management tool in the IaC ecosystem, is known for its unexpected semantics and its unintuitive variable scope management, which can result in misconfigurations. We introduce Scopeannon, a work in progress static analyzer for Ansible that aims to help practitioners better understand and handle this peculiar scope management.

Keywords

Ansible, Infrastructure-as-Code, Static analysis, Variable scoping

ACM Reference Format:
Olivia Proust, Jolan Philippe, Hélène Coullon, and Frédéric Loulergue. 2026. Scopeannon: A Static Analyzer of Ansible's Scope Ambiguities. In . ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3844136.3845868

1 Introduction

In recent decades, Infrastructure-as-Code (IaC) has been widely adopted to manage large distributed systems. This success stems from simpler and more scalable automation of system configuration, deployment, and orchestration. through their languages, IaC tools bring many good practices from software development such as composition, reusability, or parametrization. Moreover, coding infrastructures instead of manually acting on machines and resources enables testing, verification, versioning, etc. Nowadays, IaC scripts are used to adapt infrastructures in case of faults, performance degradation, security issues, or updates. Managing these infrastructures requires robust and reliable scripts to avoid cascading failures that could render systems or applications unavailable. However, even if the use of IaC helps to prevent potential errors that could be made with the historic command line, it also brings its own complexity and risks. Although they manage critical infrastructure, IaC tools are based on non-formally specified configuration languages. This has led to unexpected semantics and behaviors and, consequently, to bad practices [1, 5]. Those bad practices can either be common to all IaC tools [6–8] through shared patterns and concepts, or specific to a tool's semantics [5]. Recently, the literature has highlighted those peculiar behaviors and ambiguities of IaC through code quality studies [5, 8–10].

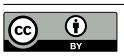
Ansible, an open-source and Python-based configuration management tool, is mainly adopted and appreciated for its simplicity. Ansible follows an agentless, push-based model, requiring only a control machine able to interpret YAML playbooks to manage a whole infrastructure. However, Ansible is also known for its unintuitive semantics, scope management, and code smells [2, 5]. Unlike most programming languages, when a variable has multiple definitions, Ansible does not always prioritize the most local one. Ansible's variable scoping can lead to unexpected behavior, particularly when variables remain accessible outside their local structure, when multiple scopes have the same precedence, or when globally defined expressions are evaluated differently across tasks due to precedence rules.

This paper makes two contributions. First, it characterizes how Ansible's variable resolution combines precedence, dynamic availability, role ordering, and lazy evaluation in ways that violate ordinary lexical-scoping intuitions. Second, it introduces the work in progress *Scopeannon*, a static analyzer that makes this resolution explicit for a supported subset of playbooks, helping practitioners identify unexpected scope selection and unintended variable overrides.

The rest of the paper is organized as follows. Section 2 reviews related work on Infrastructure-as-Code analysis and code quality. Section 3 introduces Ansible's language and details its scope management. Section 4 presents Scopeannon and its analysis methodology. Finally, Section 5 discusses future work.

2 Related Work

The analysis of typical behavior of IaC tools requires a deeper understanding of their semantics. Few studies focus on this aspect by proposing formalizations of IaC tools. For instance, Fu et al. [3] provide an operational semantics for a subset of Puppet (μPuppet), which was subsequently implemented, revise and extended in Why3 by Loulergue et al. [4]. Similarly, Anderson et al. [1] work on the core of SmartFrog, an open-source configuration tool. They demonstrate how formal methods can improve the understanding of IaC tools by highlighting defects or ambiguities. Such foundations may



Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/3844136.3845868

be the basis upon which to build semantic analysis in the future that could widely improve the reliability of IaC.

However, nowadays, most of the literature focuses on the detection of smells in IaC code through static analysis tools, similar to our proposal. For example, Rahman et al. [7] propose SLAC, a polyglot rule-based analysis of security smells in Ansible and Chef. Similarly, Saavedra et al. [8] proposed GLITCH for Puppet, Ansible, and Chef programs. More recently, De Vito et al. [11] also produced a polyglot tool with an LLM-based approach, based on the same smells detected by the two previously presented tools. Those studies show that some smells can be detected using several technologies. However, they do not consider variable management smells.

Dedicated static analysis tools have also been produced. Sobhani et al. [10] tackle the problem of reproducibility smells in Ansible, i.e., the possible reasons for the inconsistency of a script execution across different infrastructures. They identified five main smells and introduced *REDUSE*, a static analysis tool with a rule-based approach. Opdebeeck et al. [5] highlight Ansible's lazy evaluation and give a first look at its unusual variable scoping. From these observations, they identified three main smells that could be a threat to a good configuration and demonstrated that some smells require a deeper understanding of the studied tool. These smells mainly arise from the risk of producing a new value each time an expression is evaluated. This can be caused by excessively high precedence or impure expressions (e.g. random). To highlight the dynamics of the evaluation of an Ansible variable, they propose a Program Dependence Graph (PDG) based analysis, representing both the control flow and the data flow of a role. However, this study does not cover some aspects of Ansible's scope management, such as variables accessible outside their local structure, priority between two scopes with the same precedence, or globally defined expressions propagated to each instruction and evaluated locally.

Listing 1: Example of a playbook that performs the installation of PostgreSQL and Docker

```
1  - name : Install dependencies on all
2    hosts: all
3    vars:
4      version: 2.3
5      database: "EXAMPLE_DB"
6      dependencies :
7          - python3-pip
8          - virtualenv
9    roles:
10     - name: postgre-installer
11       database_name: "{{database + version}}"
12       vars:
13         user: "EXAMPLE_USER"
14   tasks:
15     - name: Install dependencies
16       ansible.builtin.apt:
17         pkg: "{{item}}"
18         state: present
19       loop: "{{dependencies}}"  #dependecies: PlayScope
20     - block:
21       - name: Check Docker requirement
22         ansible.builtin.stat :
23           path: /code/bin/setup-docker
24         register: script
25       - name: Installation with the script
26         ansible.builtin.command:
27             cmd: /code/bin/setup-docker -t
28             creates: /usr/bin/docker
29         when: script.stat.exists  #script: FactScope
30       when: version < 3  #version: PlayScope
```

Listing 2: Excerpt of the typical directory structure of a role

```
.
├ defaults
│   └ main.yml
├ tasks
│   └ main.yml
└ vars
    └ main.yml
```

3 Ansible

Ansible is an infrastructure automation tool for system deployment and configuration. Its YAML-based *playbooks* specify the desired state of managed hosts as sequences of *plays* composed of *tasks*. Playbooks are executed from a control node, which connects via SSH to hosts declared in an *inventory*. Hosts in the inventory can be grouped to apply common configurations to sets of machines sharing the same goal (e.g., database).

An example of a playbook for installing PostgreSQL and Docker is given in Listing 1. This playbook is used as an example for the rest of the section. Each *play* of a playbook is processed iteratively. A play can define its variables with values and expressions (e.g., Line 3), whose scope is denoted as *PlayScope*. In addition, a play may invoke a set of *roles* (Line 9). A role can be regarded as a parameterized function with a set of *tasks*. The role's implementation is organized across several files stored in a dedicated directory. Listing 2 gives an excerpt of the typical directory structure of a role. The `main.yml` file in the `tasks` folder is the set of tasks executed during the role call, while files in the `defaults` and `vars` directories contain predefined variables of the role with different scopes, respectively *RoleDefaultScope* and *RoleVarsScope*. This separation promotes code reuse and facilitates the sharing of roles through public platforms such as Ansible Galaxy.[1] A play invoking roles can override the values of these variables in two ways. For example, Line 11 in Listing 1 declares a new value for the `database_name` variable. In this case, Ansible treats this as a third type of role variable: a *role params* variable, within the scope *RoleParamsScope*. It is also possible to declare variables and values with the keyword `vars` as presented on Line 12. Those variables are considered *role vars* in the scope *RoleVarsRedefScope*.

Tasks are the main and smallest executable components of a playbook. They can be encapsulated within a role or defined directly in a play. Variables defined in tasks are contained in the *TaskScope*. Tasks invoke *modules*, which specify the actions to perform. Modules provide different levels of abstraction. Some encapsulate high-level operations, such as managing a database, thereby hiding the underlying commands. Others expose low-level operations, such as executing a shell command, as exemplified by Lines 25-29 in Listing 1. Moreover, their execution can be affected by attributes such as conditions and loops (Lines 15-19). During the execution of a loop, if no specific name is given for the loop variable, Ansible defines a new variable `item`, in a scope referred to as *LoopScope* in our analysis. Tasks can be grouped into *blocks* using the keyword `block`, as shown on Lines 20-30 of Listing 1. A block can also define

[1] https://galaxy.ansible.com/ui/

its own variable, in the scope *BlockScope*. The result of a task can be registered in a global scope referred to as *FactScope* using `register`. Finally, roles, blocks, and tasks can be guarded by a boolean condition using the attribute `when`. However, the condition expression is lazy-evaluated for each task it contains, thus making it highly sensitive to potential changes.

According to Ansible's documentation[2], variables can be defined in twenty-one locations, ordered by precedence. In this work, we model a subset of these definition contexts as scopes. This subset is composed of the previously presented scopes, defined in structures such as plays, roles, tasks, and blocks, and of the global variables dynamically defined during the execution of the playbook (e.g., registered vars). A natural intuition of scope management is that the more local a variable is, the higher its precedence, meaning that it overrides previous definitions within the corresponding local scope. Furthermore, such local definitions should remain restricted to that scope and should not be visible outside their associated structure, meaning that they could be temporary overrides. However, in Ansible, those intuitions are not always true. Figure 1 depicts the gap between the natural intuition of scope management and Ansible's actual scope precedence.

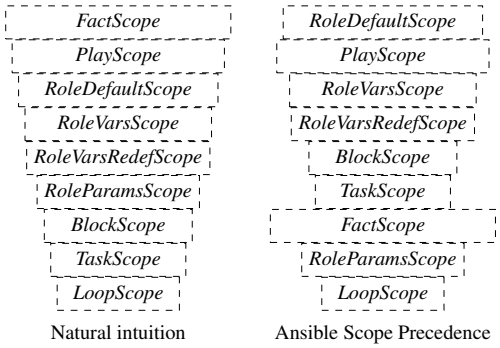


Figure 1: On the left, the expected natural order; on the right, Ansible's actual scope precedence.

The left-hand side of Figure 1 presents the natural intuition of precedence order described earlier. The right-hand side of the figure represents Ansible's actual precedence model. The lookup of a variable goes through those scopes from bottom to top according to the evaluation context. As represented in this figure, *FactScope*, which is accessible during the whole playbook evaluation and is therefore the most global scope, can easily override most local scopes. However, this specific priority order is not the only rule that could be confusing for practitioners. Variables defined in *TaskScope*, *BlockScope*, and *PlayScope* are only accessible during the execution of their respective structures. *RoleVarsScope* and *RoleDefaultScope* contain variables that are not exclusive to their respective roles but are accessible throughout the play in which the roles are invoked. Thus, they override same-named variables defined in *PlayScope* because of their higher priority. This case is demonstrated by Figure 2, in which the variable `x` of the play is never looked up. Furthermore, invoking multiple roles can heavily complicate variable evaluation. Figure 2 illustrates the lookup of variables according to the moment of their evaluation. The representation shows an example of a play that successively calls two roles, first `role A`, then `role B`. The execution workflow of Ansible (i.e., action order) is depicted by numbers.

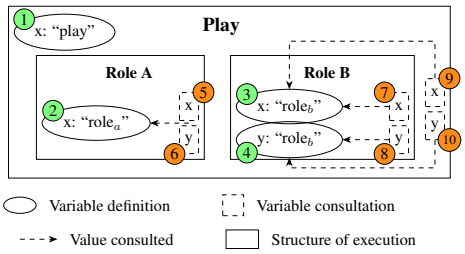


Figure 2: Representation of variable consultation according to the evaluation context. The example is a play that calls two roles, with x and y consulted at different points. Numbers indicate the order of variable definitions and consultations.

The first step, represented by green numbers in Figure 2, is the initialization of variables of the play and all its roles. If a variable is defined in two different roles, the priority depends on where the evaluation takes place. This second step of evaluation is depicted by orange numbers in Figure 2. If the evaluation occurs within a task in the play, the definition from the last role listed by the play takes precedence. This case is represented by the numbers 9 and 10 in Figure 2. However, if the evaluation occurs within a role, priority is given to its local definition. For example, in Figure 2, lookup 5 shows that `role A` adopts its own definition of `x`, as does `role B` in lookup 7. Despite this redefined variable, the role also has access to variables from other *roles*. For example, consultation 6 adopts the value defined by `role B` for `y` because `role A` provides no definition for it. Moreover, the two other types of role variables could also complicate expression evaluation. In addition to having higher priority than *RoleVarsScope* and *RoleDefaultScope*, *RoleParamsScope* and *RoleVarsRedefScope* are not accessible throughout the play and are only active during the execution of their role.

4 Scopeannon

Scopeannon is an OCaml-based static analyzer that reports the scopes consulted when evaluating Ansible playbook expressions. It produces an *augmented playbook* where each analyzed expression is replaced by a *ScopeAnalysis*, containing the original expression and a mapping of variable names to the scopes consulted during lookup. Developers can use this information to detect unexpected scope selection or unintended variable overrides. To simplify traversal, the "name" attribute of a role call (e.g., Line 10 in Listing 1) is replaced by the corresponding role abstraction. Scopeannon abstracts expressions as constants, lists of variable names, or lists of subexpressions.

Algorithm 1 depicts Scopeannon's analysis. It traverses the playbook in Ansible's execution order and progressively records variable definitions and scopes in a *store*. The store contains two lists: definitions in *FactScope*, which remain active throughout the playbook, and the currently active scopes with their definitions.

[2]Ansible Community Documentation for Ansible 14.2.0, accessed August 6, 2026.

Algorithm 1 Scopeannon

Input: playbook : (Expression Play) list
Output: (Scopeannon Play) list
1: store ← EmptyStore
2: anlPlaybook ← ∅
3: **for all** play ∈ playbook.plays **do**
4: *{Registration of play and role variables in order}*
5: playstore ← $RegisterVars$ PlayScope play.vars store
6: playstore ← $RegistrationAllRolesVarsDefault$ play.roles playstore
7: *{Analyze roles & tasks. Returned store may contain new global vars.}*
8: anlRoles, playstore ← $ScopingRoles$ play.roles playstore
9: anlTasks, playstore ← $ScopingTasks$ play.tasks playstore
10: *{Registration of the analysis in an augmented play}*
11: anlPlay ← $UpdatePlay$ play anlRoles anlTasks
12: anlPlaybook ← anlPlaybook + anlPlay
13: *{Update of the store with potential new global vars for the next play}*
14: store ← $GetOutStore$ store playstore
15: **end for**
16: **return** anlPlaybook

The order of the active scopes is significant because it resolves conflicts between scopes with equal precedence: the most recently registered scope has priority. Moreover, when Scopeannon enters a structure, it adds the scopes initialized by that structure to the store. When the analysis of the structure is over, it restores the previous store while retaining any newly introduced *FactScope* definitions. This action is performed by `GetOutStore`, as shown on Line 14 and at the end of each structure analysis. Scopeannon processes each play iteratively. It first registers play variables, followed by the default and variable definitions of called roles (Line 5), then analyzes roles in calling order (Line 8). For each role, it registers *RoleParamsScope* and *RoleVarsRedefScope*, then re-registers *RoleVarsScope* and *RoleDefaultScope* so the role's own definitions take priority during execution. When a role has a "when" expression, Scopeannon propagates it to its tasks and merges it with existing conditions when needed. Blocks are handled similarly.

Role tasks are then analyzed like tasks declared directly in the play (Line 9). The `ScopingTasks` function processes tasks iteratively, determining whether each uses a module or contains a block, and initializing the applicable *BlockScope*, *TaskScope*, and *LoopScope*. Blocks are analyzed recursively. For each task, Scopeannon considers expressions in the `loop` and `when` attributes. Using Ansible's precedence rules and the ordered scopes in the store, it maps variable names to the scopes that may be consulted during evaluation. The expression is then replaced by the resulting ScopeAnalysis. If the task registers its result, the corresponding definition is added only after the task has been analyzed, reflecting that the variable becomes available after the task executes. Both `ScopingRoles` and `ScopingTasks` return the list of analyzed structures and the updated store with dynamically registered global variables. Once converted to YAML, Scopeannon's augmented playbook exposes relevant lookup information through comments, as illustrated in Listing 1, allowing developers to inspect the analysis.

5 Future Work

Due to precedence, variables accessible outside their structures, and lazy evaluation, Ansible's scope management is intricate and error-prone. Moreover, some scopes are not explicitly described in Ansible's documentation. Scopeannon provides a static analysis to help practitioners understand Ansible's variable management and identify potential misuse of definitions. This work in progress still requires evaluation to assess its accuracy. The tool will be validated on playbooks ranging from synthetic test cases to widely used ones. However, this first version of Scopeannon remains limited. It considers only a subset of Ansible's scopes and does not account for *modules*. Expressions in *module* arguments are not analyzed, and some *module* actions can strongly affect variable-definition scopes, potentially leading to incorrect analyses. Addressing these aspects could improve Scopeannon's accuracy.

Acknowledgments

This work was funded by the French National Research Agency (ANR) through the project For-CoaLa (ANR-24-CE25-3158)

References

[1] Paul Anderson and Herry Herry. 2016. A Formal Semantics for the SmartFrog Configuration Language. *Journal of Network and Systems Management* 2 (April 2016), 309–345. doi:10.1007/s10922-015-9351-y
[2] Carolina Carreira, Nuno Saavedra, Alexandra Mendes, and João F Ferreira. 2026. The Ultimate Configuration Management Tool? Lessons from a Mixed Methods Study of Ansible's Challenges. (2026). doi:10.48550/arXiv.2504.08678
[3] Weili Fu, Roly Perera, Paul Anderson, and James Cheney. 2017. muPuppet: A Declarative Subset of the Puppet Configuration Language. In *Object-Oriented Programming (ECOOP) (Leibniz International Proceedings in Informatics (LIPIcs))*. 12:1–12:27. doi:10.4230/LIPIcs.ECOOP.2017.12
[4] Frédéric Loulergue and Mohamed Haady Tiemtore. 2026. Semantics of a Subset of Puppet 4.8 Mechanized in Why3. In *International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA)* (Kos, Greece) *(LNCS)*. Springer. to appear.
[5] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly variables in ansible infrastructure code: detection, prevalence, and lifetime. In *International Conference on Mining Software Repositories (MSR '22)*. 61–72. doi:10.1145/3524842.3527964
[6] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 164–175. doi:10.1109/ICSE.2019.00033 ISSN: 1558-1225.
[7] Akond Rahman, Md Rayhanur Rahman, Chris Parnin, and Laurie Williams. 2021. Security Smells in Ansible and Chef Scripts: A Replication Study. *ACM Trans. Softw. Eng. Methodol.* 1 (Jan. 2021), 3:1–3:31. doi:10.1145/3408897
[8] Nuno Saavedra and João F. Ferreira. 2023. GLITCH: Automated Polyglot Security Smell Detection in Infrastructure as Code. In *International Conference on Automated Software Engineering (ASE '22)*. 1–12. doi:10.1145/3551349.3556945
[9] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does your configuration code smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16)*. 189–200. doi:10.1145/2901739.2901761
[10] Ghazal Sobhani, Israat Haque, and Tushar Sharma. 2025. It Works (only) on My Machine: A Study on Reproducibility Smells in Ansible Scripts. In *International Conference on Mining Software Repositories (MSR)*. 384–395. doi:10.1109/MSR66628.2025.00069
[11] Gabriele De Vito, Fabio Palomba, and Filomena Ferrucci. 2025. SecLLM: Enhancing Security Smell Detection in IaC With Large Language Models. *IEEE Access* (2025), 204480–204498. doi:10.1109/ACCESS.2025.3637505