Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime

Ruben Opdebeeck
Vrije Universiteit Brussel
Brussels, Belgium
ruben.denzel.opdebeeck@vub.be

Ahmed Zerouali
Vrije Universiteit Brussel
Brussels, Belgium
ahmed.zerouali@vub.be

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

ABSTRACT

Infrastructure as Code is the practice of automating the provisioning, configuration, and orchestration of network nodes using code in which variable values such as configuration parameters, node hostnames, *etc.* play a central role. Mistakes in these values are an important cause of infrastructure defects and corresponding outages. Ansible, a popular IaC language, nonetheless features semantics which can cause confusion about the value of variables.

In this paper, we identify six novel code smells related to Ansible's intricate variable precedence rules and lazy-evaluated template expressions. Their detection requires an accurate representation of control and data flow, for which we transpose the program dependence graph to Ansible. We use the resulting detector to empirically investigate the prevalence of these variable smells in 21,931 open-source Ansible roles, uncovering 31,334 unique smell instances across 4,260 roles. We observe an upward trend in the number of variable smells over time, that it may take a long time before they are fixed, and that code changes more often introduce new smells than fix existing ones. Our results are a call to arms for more in-depth quality checkers for IaC code, and highlight the importance of transcending syntax in IaC research.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Infrastructure as Code, Ansible, code smells, program dependence graphs, empirical study, software quality

ACM Reference Format:
Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2022. Smelly Variables in Ansible Infrastructure Code: Detection, Prevalence, and Lifetime. In *Proceedings of MSR '22: Proceedings of the 19th International Conference on Mining Software Repositories (MSR 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

MSR 2022, May 23–24, 2022, Pittsburgh, PA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Manually provisioning, configuring, and managing machines in computing infrastructure is laborious and error-prone. *Infrastructure as Code* (IaC) [21] tools enable automating the provisioning, configuration, and management of computing infrastructure. As infrastructure code is just source code, it can be versioned alongside production code in repositories.

The correctness of infrastructure code is critical to the functioning of an application. Various recent outages can be traced back to defects in the infrastructure code [9, 16]. Rahman et al. estimate mistakes in configuration data as the most common defect type [31]. Unfortunately, the lack of test and verification tools for infrastructure code hampers ensuring its correctness [10].

Ansible is one of the most popular IaC tools today [10, 39]. It employs variable and expression semantics that are unlike those of many other languages. For instance, the precedence of variable declarations in Ansible has already caused widespread confusion among practitioners [3, 6]. We hypothesise that some developers may not be fully aware of some of Ansible's intricacies and may unwittingly introduce maintainability issues and defects into their infrastructure code. For example, Ansible's expression evaluation semantics have already caused defects in popular Ansible projects [2].

In this paper, we propose an automated approach based on program dependence graphs to detecting potential misunderstandings of Ansible's variable precedence and expression evaluation semantics. Using a prototype implementation, we empirically investigate their prevalence and lifetime in open-source Ansible roles, which are reusable blueprints of infrastructure code. The results illustrate the negative impact of these semantic misunderstandings on the quality of infrastructure code. This paper makes the following contributions:

- We present a catalogue of 6 novel infrastructure code smells related to Ansible's variable precedence and lazily-evaluated template expressions.
- We highlight the peculiarities of these semantics while transposing the concept of program dependence graphs from application to infrastructure code.
- We use the resulting control and data flow representation to implement a rule-based detector for the 6 variable smells.
- We empirically investigate the prevalence and the lifetime of variable smells in 21,931 open-source Ansible roles and provide the studied dataset in a replication package.

The proposed smells can be used by Ansible practitioners to judge the quality of infrastructure code. Our empirical results show that certain smells are widespread, suggesting the need for tool support to detect and repair these flaws. They also point to likely

```
1  # Contents of vars/main.yml
2  docker_network_name: "network-{{ 999 | random }}"
3
4  # Contents of tasks/main.yml
5  - name: Ensure network is created
6    docker_network:
7      name: "{{ docker_network_name }}"
8
9  - name: Define container name
10   set_fact:
11     container_name: "redis-{{ 999 | random }}"
12
13 - name: Ensure redis container is started
14   docker_container:
15     name: "{{ container_name }}"
16     image: redis
17     state: started
18     networks:
19       - name: "{{ docker_network_name }}"
20   vars:
21     container_name: "unused!"  # Because of set_fact on line 11
```

Figure 1: Defective role managing a Docker container.

misunderstandings about Ansible's semantics, which may aid language designers in building safer IaC languages.

2 INTRODUCTION TO ANSIBLE

Ansible enables automating the provisioning, configuration, and management of computing infrastructure. Practitioners create a *playbook* consisting of a number of *plays*. A play targets a group of nodes, which are configured according to the play's *tasks*. A task executes exactly one *action* which performs the necessary changes on the node. Actions enable declaring what the machine's configuration should be without specifying how. Ansible ships with several predefined actions, whose responsibilities range from installing software through a package manager, over managing firewall rules, to provisioning cloud servers. Plays can also include reusable *roles*, which are collections of tasks intended for a specific purpose, such as installing and configuring an HTTP server. The Ansible Galaxy ecosystem[1] curates open-source roles, which can be considered the libraries of Ansible [27] and are the main subject of this study.

2.1 Motivating Example

Figure 1 depicts an example role comprising three tasks (lines 5–7, 9–11, and 13–21, respectively). This role ensures that a Docker container is started and associated with a certain Docker network (a virtual network of Docker containers). The first task (lines 5–7) will create this network if it does not exist, through the `docker_network` action. This action takes the desired name of the network as an argument. Subsequently, the second task (lines 9–11) will define a variable containing the name of a Docker container. Finally, the third task (lines 13–21) will ensure that the container is started, is given the previously-defined name, and is assigned to the newly-created network.

As illustrated, Ansible code is essentially a data structure written in YAML. A task is a key-value mapping, the keys of which determine its execution by Ansible. The example assigned each task a diagnostic name through the key `name`. In general, a key that does not correspond to a built-in is considered the task's action to execute. Inner keys can be used to shape the execution of an action. Among others, the `when` key can be used to skip the action's execution under given conditions, and the `loop` key can be used to execute the action for each item in a list.

The example shows the definition of a variable named `docker_network_name` on line 2. The variable is initialised using an expression that generates a random network name. It is then used twice, in the expressions on lines 7 and 19, as the name of the created network. Somewhat surprisingly, these two occurrences of the variable will have different values thus rendering the role defective. In fact, this is a simplified version of a real-world mistake found in the `RedHatGov.openstack_instance` role. To understand why, we need to delve into the intricate details of Ansible semantics.

2.2 Template Expressions

Ansible programs can feature Jinja2 *template expressions* demarcated by double braces. Each template expression within a string will be evaluated and will be substituted by its result into the string. Expressions can manipulate data through filters (denoted by the `|` character), tests, "lookup" calls, *etc.* Importantly, Ansible features a form of lazy evaluation for template expressions. An expression is not evaluated until its value is needed. Thus, a variable is bound to an expression rather than to its value. The expression is evaluated to a value when the variable is looked up and consumed. Note that the resulting evaluation value is not cached. Instead, the expression bound to a variable is re-evaluated for each of its occurrences.[2]

This reliance on referential transparency to evaluate variable references requires that initialisers are pure and do not affect or rely on any form of state. Nonetheless, it is easy to construct Ansible variable initialisers that lack purity by using expressions that use randomness, read data from files, perform network operations, or execute arbitrary shell commands. Consider the expression on line 2 of Figure 1 again. Ansible will apply the `random` filter to the number `999`, which will produce a random number between 0–999. However, this filter is impure, and a second evaluation will give a different result. The variable is first used on line 7, where the expression may produce a value such as "`network-529`", which will be used to create a Docker network. However, on the second use of the variable (line 19), the expression may produce a different value (*e.g.,* "`network-11`"). The role is thus defective, as the Docker container would be assigned to an entirely different network. It is a prime example of the kind of defects developers unfamiliar with Ansible's intricacies might unwittingly introduce, and serves to motivate our detection approach.

To prevent impure expressions from causing such issues, Ansible supports two means for binding a variable to a constant value rather than an expression. First, the built-in `set_fact` action (exemplified on lines 9–11) can be used to eagerly evaluate an expression and bind a variable to its value, thereby preventing impure expressions from being re-evaluated. Second, the `register` key can be used within tasks to bind a variable to a data structure that represents the task's result. Variables defined in either way will always resolve

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

[2] There is one exception where Ansible does apply caching under very strict circumstances. However, it is undocumented and should not be relied upon.

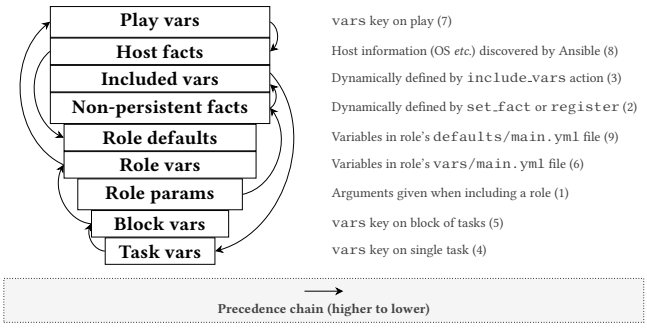

Figure 2: Variable scoping and precedence rules summarised.

to the same value, unless they are redefined. Therefore, in the example, since the `container_name` variable is defined by `set_fact`, multiple uses of this variable will still lead to the same value, even though randomness is used.

2.3 Variable Scoping and Precedence

In traditional languages, variables are scoped lexically and during lookup a definition from the closest encompassing scope takes precedence. Ansible variable precedence, in contrast, is governed by 22 precedence rules[3], the order of which does not follow the nesting of scopes. Figure 2 summarises Ansible's variable scoping and precedence rules. Nine scopes in which variables can be defined have been depicted. The top four are global to a play, and their definitions are visible in each task executed by the play, including role tasks. The scopes for role defaults and role variables are loaded automatically when a role is included, and the variables defined within these scopes are visible to all tasks in that role. Similarly, block variables are visible throughout all tasks in a block, and task variables are only visible to a single task. In Figure 1, the variable `docker_network_name` (line 2) is a role variable and is visible to all tasks in the role. The `container_name` variable defined on line 11 is a non-persistent fact, and is therefore visible to all tasks, including tasks in the play that includes the role, or even tasks in other roles. On the other hand, the `container_name` variable defined on line 21 is a task variable, and is only visible to that single task.

Naturally, variables defined in a block can reference variables from outer blocks in their initialisation expression. For example, a task variable can use a role variable. However, lazily-evaluated template expressions do not close over their lexical scope (*i.e.,* the scope in which they are bound to a variable). Variable references within a template expression are instead resolved according to the scope in which the expression is evaluated (*i.e.,* every scope in which the variable they are bound to occurs).

Figure 2 uses arrows to depict the precedence rules according to which variables are looked up. In contrast to other languages, variable precedence does not strictly follow the nesting of scopes. It is possible for variable definitions in certain outer scopes to take precedence over variable definitions in the local scope. Variables defined by the `set_fact` action (*i.e.,* non-persistent facts) or those included via the `include_vars` action are globally visible, yet take precedence over variable definitions in local blocks or tasks. In fact, the `container_name` variable defined on line 21 of Figure 1 can never be used, because the non-persistent fact of the same name defined on line 11 will take precedence. This is another example of a potential pitfall caused by Ansible's intricacies.

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

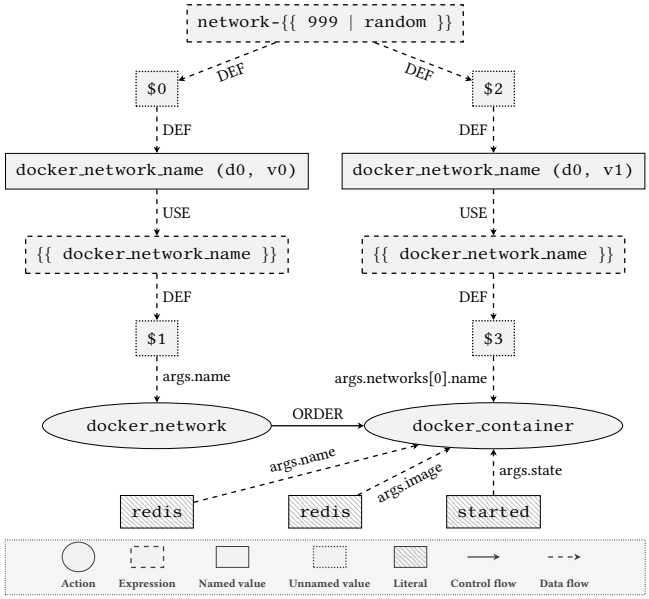

Figure 3: PDG for the defective Ansible role of Figure 1.

In summary, Ansible's semantics encumbers tracing variable usages to definitions. Initialising expressions for variables can refer to variables, yet do not close over their lexical scope. They are evaluated lazily for every occurrence of the variable they are bound to, and may even introduce higher-precedence definitions for existing variables. The entire play must therefore be considered, when analysing Ansible variable usages and definitions, rendering it challenging for humans and machines alike.

3 ANSIBLE PROGRAM DEPENDENCE GRAPHS

We introduce a Program Dependence Graph (PDG) for Ansible roles that accurately and succinctly represents their control and data flow. This representation accounts for the intricacies of the variable precedence and lazily-evaluated template expression semantics laid out in Section 2. Importantly, as variable initialisers are re-evaluated on demand, variable occurrences may have multiple values and thus multiple PDG nodes.

3.1 Program Dependence Graph Structure

The nodes of our PDGs can be categorised into control and data nodes. The former represent control flow structures, whereas the latter represent data. These nodes are interconnected using edges representing control flow and data flow. Figure 3 depicts the PDG for the Ansible code in Figure 1.

3.1.1 Control Nodes and Control Flow Edges. *Action* control nodes represent the action executed by each task. *Conditional* and *Loop*

control nodes (not depicted) represent branching and looping. *Order* control flow edges connect these control nodes, representing possible execution paths.

3.1.2 Data Nodes. Template expressions are represented as *expression* nodes. Furthermore, we distinguish between three types of abstract data values in the PDG. *Literal value* nodes correspond to literals in the role. *Unnamed value* nodes represent abstract values produced by an expression. These can either be used directly by a control node, or they may be bound to a variable in a definition. *Named value* nodes represent a unique abstract value produced whenever a variable is looked up and its initialiser expression is evaluated. These nodes carry the name of the variable, as well as a *lexical definition version* (d_i) and a *value version* (v_i) for that variable. Figure 3 shows two named value nodes for the variable `docker_network_name` with the same lexical definition version (d_0) but differing value versions (v_0 and v_1). This indicates that both values originate from the same lexical definition, but that this definition may produce different values, in this case because of the impure initialiser. The lexical definition version changes when a new definition for the variable is introduced. Value versions change when the lexical definition remains the same, but the unnamed value defining the named value has changed. This can happen either because the expression producing the unnamed value is not pure, or because that expression references a variable whose value has changed.

3.1.3 Data Flow Edges. Data nodes are connected through data flow edges whose labels indicate the type of data flow that occurs. *Def* edges represent definitions, such as from an expression to an unnamed value, from an unnamed value to a named value, or from a loop control node to the iteration variable. *Use* edges represent usages, such as from a named value to an expression, or from an unnamed value to a conditional or loop control node. *Arg* edges connect a value to an action node, whose label contains the name of the task argument that uses the value. *Defined-if* edges link a condition to a value, indicating that the value is only defined conditionally, and may not exist when the condition is not met.

In the example in Figure 3, we can see that the expression `network-{{ 999 | random }}` produces two separate unnamed values (`$0` and `$2`) because it is not pure. Each of these unnamed values then defines a named value `docker_network_name`, which differ in their value versions. Each of these named values is subsequently used in another expression. Note that these expressions are represented by separate nodes, as their input values differ. If these expressions used the same input values, our PDG would reuse the same expression node, regardless of the fact that both expressions lexically appear in different places in the program. This careful reuse of nodes enables distinguishing between the cases of impure expressions, variable redefinitions, *etc.*, only by looking at the structure of the PDG.

3.2 Computing Data Dependences

The PDG builder maintains a collection of scopes during its role analysis. This collection starts with an empty scope for each global precedence level (*e.g.,* non-persistent facts, host variables, *etc.*, cf. Section 2.3). Whenever the builder enters a lexical module (*e.g.,* a block or task), a new scope for this module is added, and subsequently removed when the builder exits the module. The *lexical nesting order* traverses the collection in the order in which these scopes were added, while the *precedence order* traverses from higher to lower precedence and is used during variable lookups. Data dependences are computed from three types of records:

- Variable definition (VDef) records represent variable definitions with their initialisers. These are stored within the aforementioned scope collection.
- Expression value (EVal) records uniquely represent abstract values produced by an expression, and contain the expression's data dependences and a version number to distinguish between values produced by impure expressions. These records map to an expression node and the unnamed value node it defines.
- Variable value (VVal) records uniquely represent abstract variable values and combine a VDef and an EVal record. These map to the named value nodes in the graph.

Since expressions are lazy, the PDG builder does not add value nodes to the graph at definition time. Instead, it defines a variable by inserting a new VDef record into the appropriate scope depending on the definition's precedence. The value nodes are only created once an expression is used in a task argument. Algorithm 1 describes how these expressions are resolved to values. Note that the algorithm does not evaluate an expression to its concrete run-time value, but statically analyses the records in the scope collection to compute a unique abstract data value representing the expression's possible run-time values.

At a high level, Algorithm 1 uses two mutually-recursive functions, Resolve-Expression and Resolve-Variable. The former function resolves all variable values referenced by an expression using the latter function (lines 5–6), and produces an EVal record.

Algorithm 1 Expression resolution

```
1:  function Resolve-Expression(e, CS)
2:      input: expression e, scopes collection CS
3:      output: EVal record r_e
4:      d ← ∅                                          ▷ Set of data dependences
5:      for n ∈ Get-Var-References(e) do               ▷ Resolve all variables
6:          d ← d ∪ Resolve-Variable(n, CS)
7:      if Is-Pure-Expr(e) then
8:          r_e ← Create-EVal-Record(e, d, 0)
9:      else                                           ▷ Distinguish with value version
10:         v ← Get-Next-Value-Version(e, d)
11:         r_e ← Create-EVal-Record(e, d, v)
12:     return r_e

13: function Resolve-Variable(n, CS)
14:     input: variable name n, scopes collection CS
15:     output: VVal record r_v
16:     r_d ← Find-VDef-Record(CS, n)
17:     if Initialiser-Is-Expression(r_d) then
18:         e ← Get-Expression(r_d)
19:         r_e ← Resolve-Expression(e, CS)
20:         r_v ← Create-VVal-Record(r_d, r_e)
21:     else                                           ▷ Constant initialiser, e.g. set_fact
22:         r_v ← Create-Constant-VVal-Record(r_d)
23:     return r_v
```

Conversely, the latter function produces a VVal record by looking up a VDef record in precedence order (line 16) and resolving its initialiser through Resolve-Expression. If the variable definition has no initialiser (*e.g.,* for `set_fact` or `registered` variables), it returns a constant VVal record instead (line 22).

The records returned by these functions uniquely identify abstract values used in the role. If two subsequent applications of Resolve-Expression produce identical EVal records, it is guaranteed that the concrete values during role execution will be identical. This enables the PDG builder to represent fine-grained data dependences, since each EVal and VVal record uniquely maps to an unnamed or named value node in the PDG respectively.

However, care must be taken with impure expressions, as their value may change arbitrarily. The algorithm uses the value version stored in the EVal record to distinguish the different values produced by an impure expression (lines 10–11). In such cases, the PDG builder will reuse only the expression node, since data dependences remained the same, but will create new unnamed value nodes to represent the changed values produced by the expression. This can be seen in Figure 3, where the top expression defines two separate unnamed value nodes. The change in the produced values caused the VVal records for the `docker_network_name` variable to be different, which in turn led to a change in data dependences for the expressions referencing this value, thereby leading to the creation of new nodes.

To determine expression purity, our algorithm consults a set of built-in filters, tests, and "lookup" call names which are known to be pure. We distilled this set from the Ansible documentation, its source code, and experience. For example, the `first` filter, which returns the first element of a sequence, is contained within this set. However, filters such as `random` or tests such as `exists` (which checks whether a given path exists on the file system) are not in this set, as they are not pure because of their reliance on internal or external state. We consider a template expression to be pure if each filter, test, and "lookup" call it uses is within this set, which necessarily under-approximates the pure expressions in an Ansible program. Therefore, this straightforward approach can never mark an expression as pure while it is not, but can suffer from false negatives when user-defined filters or tests are used within a template expression.

3.3 Building Program Dependence Graphs

Our PDG builder traverses an Ansible role module by module, and populates the graph with nodes and edges determined by the type of module encountered. For instance, the builder first enters the scope of the default and role variables and registers each of them. Subsequently, it traverses each of the role's tasks and connects the resulting action nodes via order edges.

For generic tasks, the builder enters a new task scope and registers the task-local variables. It then builds the looping and conditional control structures, if present. Afterwards, it inserts the action node for the task into the PDG, and computes the task's data dependences (cf. Section 3.2). The builder uses the EVal and VVal records produced by Resolve-Expression from Algorithm 1 to insert expression, named value, and unnamed value nodes into the PDG if necessary, and connects them to one another and to the action node using the appropriate data flow edges. Finally, if the task has a `register` key, a named value node is created for the task result and a constant variable is registered in the scope collection.

Some actions are handled as special cases. For instance, `set_fact` and `include_vars` dynamically define variables at high precedences (cf. Section 2). As the former eagerly evaluate their variable initialisers, the builder resolves their initialisers and registers the new variables with constant initialisers. The latter dynamically define variables from an external file, which is handled by attempting to determine the file that would be loaded, and registering its variables. However, the file name provided to `include_vars` may itself be an expression, in which case the builder does not attempt to resolve the expression and simply ignores this action. Both of these actions may also be conditionally executed. If this is the case, the builder adds the defined-if edges to all registered variables. The `import_tasks` and `include_tasks` actions, for which control flow jumps to external files, are treated similarly.

4 PDG-BASED VARIABLE SMELL DETECTION

We describe 6 novel code smells, structured into 3 categories, concerning the usage and declaration of variables in Ansible roles. Due to Ansible's unique variable precedence and template expression evaluation semantics, their presence may cause confusion among IaC practitioners and lead to unexpected consequences and potential bugs when the role is included in an Ansible play. For each smell, Table 1 presents a detection rule which identifies the smell's fingerprint in the program dependence graph.

The first variable smell category, *unsafe reuse*, concerns the reuse of a variable whose value may have changed in between two usages. This can be due to an impure initialiser of the variable declaration (*UR1*), or due to the data dependences of the initialiser having changed (*UR2*) which occurs when an upstream variable has been redefined. These smells are derived from potential pitfalls caused by Ansible's unique expression evaluation semantics. When present, these smells may be indicative of a bug in case developers expected different occurrences of the same variable to evaluate to the same value, as is the case in the example of Figure 1. The detection rules for the smells consider each pair of named value nodes that originate from the same lexical definition (v_d) but have different value versions (v_v), and compare the data dependences of the initialiser expressions for the variables that define the named values. For example, the rule for *UR1* matches the example depicted in Figure 3.

The *unintentional override* category groups smells related to Ansible's variable precedence intricacies. We discern two smells in this category, namely unconditional and unused overrides. The former occurs when a new definition overrides a previous definition at a higher precedence without taking the previous definition into account. It is inspired by "suspicious variable shadowing" smells for general-purpose languages. Its detection rule considers an override to be unconditional if the previous definition is not used in the new initialiser, and when there are no common data dependences in the conditions under which the two definitions exist. Line 9 of Figure 4 exemplifies this smell, where the inclusion of a variable will unconditionally override an existing definition implicitly loaded as a role variable, defined on line 2. The latter code smell manifests itself as a variable definition while a previous definition

Table 1: Summary of proposed code smells. In the presented rules, we assume n, n_1, and n_2 are named value nodes, and that n_1 and n_2 share the same name. d_i and v_i refer to the lexical definition version and value version of n_i, respectively.

Category	Code	Smell name	Detection rule
Unsafe reuse	UR1	Impure initialiser	$d_1 = d_2 \wedge v_1 < v_2 \wedge getDeps(getExpr(n_1)) = getDeps(getExpr(n_2))$
	UR2	Changed data dependence	$d_1 = d_2 \wedge v_1 < v_2 \wedge getDeps(getExpr(n_1)) \neq getDeps(getExpr(n_2))$
Unintentional override	UO1	Unconditional override	$d_1 < d_2 \wedge getPrec(n_1) \leq getPrec(n_2) \wedge n_1 \notin getDeps(getExpr(n_2)) \wedge getDeps(getCond(n_1)) \cap getDeps(getCond(n_2)) = \emptyset$
	UO2	Unused override	$d_1 < d_2 \wedge getPrec(n_1) > getPrec(n_2)$
Too high precedence	HP1	Unnecessary `set_fact`	$isSetFact(n) \wedge isPure(getExpr(n)) \wedge isPure(getCond(n))$
	HP2	Unnecessary `include_vars`	$isIncludeVars(n) \wedge getCond(n) = \emptyset$

```
1  # Contents of vars/main.yml
2  foo: 1
3
4  # Contents of vars/other.yml
5  foo: 2
6
7  # Contents of tasks/main.yml
8  - name: include variables
9    include_vars: vars/other.yml
10   # ^ HP2: Unconditional include_vars
11   # ^ UO1: Unconditionally overrides foo on line 2
12
13 - name: print foo
14   debug:
15     msg: "foo is {{ foo }}"
16   vars:
17     foo: 3
18     # ^ UO2: Shadowed by included variable on line 9
```

Figure 4: Role exhibiting *UO1*, *UO2*, and *HP2* smells.

already exists at a higher precedence. Such variable definitions have no effect and can never be used since the old definition will take precedence. This smell is derived from "unused variable" smells for general-purpose languages in combination with Ansible's complicated variable precedence system. Line 17 of Figure 4 depicts an example in which the task-local variable can never be used, since the variable included on line 9 will take precedence.

The final category, *too high precedence*, groups bad practices with regards to the scoping and precedence of variable definitions, taking inspiration from "too broad scope" smells for general-purpose languages. In Section 2.3, we identified `set_fact` and `include_vars` as task actions that dynamically define variables at a high precedence. Both of these actions have valid use cases. The former can be used to eagerly evaluate impure expressions, whereas the latter can be used to dynamically decide whether variables need to be defined. Nonetheless, we argue that these means of defining variables should be used sparingly in Ansible roles, as they will break the role's encapsulation and will be added to the global scope of any play into which the role is included eventually. Detection rule *HP1* therefore emits a warning for every usage of `set_fact` of which both the expression and all task conditions are strictly pure, whereas rule *HP2* emits a warning for every `include_vars` that will be executed unconditionally. Line 9 of Figure 4 is an example of the latter. The unconditionally included variable should instead be defined in a different scope (*e.g.,* as a role variable).

We have implemented the proposed rules in a prototype smell detection tool, which operates in two phases. First, it builds a PDG for each Ansible role given as input. Then, it traverses the PDG in search of nodes that match the presented detection rules.

5 EMPIRICAL ANALYSIS

We now present our empirical study into the real-world prevalence and lifetime of the 6 proposed variable smells.

5.1 Study Design

Our study analyses the open-source Ansible Galaxy role repositories contained in Opdebeeck et al.'s Andromeda dataset [26], and aims to answer the following research questions:

- RQ_1: How precise is our code smell detector?
- RQ_2: How prevalent are the proposed code smells in Ansible roles?
- RQ_3: Do the proposed code smells co-occur in Ansible roles?
- RQ_4: What is the lifetime of a code smell in an Ansible role?

After filtering out 231 repositories containing multiple roles and 3,438 forks from the Andromeda dataset, we are left with 21,931 repositories for analysis. For each commit, we ran our PDG builder on the corresponding snapshot, producing a total of 629,073 graphs. Table 2 depicts the distribution of variable precedences. 19,661 roles (89.4%) have defined variables at least once in their history, and 19,393 still feature variables in their latest commit. Role defaults are the most common, while block variables are rare.

Table 2: Variables extracted from all commits of 19,661 roles.

precedence	# roles	mean	std	median
block variables	89	4.2	5.1	2
include parameters	9,949	1.2	1.3	1
included variables	294	5.3	11.9	3
role defaults	17,229	13.5	26.2	6
role variables	4,780	5.2	8.3	3
non-persistent facts	8,638	4.1	7.9	2
task variables	1,157	2.7	4.3	1
All	19,661	15.8	29.5	8

We ran the variable smell detector on the produced PDGs. We consider smells occurring across different commits to a role the same if both are of the same type and are caused by the same lexical definition of a variable. This enables computing the number of unique variable smells, and when each smell was introduced and fixed. The resulting smell instances form the dataset of our study. This dataset, the extracted PDGs, and data analysis notebooks are

Table 3: Validation results

Category	Smell	# TP
Unsafe reuse	UR1	17
	UR2	20
Unintentional override	UO1	16
	UO2	17
Too high precedence	HP1	20
	HP2	20

provided in a replication package at https://doi.org/10.6084/m9.figshare.18819074.

5.2 Results

We now present the results of our empirical study.

RQ_1: How precise is our code smell detector? To validate our PDG builder and smell detector, we randomly sampled and manually validated 20 detected instances of each proposed variable smell (i.e., 120 in total). We consider a smell instance to be a false positive if the detection rule should not have matched, but do not take the role developer's intent into account to eliminate subjectivity from the validation. For example, we consider *unconditional override* (*UO1*) instances to be true positives even when it seems that the developer intentionally overrides the variable, since the detector is intended to produce objective warnings, not all of which may truly be bugs. Table 3 summarises the results.

The detector achieves good precision for most smell types. Moreover, all of the encountered false positives stem from limitations of the PDG builder. Specifically, the 3 false positives for the *unsafe reuse due to impure initialiser* (*UR1*) smell are caused by over-approximations of the PDG builder due to unrecognised filters in expressions, and can be remedied easily in future work. Similarly, the 4 false positives of *UO1* instances are caused by a builder limitation related to multi-level task conditionals. The 3 *UO2* false positives are caused by the builder not recognising certain dynamic task inclusion actions, which caused it to assign the wrong precedence to a small number of variable definitions. Finally, we find no false positives for the *unnecessary* `set_fact` (*HP1*) smell.

Findings: From the manual validation of 20 random instances per smell, the detector's precision is sufficient for the rest of the study.

RQ_2: How prevalent are the proposed code smells in Ansible roles? Overall, we found 31,334 unique smell instances, spread across 4,260 (19.4%) of the considered roles. Of these, 21,934 (70%) are still present in the latest version of the roles. Smells of the same type caused by the same variable definition across different commits are only counted once. Table 4 summarises the results. Unnecessary usages of `set_fact` (*HP1*) and unconditional overrides (*UO1*) are by far the most prominent smells. Conversely, both types of unsafe variable reuses occur rarely. This may suggest that *UR1* and *UR2* are more likely to result in defects, and are therefore fixed before they are committed to the repository. It may also be the case that most variables are only used once.

We also investigated the evolution of the smell prevalence over time, depicted in Figure 5. It is clear from the plot on the left that

Table 4: Smell instances during the lifetime of 4,260 roles. The distribution refers to the number present in each project.

category	name	# roles (%)	mean	median
Too high precedence	HP1	3,345 (78.5)	5.7	2.0
	HP2	184 (4.3)	6.4	3
Unintended override	UO1	2,124 (49.9)	5.1	2.0
	UO2	14 (0.3)	2.0	1.5
Unsafe reuse	UR1	37 (0.9)	1.8	1.0
	UR2	30 (0.7)	2.2	2.0

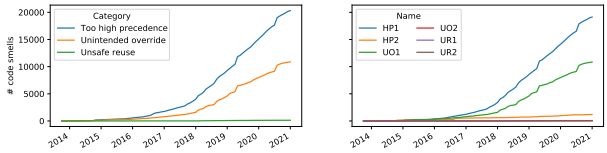

Figure 5: Cumulative number of smell instances over time.

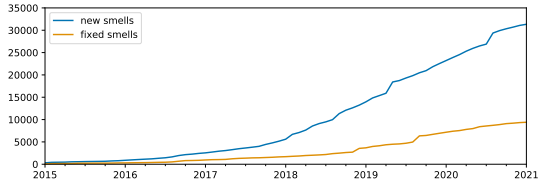

Figure 6: Cumulative number of new and fixed instances.

too high precedence and *unsafe override* smells continue to be introduced. The plot on the right shows that the *HP1* and *UO1* smells are the main reasons for the observed trends. Figure 6 compares the cumulative evolution in the number of added and number of fixed smells on a monthly basis. The rate at which new smells are introduced, outpaces that of their fixes.

Findings: 19.4% of the roles are affected by the proposed variable smells. Most smells concern definitions with too high a precedence, followed by unintended overrides. Unsafe reuses occur rarely. New smells are introduced more frequently than existing ones are fixed.

RQ_3: Do the proposed code smells co-occur in Ansible roles? Co-occurrences of code smells in the same role snapshot may indicate a causal link between the smells. As hypothesised in Section 4, variables defined with too high a precedence may lead to unintended overrides. Among the 109,719 role snapshots with smells, we found 43,255 (39.4%) in which at least two smells from different categories co-occurred, spread over 1,334 projects. Figure 7 shows the proportion of repositories of roles that had smells co-occurring within the same snapshot. Note that the totals do not sum to 100% since it is possible for a role to have a co-occurrence in one version, but not in another. For example, 60.3% of the roles have at least one version in which only *too high precedence* smells occurred, while 30.8% of roles have at least one version in which smells from both the *too high precedence* and *unsafe override* categories occurred. A role can thus be in both of these proportions. We observe that *unintended override* smells co-occur with *too high precedence* smells more often

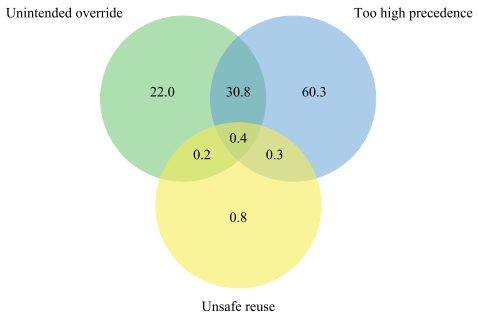

Figure 7: Roles with co-occurring smells within a snapshot.

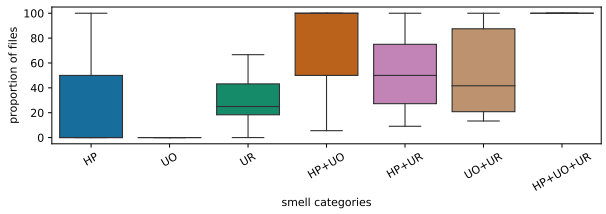

Figure 8: Distribution of files in which smells (co-)occurred.

than they occur alone within the same role snapshot. This finding confirms that variables defined with too high a precedence may lead to problems, including other smells. Nonetheless, as shown by the high proportion of roles in which smells occur alone, both can occur independently.

Finally, we investigated whether variable smells co-occur within the same files within an affected role snapshot. When a file exhibits the same co-occurrences in different role snapshots, it is only considered in the first snapshot to avoid overestimating the file-level co-occurrences for roles with many commits. Figure 8 shows the proportional distribution of files in which smells (co-)occur in a role snapshot. Smells from different categories frequently co-occur within the same files. This is less the case for *unsafe reuse* smells. The *unsafe reuse* smells occurred on their own in an average 28.3% of files per role, while *unintended override* and *too high precedence* occurred alone in 10.8% and 20% of files respectively.

Findings: Although only 39.4% of roles exhibit co-occurring code smells, we find that *unsafe reuse* smells often co-occur with *too high precedence* smells. We also find that code smells frequently co-occur within the same file.

RQ_4: What is the lifetime of a code smell in an Ansible role? To understand the lifespan of variable smells, we investigated when they get introduced in a role and how long it takes for them to be removed. Figure 9 shows the cumulative evolution of their first appearance in function of the time elapsed since the affected role's initial commit. We observe that *too high precedence* smells are the first to appear. It took 0.3, 0.96 and 4.05 months for 50% of *too high precedence*, *unintended override* and *unsafe reuse* smells to appear, respectively. The figure on the right shows that *HP2* smells reached 50% before *UO1* smells, even though the former are less frequent. It

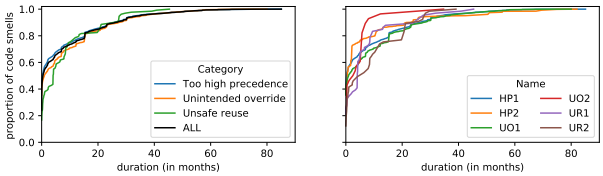

Figure 9: Cumulative proportion of code smells in function of the time elapsed since the first commit.

Table 5: Statistics about newly added files with smells.

	# files (%)	# roles (%)	# smells (%)
Too high precedence	3,138 (56.8)	2,112 (61.0)	8,667 (42.7)
Unintended override	1,568 (48.2)	1,126 (52.8)	3,333 (30.6)
Unsafe reuse	25 (29.4)	22 (35.5)	35 (26.7)
All	3,966 (55.3)	2,580 (60.6)	12,035 (38.4)

took 0.53 months and 0.96 months for 50% of the *HP2* and the *UO1* smells to appear respectively. Overall, 50% of all smells appeared within 0.46 months since the role's initial commit. Further analysis shows that 16.8% of all smells were already present in the role's initial commit. This is the case for 1,359 roles or 31.9% of all roles with smells. Of these roles, 82.3% contained *too high precedence* smells, 38.5% contained *unintended override* smells, while only 0.66% contained *unsafe reuse* smells.[4]

As many roles already contain smells from their first commit, we investigated whether the same applies to the addition of new code files. We found that only 7,171 out of the 58,881 code files that were analysed (12.2%) contained smells. We then determined the number of smells within those files at the commit in which the file was added. Table 5 summarises the results. We observe that 55.3% of smelly files already had the smells when being added to the corresponding role. Furthermore, 42.7% of *too high precedence* smells are introduced together with their encompassing file. Similarly, for 35.5% roles that contain *unsafe reuse* smells, at least one of these smells was introduced together with the file.

Finally, we investigated the amount of time required before a smell is removed since its original introduction. Since the majority of smells are still present in a role's last snapshot and have therefore not been fixed (cf. RQ_2), we used a survival analysis [18][5] to estimate the probability over time for a smell to be removed, with respect to the date of the first commit introducing the smell. Figure 10 shows the Kaplan-Meier survival curves for our smells. The curves for *unintended override* and *unsafe reuse* overlap, which suggests that there is no clear difference in terms of fixing time for smells in these two categories. In contrast, the *too high precedence* curve does not overlap with any other. It also takes longer before smells of this category are fixed. It takes 17.6 and 12.8 months for 50% of *unintended override* and *unsafe reuse* smells to be fixed, respectively, while it takes 37.4 months for 50% of *too high precedence* smells to be fixed. Log-rank tests confirmed statistical differences between

[4] A project might have code smells of different categories.
[5] Survival analysis creates a model estimating the survival rate of a population over time until the occurrence of an event, considering the fact that some subjects may leave the study, while for others the event of interest might not be observed.

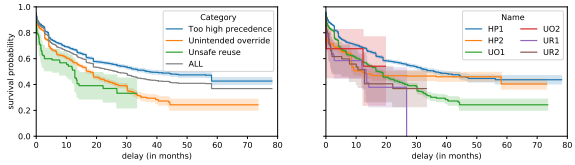

Figure 10: Probability for a code smell to fixed w.r.t. to the first commit in which the smell appeared, grouped by smell category and name. The shaded coloured areas represent the confidence intervals ($\alpha = 0.05$) of the survival curves.

the *too high precedence* category and the other two categories ($p < 0.002$ [6]). This difference is mainly due to *HP1* smells (in the right figure). Conversely, the tests did not confirm such a difference between the *unintended override* and *unsafe reuse* categories ($p = 0.016$). Without categorizing smells, we found that it takes 26.7 months for half of all smells to be fixed.

Findings: Half of the discovered code smells appeared within the first month of a role's existence, with 16.8% already existing in the first commit. Smells take much longer to be removed, still having a survival probability of over 50% after more than 2 years.

6 DISCUSSION

Assessing a role for the variable smells proposed in this paper can warn practitioners about maintainability issues, and point them towards potential defects. Almost a fifth of the roles in our dataset is affected (RQ_2), and not seldom from their first commit onwards (RQ_4). Next to a call to arms for better tool support and safer IaC languages, this suggests that there are likely misunderstandings about Ansible's semantics among practitioners.

Practitioners' perception of smells. Although we have not consulted with practitioners for this study, we found various fixing commits (cf. RQ_4) suggesting that practitioners agree with these smells. For instance, one developer fixed an unnecessary `set_fact` usage to address an issue where built-in facts were overridden [8], whereas another developer removed three such usages to improve their role's quality score [13], a metric computed by Ansible Galaxy to rate roles based on syntactical linter warnings. Similarly, various smells related to unsafe reuses of impure expressions correspond to defects that were later addressed [2, 22]. This suggests that the discovered smells could be used to warn developers about potential defects and to suggest fixes. We leave this as future work.

Common pitfalls. During the validation of the smell detector (RQ_1), we discovered certain patterns in the misunderstandings of Ansible's semantics. We discuss them as knowing which pitfalls to avoid can be valuable for practitioners. For instance, many of the *unnecessary* `include_vars` (*HP2*) instances were caused by practitioners loading files defining their default and role variables manually. Not only is this redundant since these files are loaded implicitly, it also unconditionally overrides the already-defined

[6] The lower $p-$value is defined using a Bonferroni correction. We originally considered a global confidence level of 95%, corresponding to a significance level of $\alpha = 0.05$, *i.e.*, H_0 is rejected if $p < \frac{0.05}{n}$. In our case $n = 18$, *i.e.*, $p < 0.002$.

```
# Contents of tasks/configure-preferences.yml
- include_tasks: types/pin.yml
  vars:
    apt__pin: ...

# Contents of tasks/types/pin.yml
- template: ...
  vars:
    apt__pin: ... # UO2: Shadowed by include param on line 5
```

Figure 11: *UO2* smell in `Turgon37.apt` role, commit `948c785c`.

```
# Contents of tasks/tasks_python_fallback.yml
- shell: yum -y install python3
  register: result # Define `result`
  until: result is not failed # Use `result` from line 6 locally
- shell: yum -y install python3-devel python3-setuptools
  register: result # UO1: Unconditional redefinition of `result`
  until: result is not failed # Use `result` from line 12 locally
```

Figure 12: *UO1* smell in `softasap.sa_docker` role, commit `96548e70`.

```
# Contents of tasks/configure-preferences.yml
- include_tasks: addkey.yml
  loop: "{{ keys }}" # Defines `item`
  vars: # These expressions should use `item` defined on line 3
    ssh_config_dir: "{{ item.ssh_config_dir }}"
    ssh_host_usewith: "{{ item.ssh_host_usewith }}"

# Contents of tasks/addkey.yml
- file:
    path: "{{ ssh_config_dir }}" # OK: Uses `item` defined on line 3
- blockinfile:
    path: "{{ ssh_config_dir }}" # UR2: `item` redefined on line 15
  loop: "{{ ssh_host_usewith }}" # Redefines `item`
```

Figure 13: *UR2* smell in `dottgonzo.add_ssh_key` role, commit `7140f935`.

variables (leading to *UO1* smells) and leads to unnecessarily high precedence, which may make it impossible for a client play to override these variable definitions to customise the role's behaviour. Moreover, many of the *unused override* (*UO2*) smells were caused by the use of *include parameters*, which have the highest possible precedence, exemplified in Figure 11.

Additional language features. We also found apparent workarounds for a lack of language support for particular variable use cases, which can serve as the motivation to introduce new language features to address these use cases. A number of *HP2* smells, although true positives, would be difficult to address since the dynamically loaded variable files were used to modularise variable definitions. Similarly, many unconditional overrides (*UO1*) affected variables defined as task results through the `register` key. Although these variables were only used locally, `register` defines them globally, leading to conflicts as shown in Figure 12. Finally, all of the sampled data dependence changes (*UR2* smells) were caused by a form of dependency injection, where the initialiser of a global variable depended on a variable defined in the local scope of a task. Although many of these redefinitions seemed intentional, we classified them as true positives since the detector correctly identified the change in variable values and this usage of variables

may be confusing. Nonetheless, this smell uncovered another instance of a real defect in an Ansible role, illustrated in Figure 13.

Additional tool support. Our results can also serve to motivate future research on infrastructure as code maintainability. For instance, unnecessary usages of `set_fact` (*HP1*) are the most common smell we detected (cf. RQ_2). Due to its imperative nature and high precedence, we believe these to have a negative impact on the clarity and maintainability of infrastructure code. Future work may investigate its use cases, and possibly suggest safer alternatives. Similarly, the results of RQ_2 show that smells are introduced more often than they get fixed, while RQ_4 shows that it may take a long time for a fix to arrive. Therefore, tool support for IaC practitioners to detect, comprehend, and repair these smells may be beneficial.

7 THREATS TO VALIDITY

We present the threats to validity of this paper following the classification and recommendations of Wohlin et al. [43]. The main threat to *construct validity* stems from the prototypes that we built to detect the smells. Our PDG builder and smell detector suffer from technical limitations, discussed in the next section. They may have introduced false negatives and false positives in our results. However, we have mitigated the threat of the latter by manually validating the precision of our prototypes on a random sample (cf. RQ_1). Second, the removal of a smell does not necessarily imply that it was fixed, as the containing code might have been deleted. The survival probabilities shown in RQ_4 thus form a lower bound.

Internal validity concerns choices and factors internal to the study that could influence the observed results. We decided to only study roles that are hosted on Ansible Galaxy, while there might be other roles available in GitHub but not distributed via Ansible Galaxy. However, since Ansible Galaxy is Ansible's official hub for sharing Ansible content, we think that the majority of roles are hosted on it which means that our results are representative for the majority of Ansible roles' developers and users.

Conclusion validity concerns the degree to which the conclusions we derived from our data analysis are reasonable. Since our conclusions are mostly based on empirical observations, our work is unlikely to be affected by such threats. However, it is important to mention that our conclusions concern Ansible role smells which are not necessarily bugs, but are indicative of bad practices that may lead to bugs when the role is included in a play.

As a threat to *external validity*, our findings cannot be generalised beyond Ansible roles, *i.e.,* to other Ansible components such as playbooks, or to other infrastructure as code languages such as Puppet and Chef. However, it is possible to replicate the design of our study for the aforementioned IaC languages.

8 TECHNICAL LIMITATIONS

Our PDG builder only supports a subset of the Ansible language. Most importantly, it currently does not consider *handlers*, which are special types of tasks that are executed out-of-order. Moreover, certain task keys are not inspected during the analysis. Therefore, any usages of variables in such unsupported elements will not be present in the graph and cannot be considered in the smell detection. Nonetheless, the builder supports the most common and most important components of the Ansible language, including task actions, action arguments, conditionals, loops, and all forms of variable definitions.

Secondly, certain operations are too dynamic for our PDG builder to statically approximate. This includes dynamically including tasks (`include_tasks`) or variables (`include_vars`) where the file name of the included file is not a literal. We currently ignore such actions, and any code smells occurring in tasks or variables that are included in such manner may be missed. Similarly, although we indicate when variables are conditionally defined, the PDG builder does not consider the conditions under which variables may be defined when resolving variable references. It may therefore use the wrong variable definitions under certain circumstances. However, we did not observe a large negative impact of this limitation during our validation. We leave properly resolving conditional definitions as future work.

Finally, the algorithm used to determine whether an expression is pure is naive and can lead to an under-approximation. For instance, it does not support user-defined filters and tests, and considers them impure by default. We do not consider this to be an inherent limitation of our approach, since the implementation of this algorithm can be interchanged with an improved implementation without substantial changes to Algorithm 1. However, an improved implementation aiming to automatically determine purity of user-defined filters and tests would need to perform complicated analysis of non-Ansible code. Alternatively, if the current algorithm was to be used in a practical tool implementation, it would be straightforward to allow a user to configure their own list of pure tests and filters.

9 RELATED WORK

Research on Infrastructure as Code (IaC) has been categorised into 4 categories [32], namely tools and frameworks, empirical studies, use cases and experience reports, and testing. Identifying defects in IaC projects can be challenging [10] and this has led many researchers to study good and bad practices in IaC, to identify code smells and accompanying detectors, or to propose and measure quality metrics [5, 10, 19, 31, 33, 35, 37, 42]. Other researchers have instead focused on verifying semantic properties of IaC scripts [14, 15, 36, 38] using specialised semantic models.

One such property is idempotence of script execution, which ensures that running an IaC script on an already-configured infrastructure does not lead to unnecessary changes. Although such idempotence issues can be caused by reuse of impure expressions, they are separate issues. Therefore, while some overlap may exist between problems detected by tools such as Rehearsal [36] and our *Unsafe reuse of impure expression (UR1)* detection rule, the reported problems are different. Primarily, lack of idempotence can be caused by various other reasons, and it is also possible for execution to still be idempotent while impure expressions are reused. Moreover, idempotence is a property of whole-script execution, whereas our *UR1* smell concerns single expressions and is thus more fine-grained. Note also that Rehearsal targets Puppet code, and that no static idempotence checker for Ansible code exists. Although Rehearsal's technique can possibly be replicated for Ansible code, Ansible's unique semantics may pose a significant challenge.

Existing analysis tools for Ansible [4, 5, 27, 34] mostly remain at the syntactical level and do not compute complex control nor data flow. This may in part be due to the intricate variable precedence rules and lazily-evaluated template expressions. However, the lack of data flow information leads to tool limitations. For example, Opdebeeck et al. [27] introduce a structural model that abstracts over Ansible syntax and use it to distil change metrics between Ansible role releases. As the change distiller cannot trace variables to their definitions, it is limited in recognising variable renaming changes and distils removal and addition changes instead. The so-called SRT introduced by Dai et al. [4] is similar to Opdebeeck et al.'s structural model, but also models sequential ordering of tasks. Control flow induced by loops and conditions is not supported. The SRT is used to analyse the security of shell scripts executed by Ansible tasks. To this end, template expressions are simplified by substituting literal values for statically-known variables. However, the tool does not fully account for the complex Ansible semantics.

We have proposed a Program Dependence Graph based representation of Ansible roles that models their control and data flow succinctly. PDG-based representations have already proven themselves as enablers of advanced development tooling. Applications include optimisation [7, 29] and program slicing [28, 40, 41], code clone detection [20], refactoring [12, 30], and static security analysis [11, 17]. Moreover, derivatives of the PDG representation have been paired with graph mining algorithms to recommend code snippets [24], assess migration effort [25], mine code change patterns [23], and detect defects in library usages [1].

This paper is the first to introduce a PDG-based representation for Ansible code. We have demonstrated its applicability by detecting variable-related code smells. To this end, the representation and its builder accurately account for the Ansible's variable precedence rules and lazily-evaluated template expressions. It is the first representation that captures this semantics in such detail, and we hope that it will enable developers and researchers to build tooling for and study software engineering problems of infrastructure code.

10 CONCLUSION

Mistakes in infrastructure configuration data are a major cause of infrastructure defects. The intricate details of Ansible's variable and expression semantics may lead developers to unwittingly introduce such defects into their infrastructure code. Moreover, this semantics encumbers program comprehension and therefore introduces maintenance difficulties. We have proposed a catalogue of 6 novel code smells related to the usage of Ansible variables. The smells indicate unsafe reuses of variables, unintended overrides of variables, and variables defined with unnecessarily high precedence. To detect the proposed smells, we have transposed the concept of program dependence graphs from application to infrastructure code. Using the detector, we have conducted an empirical analysis into the prevalence and lifetime of the smells in over 20,000 open-source, reusable Ansible roles. The results show that these smells are becoming increasingly common and may take a long time before they get fixed. Furthermore, the rate at which new smells are introduced outpaces their fixes. Some smells often co-occur within the same role, suggesting that one smell may cause another. We have also found evidence in the form of fixing commits suggesting that certain smells may be indicative of infrastructure defects. The proposed smells and the accompanying detector can therefore serve as a valuable asset for practitioners to spot maintainability and reliability issues in their infrastructure code.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their helpful comments and suggestions. This work is supported by the Research Foundation Flanders under Grant No. 1SD4321N and the Excellence of Science project 30446992 "SECO-Assist" financed by Research Foundation Flanders and F.R.S. - FNRS.

REFERENCES

[1] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. 2019. Investigating next Steps in Static API-Misuse Detection. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. 265–275. https://doi.org/10.1109/MSR.2019.00053

[2] Marc Aschmann. 2015. *Commit in servergrove.symfony2 role.* Retrieved January 19, 2022 from https://github.com/servergrove/ansible-symfony2/commit/2285f4d8e9f4da7b33d08ade38102869a45f1455

[3] Brian Coca. 2018. *Ansible Proposal: redesign variable interface.* Retrieved January 9, 2022 from https://github.com/ansible/proposals/issues/127

[4] Ting Dai, Alexei Karve, Grzegorz Koper, and Sai Zeng. 2020. Automatically Detecting Risky Scripts in Infrastructure Code. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC '20)*. 358–371. https://doi.org/10.1145/3419111.3421303

[5] Stefano Dalla Palma, Dario Di Nucci, Fabio Palomba, and Damian Andrew Tamburri. 2020. Toward a catalog of software quality metrics for infrastructure code. *J. Syst. Softw.* 170, Article 110726 (Dec. 2020), 8 pages. https://doi.org/10.1016/j.jss.2020.110726

[6] Christophe-Marie Duquesne. 2021. *Ansible Issue: loop_var is taking precedence over variable defined in the var block of the task.* Retrieved January 9, 2022 from https://github.com/ansible/ansible/issues/75616

[7] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Trans. Program. Lang. Syst.* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[8] Fran Fitzpatrick. 2019. *Issue: Role Should NOT Smash Ansible Facts.* Retrieved January 19, 2022 from https://github.com/dj-wasabi/ansible-zabbix-agent/issues/207

[9] John Graham-Cunning. 2020. *Cloudflare outage on July 17, 2020.* Retrieved January 9, 2022 from https://blog.cloudflare.com/cloudflare-outage-on-july-17-2020/

[10] Michele Guerriero, Martin Garriga, Damian A. Tamburri, and Fabio Palomba. 2019. Adoption, Support, and Challenges of Infrastructure-as-Code: Insights from Industry. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution, Industrial Track (ICSME '19)*. 580–589. https://doi.org/10.1109/ICSME.2019.00092

[11] Christian Hammer and Gregor Snelting. 2009. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *Int. J. Inf. Sec.* 8, 6 (Dec. 2009), 399–422. https://doi.org/10.1007/s10207-009-0086-1

[12] Keisuke Hotta, Yoshiki Higo, and Shinji Kusumoto. 2012. Identifying, Tailoring, and Suggesting Form Template Method Refactoring Opportunities with Program Dependence Graph. In *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR '12)*. 53–62. https://doi.org/10.1109/CSMR.2012.16

[13] Mah Chia Hui. 2020. *Commit: updated code to get higher score on ansible galaxy.* Retrieved January 19, 2022 from https://github.com/Senzing/ansible-role-stream-producer/commit/57f0f85f389aa93677465952d528040082d2cddc

[14] Waldemar Hummer, Florian Rosenberg, Fábio Oliveira, and Tamar Eilam. 2013. Testing Idempotence for Infrastructure as Code. In *Proceedings of the 14th ACM/IFIP/USENIX International Middleware Conference (Middleware '13)*. 368–388. https://doi.org/10.1007/978-3-642-45065-5_19

[15] Katsuhiko Ikeshita, Fuyuki Ishikawa, and Shinichi Honiden. 2017. Test Suite Reduction in Idempotence Testing of Infrastructure as Code. In *Proceedings of the 11th International Conference on Tests and Proofs (TAP@STAF '17)*. 98–115. https://doi.org/10.1007/978-3-319-61467-0_6

[16] Santosh Janardhan. 2021. *More details about the October 4 outage.* Retrieved January 9, 2022 from https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/

[17] Andrew Johnson, Lucas Waye, Scott Moore, and Stephen Chong. 2015. Exploring and Enforcing Security Guarantees via Program Dependence Graphs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. 291–302. https://doi.org/10.1145/2737924.2737957

[18] John P. Klein and Melvin L. Moeschberger. 2013. *Survival Analysis: Techniques for Censored and Truncated Data* (2nd ed.). Springer.

[19] Indika Kumara, Martín Garriga, Angel Urbano Romeu, Dario Di Nucci, Fabio Palomba, Damian Andrew Tamburri, and Willem-Jan van den Heuvel. 2021. The do's and don'ts of infrastructure code: A systematic gray literature review. *Inf. Softw. Technol.* 137, Article 106593 (Sept. 2021), 20 pages. https://doi.org/10.1016/j.infsof.2021.106593

[20] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '06)*. 872–881. https://doi.org/10.1145/1150402.1150522

[21] Kief Morris. 2016. *Infrastructure as Code: Managing Servers in the Cloud* (1st ed.). O'Reilly.

[22] Michal Muransky. 2020. *Commit in MonolithProjects.github_actions_runner role*. Retrieved January 19, 2022 from https://github.com/MonolithProjects/ansible-github_actions_runner/commit/94616c56a760f84e5738eac9b8c0534e935c2499

[23] Hoan Anh Nguyen, Tien N. Nguyen, Danny Dig, Son Nguyen, Hieu Tran, and Michael Hilton. 2019. Graph-Based Mining of in-the-Wild, Fine-Grained, Semantic Code Change Patterns. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 819–830. https://doi.org/10.1109/ICSE.2019.00089

[24] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-Based Mining of Multiple Object Usage Patterns. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE '09)*. 383–392. https://doi.org/10.1145/1595696.1595767

[25] Ruben Opdebeeck, Johan Fabry, Tim Molderez, Jonas De Bleser, and Coen De Roover. 2021. Mining for Graph-Based Library Usage Patterns in COBOL Systems. In *Proceedings of the 28th International Conference on Software Analysis, Evolution and Reengineering, Industry Track (SANER '21)*. 595–599. https://doi.org/10.1109/SANER50967.2021.00072

[26] Ruben Opdebeeck, Ahmed Zerouali, and Coen De Roover. 2021. Andromeda: A Dataset of Ansible Galaxy Roles and Their Evolution. In *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR '21)*. 580–584. https://doi.org/10.1109/MSR52588.2021.00078

[27] Ruben Opdebeeck, Ahmed Zerouali, Camilo Velázquez-Rodríguez, and Coen De Roover. 2021. On the practice of semantic versioning for Ansible galaxy roles: An empirical study and a change classification model. *J. Syst. Softw.* 182, Article 111059 (Dec. 2021), 21 pages. https://doi.org/10.1016/j.jss.2021.111059

[28] Karl J. Ottenstein and Linda M. Ottenstein. 1984. The Program Dependence Graph in a Software Development Environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE '84)*. 177–184. https://doi.org/10.1145/800020.808263

[29] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2016. Dependence-Driven Delimited CPS Transformation for JavaScript. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '16)*. 59–69. https://doi.org/10.1145/2993236.2993243

[30] Laure Philips, Joeri De Koster, Wolfgang De Meuter, and Coen De Roover. 2018. Search-based Tier Assignment for Optimising Offline Availability in Multi-tier Web Applications. *The Art, Science, and Engineering of Programming* 2, 2, Article 3 (2018), 29 pages. https://doi.org/10.22152/programming-journal.org/2018/2/3

[31] Akond Rahman, Effat Farhana, Chris Parnin, and Laurie Williams. 2020. Gang of Eight: A Defect Taxonomy for Infrastructure as Code Scripts. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 752–764. https://doi.org/10.1145/3377811.3380409

[32] Akond Rahman, Rezvan Mahdavi-Hezaveh, and Laurie A. Williams. 2019. A systematic mapping study of infrastructure as code research. *Inf. Softw. Technol.* 108 (April 2019), 65–77. https://doi.org/10.1016/j.infsof.2018.12.004

[33] Akond Rahman, Chris Parnin, and Laurie Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. 164–175. https://doi.org/10.1109/ICSE.2019.00033

[34] 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.* 30, 1, Article 3 (Jan. 2021), 31 pages. https://doi.org/10.1145/3408897

[35] Akond Rahman and Laurie A. Williams. 2019. Source code properties of defective infrastructure as code scripts. *Inf. Softw. Technol.* 112 (Aug. 2019), 148–163. https://doi.org/10.1016/j.infsof.2019.04.013

[36] Rian Shambaugh, Aaron Weiss, and Arjun Guha. 2016. Rehearsal: A Configuration Verification Tool for Puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. 416–430. https://doi.org/10.1145/2908080.2908083

[37] 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. https://doi.org/10.1145/2901739.2901761

[38] Thodoris Sotiropoulos, Dimitris Mitropoulos, and Diomidis Spinellis. 2020. Practical Fault Detection in Puppet Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering (ICSE '20)*. 26–37. https://doi.org/10.1145/3377811.3380384

[39] StackExchange, Inc. 2021. *2021 Annual StackOverflow Developer Survey*. Retrieved January 9, 2022 from https://insights.stackoverflow.com/survey/2021

[40] Quentin Stiévenart, Dave Binkley, and Coen De Roover. 2021. QSES: Quasi-Static Executable Slices. In *Proceedings of the 21th IEEE International Working Conference on Source Code Analysis and Manipulation, New Ideas and Emerging Results Track (SCAM '21)*. 209–213. https://doi.org/10.1109/SCAM52516.2021.00033

[41] Quentin Stiévenart, Dave Binkley, and Coen De Roover. 2022. Static Stack-Preserving Intra-Procedural Slicing of WebAssembly Binaries. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. accepted.

[42] Eduard van der Bent, Jurriaan Hage, Joost Visser, and Georgios Gousios. 2018. How Good is Your Puppet? An Empirically Defined and Validated Quality Model for Puppet. In *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '18)*. 164–174. https://doi.org/10.1109/SANER.2018.8330206

[43] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B. Regnell, and A. Wesslen. 2000. *Experimentation in Software Engineering - An Introduction*. Kluwer.