Understanding Execution Faults in Ansible-based Configuration Management

by

Wan Syun Cheng

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Auburn, Alabama
May 10, 2025

Keywords: ansible, rocq, devops, fault, infrastructure as code, verification

Approved by

Akond Rahman, Chair, Assistant Professor of Computer Science and Software Engineering
Ali Ghanbari, Assistant Professor of Computer Science and Software Engineering
Mark Yampolskiy, Associate Professor of Computer Science and Software Engineering

Abstract

Ansible is a popular tool for implementing the practice of infrastructure as code. While usage of the tool yields benefits for practitioners, faulty execution of Ansible configuration scripts can cause serious consequences, such as crashes. An analysis of execution faults, i.e., faults that occur in Ansible's executor component, can aid practitioners in validation and verification activities to facilitate reliable execution of configuration scripts. An empirical study of 33 execution faults collected from the open-source Ansible repository was conducted. I derive 6 fault categories and 4 fault consequences. By analyzing 33 issue reports related to execution faults, I also derive properties to perform verification of the executor component in Ansible. Using formal verification, I reproduce one fault, and identify one new fault in the executor component of Ansible, which has been confirmed to be valid by Ansible contributors. Dataset and source code used for this thesis are publicly available online.

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Akond Rahman, for his invaluable guidance, continuous support, and encouragement throughout the course of this research. His insight, patience, and expertise have played a critical role in shaping both this thesis and my development as a researcher.

My research was partially funded by the U.S. National Science Foundation (NSF) under Award numbers 2247141 and 2310179. I am grateful for this support, which made this work possible.

I am also sincerely thankful to my committee members, Dr. Ali Ghanbari and Dr. Mark Yampolskiy, for their thoughtful feedback, constructive suggestions, and for generously sharing their time and expertise to help improve the quality of this work.

The combined mentorship of my advisor and committee, along with the support of the NSF, have been instrumental in the successful completion of this thesis.

I would also like to thank Auburn University for fostering an outstanding academic environment that has supported my personal and professional growth. I am especially grateful to the Department of Computer Science and Software Engineering for its unwavering support, valuable resources, and the many opportunities it has provided throughout my time here. The encouragement and dedication of the faculty, staff, and peers have made this journey both enriching and memorable.

Lastly, I extend my heartfelt thanks to my family and friends for their unwavering emotional support. Their constant encouragement and belief in me have been a source of strength throughout this journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ansible is a widely used automation tool for configuration management, application deployment, and system orchestration. It is popular to implement the practice of infrastructure as code (IaC). IaC automates the management of computing infrastructure using scripts and software development techniques. Instead of manually configuring servers, networks, or other infrastructure components, IaC empowers practitioners to define infrastructure through configuration scripts. Ansible simplifies infrastructure management by allowing users to define desired system states using configuration scripts. By providing an agentless and declarative approach, Ansible has gained popularity among practitioners for its ease of use and scalability [1]. However, despite its benefits, the execution of configuration scripts is prone to faults. As IaC programs grow in scale and complexity, ensuring their reliability becomes increasingly critical [2]. Unreliable configuration management can cause serious consequences, such as outages for millions of end-users. Zhang et al. investigated defects in the Ansible infrastructure orchestrator, finding the 'Module' component having the most defects of 69.23% and 'Executor' component comes in third place with 6.23% [3]. Moreover, Hassan et al. categorized known defects related to state reconciliation in the Ansible repository into eight categories. They reported nine previously unknown defects spanning three of these categories [4].

Recently, researchers have shown interest in defects in IaC which results in multiple papers. Ikeshita et al. introduced a technique for reducing test suites in configuration scripts [5]. Rahman's research focuses on taxonomy for defects in configuration scripts [6]. However, Sokolowski had addressed the lack of usage of testing and formal verification in IaC [2]. They noticed that developers mainly depend on integration testing, which can boost assurance in end-to-end functionality. Pascalis addresses the problem by conducting formal verification using model checking to verify a custom IaC tool [7]. Recent research primarily investigates theoretical frameworks or custom languages for configuration management verification. Notably, there is a lack of focus on ensuring the reliability of established, real-world IaC tools, using formal verification. Despite advancements in detecting execution faults in configuration management, a critical gap exists.

This thesis answers the following research questions:

- ***RQ1***: What are the categories of faults that occur in the Ansible executor component? What are the consequences of these faults?
- ***RQ2***: How can one use identified fault categories to verify the Ansible executor component?

I analyze 33 execution faults obtained from Ansible issue reports and pull requests that are related to the Ansible executor component [8] [9]. From the analysis, I derive categories of faults and their corresponding critical properties. I systematically apply these critical properties to a reimplemented version of Ansible in Rocq [10], a theorem prover, to show how they can be used to detect faults in Ansible. Dataset and source code used for this thesis are publicly-available online [11].

The contributions of this thesis is listed as follows:

- A list of derived fault categories for faults that occur in the executor component;
- A list of consequences for faults that occur in the executor component; and
- A methodology that showcases how to use formal verification for the executor component to identify new execution faults.

Chapter 2

Background

This study involves analysis of faults in Ansible and formal verification of the Ansible executor component. This section introduces the research background, including Ansible, formal verification, and studies done regarding the reliability of IaC tools.

2.1 Infrastructure as Code (IaC)

The emergence of IaC signifies a fundamental change in the way modern IT operations manage and provision computing infrastructure. This approach centers on the systematic utilization of machine-readable scripts or configuration files to automate the deployment, configuration, and management of infrastructure components [1]. Unlike traditional methods that rely on manual configuration or ad-hoc intervention, IaC conceptualizes infrastructure as software-defined entities. Moreover, IaC leverages version control systems and automation tools to ensure consistent and efficient management practices.

2.2 Ansible

Ansible is an open-source automation framework widely utilized for configuration management, application deployment, and task automation [12]. Its declarative nature allows users to define system states through YAML-based playbooks, which are subsequently processed and executed by Ansible's core components [13].

Among all the Ansible modules, the 'executor' module is tasked with determining the next set of actions to take, based on configurations parsed from the playbooks. The Ansible executor is responsible for executing compiled tasks across the designated target hosts [8]. It orchestrates task execution, manages host states, and enforces the logic defined within the playbook. The executor also ensures compliance with execution strategies, such as serial execution or parallel task distribution, thereby optimizing automation workflows for reliability and efficiency. And thus, it plays a crucial role in the execution of automation workflows.

2.3 Formal Verification

Formal verification is a methodology to ensure the correctness of hardware and software systems through mathematical proof [14] [15]. Unlike traditional testing approaches that rely on empirical validation through sample inputs or scenarios, formal verification aims to provide guarantees about the behavior of a system under all possible conditions. The primary goal of formal verification is to identify and eliminate defects or vulnerabilities that may lead to system failures, security breaches, or unintended behavior [16]. By constructing mathematical proofs or logical arguments, formal verification can establish whether a system adheres to its specifications and meets desired correctness criteria, such as functional correctness, safety properties, or absence of certain types of errors. Formal verification includes various techniques and approaches, each suited to different aspects of system design and verification. Theorem Proving is one of the approaches that involves using formal logic and mathematical proofs to demonstrate that a system design or algorithm satisfies its intended specifications or properties [17].

At its essence, formal verification leverages mathematical techniques, particularly from logic and automata theory, to model and analyze system designs or software algorithms. This process typically involves defining formal specifications that precisely describe the intended behavior or properties that the system must satisfy. These specifications are then subjected to formal analysis using tools such as theorem provers, model checkers, and symbolic execution engines.

2.4 Related Work

The growing complexity of modern IaC programs necessitates a shift from simple configuration scripts to robust software development practices. This is because IaC failures can disrupt application deployment and expose security vulnerabilities. Therefore, traditional software engineering tools for ensuring reliability, like testing and verification, are utilized in the IaC domain on both the script and the automation tool itself.

2.4.1 Testing in IaC

While IaC is popular for mass deployment, researchers identified a lack of testing practices and frameworks, which makes complex IaC programs hard to debug, challenging their usability and correctness [1] [5]. IaC tools such as Ansible, Puppet, and Chef utilize unit testing, integration testing, and sanity testing to guarantee their correctness and code readability of their script executor [12] [18] [19]. Aside from the core module they present, some IaC tools also provide support for testing plugins and custom modules. For instance, pull requests for Ansible automatically trigger Azure Pipelines to detect bugs [20]. Besides the IaC tool itself, the configuration script fed to the tool can also cause problems. Ikeshita et al. reported that testing configuration scripts can be time-consuming, and they proposed a test suite reduction technique for configuration scripts by combining testing and static verification [5]. Hummer et al. created a tool to automatically test the idempotence of configuration scripts [21]. In IaC, idempotence means that applying the same configuration multiple times will always produce the same result, without introducing unintended changes or side effects. By doing so, they guarantee the scripts' repeatability and their ability to consistently reach the desired end state. ProTI was presented as an extensible unit testing approach for IaC, focusing on Pulumi TypeScript [22]. ProTI combines modern unit testing and fuzzing with user-defined specifications, allowing developers to tailor tests to their specific needs. Additionally, Sokolowski et al. extended ProTI by adding Automated Configuration Testing (ACT) [23]. ACT uses mocking and plugins for test generation and validation, addressing the low adoption of testing in IaC development. Besides focusing on testing, they mentioned that formal verification can provide more confidence in the correctness and reliability of IaC [2].

2.4.2 Formal Verification in IaC

Recent research has focused on improving the reliability and verification of configuration scripts. Hummer et al. introduced a model-based testing framework for verifying idempotence in IaC automations, using state transition graphs to generate test cases [21]. Ikeshita presents a method for efficiently checking idempotence in configuration management, combining testing and static verification techniques [5]. Rahman identifies key properties in defective configuration scripts like filesystem operations and infrastructure provisioning, providing a basis for prioritizing validation and verification efforts [24]. However, as mentioned by Sokolowski et al., formal verification of configuration script can be time-consuming, since every script written needs to be verified [2]. The formal verification of IaC executor bypasses the problem of needing to verify every script, reducing the time needed by the practitioners to guarantee the correctness. It is a critical aspect to increase the correctness of IaC automation; however, few research has been done in this aspect. Pascalis introduced DOML-MC, which is a model checker for DOML, an IaC tool proposed by the same research group [7]. DOML-MC guarantees the repeatability and convergence of automation by using model checking with SMT.

Chapter 3

RQ1: Fault categories and consequences for Ansible Executor component

In this chapter, existing issue reports and pull requests in the executor component are classified based on the cause of the fault and the underlying reasons behind it. The filters applied include being tagged with executor/* and labeled as a bug fix. This filtering reduced the number of pull requests from 51,793 to 62. Each of these 62 pull requests, related to the 'executor' component, was manually reviewed. The issues and commits were analyzed further to identify the root cause and categorize them. Out of the 62 pull requests, 33 of them provided sufficient information to enable such analysis. A qualitative analysis technique called open coding is applied to derive the fault categories. Open coding is used to derive themes from unstructured natural language text [25].

Previous work by Hassan et al. presented a defect taxonomy focused on state reconciliation faults across the Ansible system [4]. In their context, a state refers to the set of software artifacts and configurations that define the target infrastructure. The Infrastructure-as-Code (IaC) orchestrator interprets Ansible scripts to derive the desired infrastructure state, compares it against the current state, and applies changes only when discrepancies are found. In contrast, my work focuses on execution faults within Ansible's executor component. The executor is responsible for determining the next set of actions to perform based on configurations parsed from playbooks. Rather than targeting mismatches in infrastructure state, my analysis centers on faults that occur during the task execution process, specifically how the executor processes and coordinates execution logic.

```python
import tempfile
...
def import_non_local(name):
    f, pathname, desc = imp.find_module(name, sys.path[1:])
    module = imp.load_module(name, f, pathname, desc)
    f.close()
    return module
tempfile = import_non_local('tempfile')
```

Figure 3.1: An example Dependency-related fault

3.1 Categories

6 categories of faults are identified out of the 33 execution faults analyzed in the Executor component of Ansible. The number of faults belonging to each category is shown in Table 3.1.

Table 3.1: Fault categories and number of faults

Category	Count
Dependency-related Faults	9
State Mismanagement	8
Incorrect Detection/Analysis of User Input	7
Variable Misreference	6
Incorrect Task Iteration Logic	2
Typos	1
Total	33

3.1.1 Dependency-related Faults

This category of faults arises from incorrect usage of package dependencies. Updates to a dependency library or language used in Ansible (such as Python, Ansible collections, or system libraries) can introduce compatibility issues, unexpected behavior, or failures in playbook runs. These dependency-related faults can be particularly challenging to diagnose, as they often emerge only after external libraries or components have been updated.

Example. Figure 3.1 shows a pull request [1] of Ansible repository. This fault is caused by an update in Python 2.7 where calling import tempfile loads tempfile.py instead of tempfile.

3.1.2 Variable Misreference

This category of faults occurs when a variable is referenced without being properly checked or initialized beforehand. If a variable does not have a default value or is mistakenly referenced

[1] https://github.com/ansible/ansible/pull/23656/files

```python
def unfrack_paths(option, opt, value, parser):
    paths = getattr(parser.values, option.dest)
    if paths is None:
        paths = []

    if isinstance(value, string_types):
        paths[:0] = [unfrackpath(x) for x in value.split(os.pathsep)]
        paths[:0] = [unfrackpath(x) for x in value.split(os.pathsep) if x]
    elif isinstance(value, list):
        paths[:0] = [unfrackpath(x) for x in value]
        paths[:0] = [unfrackpath(x) for x in value if x]
```

Figure 3.2: An example variable misreference fault.

before being assigned, accessing it without checking whether it holds a value or a specific type can result in a TypeError.

Example. Figure 3.2 shows a pull request [2] of Ansible repository. This fault is caused by appending to an uninitialized variable list without checking if it is initialized or not. Thus, items are not appended successfully, causing the Python interpreter to raise TypeError.

3.1.3 Incorrect Task Iteration Logic

A task is the smallest unit of action you can automate using an Ansible playbook [13]. Playbooks typically contain a series of tasks that serve a goal, such as to set up a web server, or to deploy an application to remote environments.

In Ansible, a sequence of tasks is a series of actions that are executed in order on the target hosts. Each task in the sequence typically represents a specific configuration or action to be carried out. The Ansible executor reaches the final goals of the developer by iterating through the correct task sequence.

This category of faults occurs when the executor of Ansible is iterating through the tasks in an incorrect order.

Example. Figure 3.3 shows a pull request [3] of Ansible repository. This fault influences the sequence of tasks by not checking for the next nested task in a certain execution strategy before marking the task as complete. This results in the nested loop tasks being missing from the task sequence; thus, the executor never executes those nested loop tasks.

[2] https://github.com/ansible/ansible/pull/29663/files
[3] https://github.com/ansible/ansible/pull/33094/files

```python
if block._eor and host.name in block._role._had_task_run and not in_child:
if block._eor and host.name in block._role._had_task_run and not in_child and not peek:
        block._role._completed[host.name] = True
```

Figure 3.3: An example of incorrect task iteration logic fault

```python
if 'unreachable' in item and item['unreachable']:
    item_ignore_unreachable = item.pop('_ansible_ignore_unreachable')
    if not res.get('unreachable'):
        res['unreachable'] = True
        self._task.ignore_unreachable = item_ignore_unreachable
    elif self._task.ignore_unreachable and not item_ignore_unreachable:
        self._task.ignore_unreachable = item_ignore_unreachable
```

Figure 3.4: An example of state mismanagement fault

3.1.4 State Mismanagement

In Ansible, tasks and hosts hold a lot of information including their status, flag values, and error messages. This information is referred to as the state of a task/host.

This category of faults occurs when the state of a task or host is incorrectly modified or misrepresented by the executor, leading to the task or host holding inaccurate information. When this information is manipulated incorrectly, it can lead to misleading reports, improper handling of follow-up tasks, or failures in the playbook's logic.

Example. Figure 3.4 shows a pull request [4] of Ansible repository. This fault is caused by the mismanagement of the variable ignore_unreachable of task states within a nested loop list. This fault causes the task_executor to incorrectly ignore an unreachable error that should not be ignored within a loop.

3.1.5 Incorrect Detection/Analysis of User Input

In Ansible, the task and host state can be changed through user input during execution. Handling the user input incorrectly can cause unexpected errors/exceptions or hosts/tasks holding incorrect values. In the context of Ansible, faults caused by incorrect detection or analysis of

[4] https://github.com/ansible/ansible/pull/84049/files

```
all_vars = self._variable_manager.get_vars(play=play)
templar = Templar(loader=self._loader, variables=all_vars)
new_play = play.copy()
new_play.post_validate(templar)

if play.vars_prompt:
    for var in play.vars_prompt:
    for var in new_play.vars_prompt:
        vname = var['name']
        prompt = var.get("prompt", vname)
        default = var.get("default", None)
```

Figure 3.5: An example of incorrect detection/analysis of user input fault

```
if C.USE_PERSISTENT_CONNECTIONS or getattr(self,
    'FORCE_PERSISTENT_CONNECTION', False):
    conn_type == 'persistent'
    conn_type = 'persistent'
```

Figure 3.6: An example of typo fault

user input typically refer to issues where the system fails to properly interpret or process the input provided by the user (e.g., through playbooks, roles, variables, or commands).

Example. Figure 3.5 shows a pull request [5] of Ansible repository. The root cause of this fault is that Ansible fails to validate variables from playbooks before user prompts are processed. This fault prevents users from successfully using predefined variables from playbooks in the prompts.

3.1.6 Typo

A typographical error (typo) refers to mistakes in variable names, function names, keywords, or syntax that lead to unintended behavior or execution failures. These errors often result from misspellings, incorrect casing, or misplaced characters in the code. The commit messages of the fixed code of this category often specify that the fault is caused by typos.

Example. Figure 3.6 shows a pull request [6] of Ansible repository. The root cause of this fault is that when setting the connection type, the developer mistyped = as ==. This prevents the _get_connection function from properly assigning the value to conn_type, causing the host to use the wrong type of connection.

[5] https://github.com/ansible/ansible/pull/32802/files
[6] https://github.com/ansible/ansible/pull/24697/files

3.2 Consequences

After defining the categories of faults in Ansible's Executor component, I analyze the issue reports and commit messages of the pull requests to find out what kind of consequences can be caused by each execution fault. I identify 4 consequences.

- **Crashes**: Occurs when Python crashes unexpectedly with an error message.
- **Faulty Completion**: Happens when an Ansible playbook or role runs its tasks to completion, but the desired outcome is not achieved.
- **Silent Partial Completion**: Describes cases where tasks fail to complete but do not trigger an error or failure report.
- **Reported Partial Completion**: Refers to situations where tasks fail to complete and an explicit error or failure report is generated.

These classifications help in understanding different failure modes in Ansible execution and assessing the impact of faults such as the one identified in this study. Table 3.2 shows the correlation between the fault categories and their consequences.

3.3 Applicability of Formal Verification for the Executor Component

While faults can arise from various sources, not all categories are suitable for detection through theorem proving. The focus of this thesis is on identifying new faults using formal verification. Faults that fall outside this scope, such as dependency-related faults, typo, and variable misreference, are excluded from consideration because they cannot be effectively modeled or analyzed within a formal proof system.

Not all software faults can be detected through theorem proving. Specifically, dependency-related faults and typos fall outside the scope of theorem proving. Theorem proving operates on a formal model, meaning that the proof system verifies only the properties explicitly encoded within that model [26]. Consequently, if external dependencies—such as libraries, configuration files, or runtime environments—are not incorporated into the formalization, any issues stemming from them will remain undetected.

Moreover, dynamically loaded dependencies pose a particular challenge, as their interactions and runtime behaviors cannot always be adequately captured within a static logic system. Similarly, theorem provers function on syntactically valid input, assuming that code has already undergone basic syntax validation. As a result, if a typo renders a term syntactically invalid, the proof system may reject the code outright rather than identifying the mistake as a fault. Additionally, if a typo results in an unintended yet syntactically valid identifier (e.g., a mistyped variable or function name that coincidentally exists within scope), the theorem prover may verify properties of an unintended entity, thereby failing to detect the underlying error.

On the other hand, variable misreferences are generally detected at an earlier stage by the compiler or interpreter in a strongly typed theorem-proving language that enforces exhaustive pattern matching. This detection occurs through mechanisms such as scope resolution, type checking, and pattern-matching rules, which ensure that variables are correctly referenced before the proof process begins.

Given these limitations, my focus is on faults that are amenable to formal verification through theorem proving. This interest stems from the objective of evaluating the feasibility of using theorem proving to verify Ansible. Consequently, fault categories that fall outside the scope of theorem proving is excluded.

Table 3.2: Fault categories and their corresponding consequences.

Category	**Crashes**	**Faulty Completion**	**Silent Partial Completion**	**Reported Partial Completion**
Dependency-related Faults	7	0	1	1
Variable Misreference	1	2	1	2
Incorrect Task Iteration Logic	0	0	2	0
State Mismanagement	1	1	4	2
Incorrect Detection/Analysis of Input	0	2	0	5
Typos	1	0	0	0
Total	10	5	8	10

Chapter 4

RQ2: Formally Verifying the Executor Component

This section answers "*RQ2: How can one use identified fault categories to verify the Ansible execution component?*" The question is answered with a step-by-step methodology, an example case, properties identified and faults detected/reproduced.

4.1 Methodology

To find faults by verifying the executor of the Ansible compiler with a specific property, I divide the methodology into 7 steps. In this research, I use Ansible version 2.17.4 [27]. First, properties to be verified are identified. Then, for each identified property, formal verification is performed using the following steps.

In the context of theorem proving, a **property** refers to a formally specified condition or behavior that a system, function, or algorithm is expected to satisfy [28]. It serves as a *logical specification* against which correctness can be rigorously verified.

Step 1-Issues Analyzing and Property Identification. In this step, I determine the expected behavior related to an execution fault by examining the following sources:

- **Issue report**, which outlines both the expected and the actual faulty behavior.
- **Pull request**, which shows code changes that clarify how the function is intended to behave.
- **Ansible Documentation**, which is used to resolve any ambiguities in understanding the feature or behavior.

Once the expected behavior is identified, it must be translated into a formally verifiable property using mathematical logic.

Step 2-Defining the Scope of Verification. To ensure a rigorous verification process, the precise scope of verification must be defined first. This step establishes the boundaries of analysis by focusing on the specific components and behaviors which identified properties can be applied to.

Step 3-Reimplement component in Rocq. To verify a property in Rocq on Ansible components, the components must first be reimplemented within Rocq. Rocq allows one to formally specify lemmas, and then mechanically verify the correctness of proofs concerning these properties [29]. In this step, data structures and behaviors are modeled into Rocq's formal syntax. Types and constructs are defined to represent the core behaviors of the system, ensuring that the reimplementation closely mirrors the original system's logic. This process involves creating types, functions, and records to represent key aspects of the system. The model I provide here is not a direct implementation of Rcoq; rather, it is an abstraction of the model.

The `T` record type shown in Figure 4.1 can be derived by analyzing `TaskResult` class in Ansible source code, particularly within `executor/task_result.py`, where the class defines attributes representing task execution outcomes.

To enable the executor components to access templated variables, `Templar` is defined and represented as `Tem`. `Tem` represents a mapping of variable names to their corresponding values in the execution of Ansible tasks. The model of `Tem` is shown in Figure 4.2.

$$\mathrm{T} \triangleq \begin{pmatrix} \text{host} & : \mathbb{S} \\ \text{task} & : \mathbb{S} \\ \text{changed} & : \mathbb{B}^{?} \\ \text{skipped} & : \mathbb{B}^{?} \\ \text{failed} & : \mathbb{B}^{?} \\ \text{\_ansible\_ignore\_errors} & : \mathbb{B}^{?} \\ \text{unreachable} & : \mathbb{B}^{?} \\ \text{\_ansible\_ignore\_unreachable} & : \mathbb{B}^{?} \end{pmatrix}$$

Figure 4.1: Task Result Model

$$\mathbb{S} \text{ is the set of finite strings.} \tag{4.1}$$

$$\mathbb{B} = \{\text{true}, \text{false}\} \text{ is the Boolean set.} \tag{4.2}$$

$$\mathbb{B}^? = \mathbb{B} \cup \{\text{None}\} \text{ represents an optional Boolean value.} \tag{4.3}$$

$$\text{Tem} \triangleq \{(k, v) \mid k \in K, v \in D\}$$

Figure 4.2: Templar Model

$$K = \{s \mid s \text{ is a finite string}\} \tag{4.4}$$

$$D = \{s \mid s \text{ is a template value}\} \tag{4.5}$$

Template look up function:

$$\forall k \in K, \forall \mathcal{T} \in \text{Tem}, \quad \text{lookup}(k, \mathcal{T}) = \begin{cases} v, & \text{if } (k, v) \in \mathcal{T} \\ \bot, & \text{otherwise} \end{cases} \tag{4.6}$$

Recursive Aggregation Function:

$$\forall t, R_0 \in T, \forall \mathcal{T} \in \text{Tem}, A(t, R_0, \mathcal{T}) \in T \tag{4.7}$$

Step 4-Implement Property in Rocq. This step involves writing the identified properties in the form of lemmas, which are logical assertions that the system should satisfy under specific conditions in Rocq. The lemmas act as formal descriptions of the expected properties and behavior of the analyzed Ansible components. Lemmas are formulated from mathematical properties with the behavior and formal model formed in the previous step.

Step 5-Verify Implemented Properties. In this step, I focus on verifying the correctness of the lemmas defined in Step 4, using the Rocq proof assistant. The goal is to formally establish that the Ansible components, as reimplemented in Rocq, satisfy the lemmas specified earlier. This verification process ensures that the formal model of the Ansible components behaves as expected under all possible conditions, as outlined in the associated properties.

Upon successful verification, it can be concluded that the reimplemented Ansible components satisfy the formal properties, providing a high degree of confidence in their correctness.

These verified properties serve as a strong formal guarantee that the components behave as expected, based on the assumptions and specifications. On the other hand, any verification failures point to deviations from the expected behavior, indicating the presence of potential faults that warrant further investigation.

In the event of a failed proof, the discrepancy between the formal model and the expected behavior is analyzed. This could be due to an incorrect assumption, an incomplete model of the Ansible component, or an error in the property definition. When a lemma cannot be proven, the current goal is explored to identify a potential counterexample. I then try to show how the lemma is unsatisfiable using the counterexample. Once such cases are found, they need to be analyzed to determine if they fall within the scope of the original property corresponding to the lemma. If it is not, the lemma needs to be refined to ensure that the lemma is correctly formulated. If the lemma is correctly formulated and such a counterexample arises from the implementation, the original Python source code is then analyzed to check if this example exists in the code. If such an example can be computed using the source code, a potential fault is found. The identification of a failed proof is significant, as it suggests the presence of potential faults in the Ansible components. Components that violate properties have potential faults since they behave differently from the properties deduced from issue reports, pull requests, and the Ansible specification. By finding components that hold potential faults, these potential faults can be authenticated in the following step.

Step 6-Dynamic Analysis for Fault Detection. After identifying a property violation through formal verification, I proceed to reproduce the potential fault in Ansible using a script that mirrors the scenario where the issue was detected. First, the Ansible version 2.17.4 [27] source code package is downloaded to ensure consistency with the model. Then, on a server maintained by the university with Ubuntu 22.04.5, I try to reproduce the fault. The execution of the script is carefully observed to confirm whether the fault identified during formal verification manifests in the actual system. If the fault is reproducible, it validates that the formal verification successfully identified a real-world issue within Ansible. This step provides practical confirmation of the fault, bridging the gap between theoretical analysis and actual system behavior.

Step 7-Practitioner Feedback on Faults. Once a reproducible fault has been identified, a formal issue report can be drafted and submitted to the Ansible repository. This report should comprehensively document the issue, including a clear comparison of the expected and actual behavior. Additionally, the report must include the specific input playbook utilized to trigger the fault, ensuring that the problem can be reliably reproduced and analyzed by the development team. The issue report should also include environment details, including the version of Ansible and operating system. After submitting the issue, it is important to monitor the discussion and respond to any inquiries from the maintainers or contributors. Providing additional information or testing potential fixes when requested can help expedite the resolution process. By following this systematic approach, the fault can be effectively communicated and addressed within the Ansible development community.

4.2 Running Example

This section showcase the process step by step using the one new fault detected as an example. Figure 4.3 shows an overview of the running example.

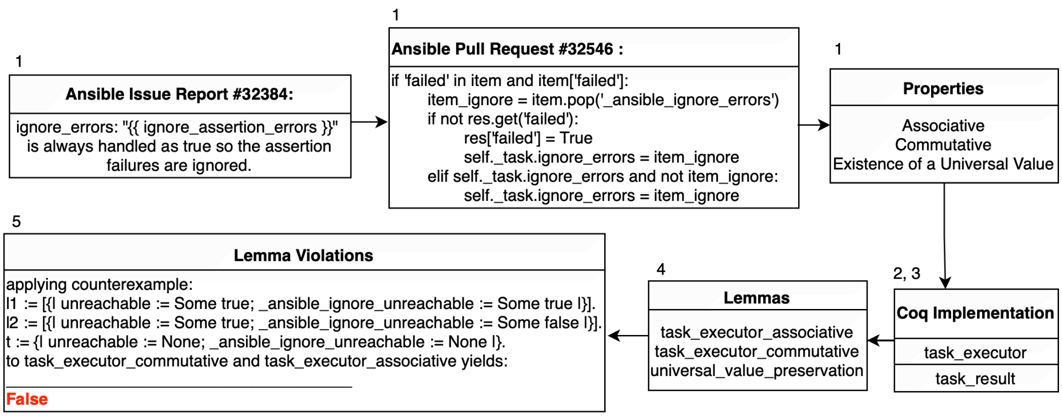

Figure 4.3: This diagram provides an overview of the running example. By examining the issue report and pull request related to ignore_errors, three key properties are identified. These properties are then applied to ignore_unreachable_errors by translating the relevant components into Rocq and expressing the properties as lemmas. Through formal verification in Rocq, violations of the associative and commutative property are detected, with a counterexample demonstrating a resulting contradiction.

Step 1–Issues Analyzing and Property Identification. I use one of the fault reports [1] obtained from the Ansible repository to demonstrate my methodology. The fault report presents

[1]https://github.com/ansible/ansible/issues/32384

an example of state mismanagement. The fault occurs because of mismanagement of the variable `ignore_unreachable` of task results within a nested loop task. In Ansible, a task result refers to the output or response generated when a task is executed on a target host. Each task execution produces a structured dictionary that contains various fields indicating the execution status, results, and flags. `ignore_errors` is a flag that is used to determine whether a failed task should be ignored or cause the playbook execution to stop. Specifically, when `ignore_errors` is set to `false`, the task should fail and prevent execution from proceeding to the next task. However, when `ignore_errors` is assigned for each task under a loop, it is incorrectly evaluated as `true`, causing assertion failures to be ignored. As a result, the first task does not halt execution upon failure as expected, allowing the subsequent task to execute. Since this is not the expected behavior when `ignore_errors` is set to false, a violation of the property is identified.

After investigating the source code from the file changed in the pull request [2], it is concluded that the root cause of the fault is that the `run` function does not aggregate the value of `ignore_errors` at all. This leads to the task loop lacking the value of `ignore_errors`. Figure 4.4 shows the aggregating function `run`. From the corrected source code, the aggregated result will be `true` if and only if all failed task explicitly sets `ignore_errors` to `true`. This means that the sequence and the grouping of the nested task should not influence the aggregated `ignore_errors`; hence, the aggregate function should possess both the commutative property and the associative property. Specifically, among all failed tasks, the `ignore_errors` attribute should be set to `false` if at least one of these tasks has `ignore_errors` explicitly set to `false`. Furthermore, the default value for `ignore_errors` should also be `false`, ensuring consistency in error handling behavior.

In the context of the issue where the `ignore_errors` directive is mismanaged, let:

- T represent `TaskResult`, which contains the task attributes. T is a set of task results.
- $\mathcal{N}(t)$ represent `item_results` which is a nested task structure (a list of T).

[2] https://github.com/ansible/ansible/pull/32546/files

```python
def run(self):
...
    for item in item_results:
        ...
        if 'failed' in item and item['failed'] and not res.get('failed'):
            res['failed'] = True
            res['msg'] = 'One or more items failed'
        if 'failed' in item and item['failed']:
            item_ignore = item.pop('_ansible_ignore_errors')
            if not res.get('failed'):
                res['failed'] = True
                res['msg'] = 'One or more items failed'
                self._task.ignore_errors = item_ignore
            elif self._task.ignore_errors and not item_ignore:
                self._task.ignore_errors = item_ignore
```

Figure 4.4: Aggregating function run

- A represent `run` which is an aggregation function that computes the universal value of a nested task structure $\mathcal{N}(t)$. The computed values include `ignore_errors` flag.

For any nested task structure $\mathcal{N}(T)$ and aggregation function A, the following conditions hold:

1. **Commutative:**

$$\forall t_1, t_2 \in T, \quad A(t_1, t_2) = A(t_2, t_1). \tag{4.8}$$

2. **Associative:**

$$\forall t_1, t_2, t_3 \in T, \quad A(A(t_1, t_2), t_3) = A(t_1, A(t_2, t_3)). \tag{4.9}$$

3. **Existence of a Universal Value:**

$$\forall \mathcal{N} \in \mathcal{N}(T), \quad \exists v \in V, \quad U(A(\mathcal{N})) = v. \tag{4.10}$$

Step 2-Defining the Scope of Verification. After analyzing the issue report, the scope of this analysis can be clearly defined. Since the issue analyzed directly affects task_executor, task_executor is selected as the component where the property will be verified. The verification process began by first deciding to focus on analyzing state mismanagement of ignore_unreachable under task_executor. Once the scope was identified, the code block activated

```python
elif task_result.is_unreachable():
    ignore_unreachable = original_task.ignore_unreachable
    if not ignore_unreachable:
        self._tqm._unreachable_hosts[original_host.name] = True

         ↪  iterator._play._removed_hosts.append(original_host.name)
        self._tqm._stats.increment('dark', original_host.name)
    else:
        self._tqm._stats.increment('ok', original_host.name)
        self._tqm._stats.increment('ignored',
         ↪  original_host.name)
```

Figure 4.5: Python implementation of ignoring unreachable host

by ignore_unreachable is selected as the specific component to model. This decision was crucial as it provided a clear context for understanding how mismanagement of task states could arise in the Ansible Executor component, particularly when handling unreachable hosts. This flag is chosen because of its significant impact on task execution behavior, especially within nested loops. The ignore_unreachable option in Ansible allows playbooks to continue execution on hosts that become unreachable instead of failing and removing them from further tasks. When enabled, unreachable hosts remain in execution, logging the failure but proceeding with subsequent tasks as shown in Figure 4.5. However, an incorrect aggregate function can result in a mismanagement of ignore_unreachable which can lead to silent partial completion.

Step 3–Reimplement component in Rocq. To formally verify the behavior of the Ansible Executor component, the relevant components are first reimplemented in Rocq. Specifically, ignore_unreachable within task result is modeled, ensuring that the reimplementation captured the behavior of task execution when encountering unreachable hosts. According to the Ansible source code in Figure 4.6, the implementation is constructed by defining a record type for task result, along with a recursive function task_executor that processes task execution over a list of items. This function represents the behavior of the run function of the task execution when handling unreachable errors in a loop. The functions defined are presented in Figure 4.7.

Step 4-Implement Property in Rocq. To verify the correctness of the ignore_unreachable flag management, three lemmas are formulated in Rocq according to the identified properties. The properties described above relate to how a nested task structure's value is computed using an aggregation function. In the context of `ignore_unreachable`, these properties could

```python
def run(self):
...
    for item in item_results:
    ...
        if 'unreachable' in item and item['unreachable']:
            item_ignore_unreachable = item.pop('_ansible_ignore_unreachable')
            if not res.get('unreachable'):
                self._task.ignore_unreachable = item_ignore_unreachable
            elif self._task.ignore_unreachable and not item_ignore_unreachable:
                self._task.ignore_unreachable = item_ignore_unreachable
```

Figure 4.6: Python implementation of task_executor and update_ignore_unreachable

```
Definition update_ignore_unreachable (res : task_result) (item_ignore_unreachable : bool) :=
  match res.(unreachable) with
  | Some true  =>
      match (res.(_ansible_ignore_unreachable), item_ignore_unreachable) with
      | (Some true, false) => res <|_ansible_ignore_unreachable := Some item_ignore_unreachable|>
      | _ => res
      end
  | _ =>
      res
          <|_ansible_ignore_unreachable := Some item_ignore_unreachable|>
  end.

Fixpoint task_executor (items : list task_result) (res : task_result): task_result :=
  match items with
  | item :: xs => match item.(unreachable) with
    | Some true => task_executor xs (update_ignore_unreachable res (get_option item.(_ansible_ignore_unreachable)))
    | _ => task_executor xs res
    end
  | [] => res
end.
```

Figure 4.7: Rocq implementation of task_executor and update_ignore_unreachable

describe how the `ignore_unreachable` variable influences the computation of the 'universal value' (e.g., the overall result or state) of a task or group of tasks. Here's how each property would relate to the behavior of `ignore_unreachable`:

1. **Commutative:** The commutative property means that the aggregation of two tasks' values does not depend on the order of the tasks. In the context of `ignore_unreachable`, this suggests that the handling of unreachable tasks (whether errors are ignored or not) should not affect the overall result of a sequence of tasks, no matter the order in which they are executed. This implies that `ignore_unreachable` should be applied uniformly across tasks, regardless of their sequence. Figure 4.8 shows that the Lemma corresponds to the commutative property.

2. **Associative:** The associative property means that the way tasks are grouped and aggregated does not change the result. This is important when tasks are nested, as the structure of the tasks may change but the overall outcome should remain consistent. For `ignore_unreachable`, this means that if an unreachable task is ignored, it should not affect the aggregation of the result in a way that depends on whether the task is grouped with others in a specific way. The handling of `ignore_unreachable` should be independent of how tasks are grouped within the hierarchy. Figure 4.9 shows the Lemma corresponds to the associative property.

3. **Existence of a Universal Value:** This property guarantees that for any given nested task structure, there exists a universal value that aggregates all the individual task results. When applied to `ignore_unreachable`, this means that if some tasks are unreachable regardless of if the errors are ignored, the entire nested task structure should still yield a final result for `ignore_unreachable`. Figure 4.10 shows the Lemma corresponds to Existence of a Universal Value.

These properties state that the task execution result should remain consistent regardless of the order in which tasks are processed.

Step 5-Verify Implemented Properties. The property is then formally verified using the Rocq proof assistant. The proof requires reasoning about how task_executor processes lists of task results and updates the ignore_unreachable of the universal task result. During this step, I attempt to prove the all three lemmas using induction. Here, I will focus on lemma task_executor_commutative, which states that the final execution result remains unchanged when task items are reordered. During proof construction, I encountered a discrepancy under the following condition:

Given:

- $l_1 = [\{\text{unreachable} = \text{Some true}, \quad \text{ansible\_ignore\_unreachable} = \text{Some true}\}]$
- $l_2 = [\{\text{unreachable} = \text{Some true}, \quad \text{ansible\_ignore\_unreachable} = \text{Some false}\}]$
- $t = \{\text{unreachable} = \text{None}, \quad \text{ansible\_ignore\_unreachable} = \text{None}\}$

Since ignore_unreachable has type option bool, the possible values for _ansible_ignore_unreachable of l_1 and l_2 are None, Some true and Some false. Upon inductive proof, it is shown that if l_1 is aggregated first, _ansible_ignore_unreachable = Some false is yielded. However, if l_2 first is aggregated first, _ansible_ignore_unreachable = Some true is yielded.

This situation provides a counterexample to the commutative property. In an operation where tasks are aggregated or combined, the order in which the tasks are processed affects the outcome, specifically the final value of _ansible_ignore_unreachable. This violates the assumption of the commutative property, which would expect that the result of combining l_1 and l_2 should be the same regardless of the order. Thus, this demonstrates that task execution with certain conditions (like `ignore_unreachable`) does not always commute, which could lead to unexpected behavior when tasks are processed in varying sequences.

Step 6-Dynamic Analysis for Fault Detection. To confirm the presence of the fault in Ansible's implementation, I manually constructed an Ansible script that replicated the counterexample found through exploring the proving space. Figure 4.11 is the playbook that reproduce the execution fault, the playbook has 2 nested loops with three unreachable tasks each. Both loops have one task with ignore_unreachable set to true and others set to false. These tasks are identical to each other except for the order in the loop. Running the script on the Ansible version 2.17.4 [27] shows that the first loop ignored the unreachable error, but the second did

```
Theorem task_executor_commutative :
  forall (l1 l2 : list task_result) (t : task_result),
    task_executor l1 (task_executor l2 t) = task_executor l2
    ↪  (task_executor l1 t).
```

Figure 4.8: Rocq Lemma task_executor_commutative

```
Theorem task_executor_associative :
  forall (l1 l2 : list task_result) (t : task_result),
    task_executor l1 (task_executor l2 t) = task_executor (l1
    ↪  ++ l2) t.
```

Figure 4.9: Rocq Lemma task_executor_associative

not. With this example script, I successfully reproduced the issue: the task_executor incorrectly ignored unreachable errors in certain loop conditions. This confirmed that the Rocq-based verification effectively identified a real fault in the Ansible Executor component.

Step 7-Practitioner Feedback on Faults. I then proceed to report this issue with the Figure 4.11. The issue I reported [3] showcases the expected behavior, actual behavior, environment setup and example input and output I use to reproduce the fault.

4.3 Answer to RQ2

4.3.1 Identified properties

Table 4.1 shows the properties identified in each of the fault categories identified in Step 1. The identified properties are written in the model defined in Step 3.

Commutative

$$\forall t_1, t_2 \in T, \quad A(t_1, t_2) = A(t_2, t_1). \tag{4.11}$$

[3] https://github.com/ansible/ansible/issues/84019

```
Theorem universal_value_preservation :
  forall (items1 items2 : list task_result) (un : option bool),
    let item := {| unreachable := Some true;
    ↪  _ansible_ignore_unreachable := un |} in
    let res := {| unreachable := None;
    ↪  _ansible_ignore_unreachable := None |} in
    let final_result := task_executor (items1 ++ [item] ++
    ↪  items2) res in
    exists v, final_result.(_ansible_ignore_unreachable) = Some
    ↪  v.
```

Figure 4.10: Rocq Lemma universal_value_preservation

```yaml
- hosts: localhost
  gather_facts: false
  become: false
  vars:
    ansible_connection: ssh
    ansible_ssh_user: "non-existent-user"
  tasks:
    - name: unreachable task loop where the last item has ignore_unreachable set to true
      action: ping
      ignore_unreachable: "{{item.unreachable}}"
      loop:
      - { name: 'testuser1', unreachable: false }
      - { name: 'testuser2', unreachable: false }
      - { name: 'testuser3', unreachable: true }

    - name: unreachable task loop where some item has ignore_unreachable set to true while the last item has ignore_unreachable set to false
      action: ping
      ignore_unreachable: "{{item.unreachable}}"
      loop:
      - { name: 'testuser1', unreachable: false }
      - { name: 'testuser3', unreachable: true }
      - { name: 'testuser2', unreachable: false }
```

Figure 4.11: Playbook that showcases violation of the commutative property

Explanation: The commutative property ensures that the execution of tasks is independent of their order, meaning that the result of executing two tasks is the same regardless of the order in which they are executed. For any two tasks t_1 and $t_2 \in T$, the execution of t_1 followed by t_2 is equivalent to executing t_2 followed by t_1.

Associative

$$\forall t_1, t_2, t_3 \in T, \quad A(A(t_1, t_2), t_3) = A(t_1, A(t_2, t_3)). \tag{4.12}$$

Explanation: The associative property ensures that the grouping of tasks during execution does not affect the final result. For any three tasks $t_1, t_2, t_3 \in T$, the result of executing t_1 and t_2 first, followed by t_3, is equivalent to executing t_2 and t_3 first, followed by t_1.

Existence of a Universal Value

$$\forall \mathcal{N} \in \mathcal{N}(T), \quad \exists v \in V, \quad A(\mathcal{N}) = v. \tag{4.13}$$

Explanation: The existence of a universal value property ensures that for any task node, there exists a universal value that represents the final result of executing that task. For every task node $\mathcal{N} \in \mathcal{N}(T)$, there exists a value $v \in V$ (where V is the set of possible universal values) such that applying the universal function U to the result of executing $\mathcal{N}$ produces v.

Expansion of Templated Variables

$$\forall k \in K, \forall \mathcal{T} \in \text{Tem}, \quad (\exists v \in D,\ \text{lookup}(k, \mathcal{T}) = v) \tag{4.14}$$

Explanation: This property ensures that if a variable name k exists in the template $\mathcal{T}$, then looking it up will return some corresponding value v from the domain D. This guarantees that templated variables expand correctly when referenced, meaning that all variables in the template must be well-defined and retrievable.

Well-Defined Template Evaluation

$$\forall \mathcal{N} \in \mathcal{N}(T), \forall \mathcal{T} \in \text{Tem}, \quad A(\mathcal{N}, \mathcal{T}) \in D \tag{4.15}$$

Explanation: This property asserts that applying a template $\mathcal{T}$ to a task representation $\mathcal{N}$ produces a well-defined value within the domain D. Essentially, it ensures that any task that depends on templated variables can be correctly evaluated to yield meaningful results, preventing undefined behavior or errors due to missing variables.

Block Completion

$$\begin{aligned}
&\forall s \in S, b \in B, h \in H, \\
&\qquad \begin{pmatrix}
\forall t \in T, \quad \text{cur_task}(s, b, t) \geq \text{length}(b) \\
\land \quad \forall b' \in B, \quad \text{child_state}(s, b') = \emptyset \\
\land \quad \text{run_state}(s) \in \mathcal{C} \\
\land \quad \text{role}(b) \neq \emptyset
\end{pmatrix} \\
&\qquad\qquad \Rightarrow \text{completed}(\text{role}(b), h) = \text{true}
\end{aligned} \tag{4.16}$$

Explanation: This property describes the conditions under which a block b is considered completed for a given host h. It states that:

- The current task index in the block has reached or exceeded the block length, meaning all tasks in the block have executed.
- There are no remaining child states in execution, ensuring that no dependent tasks are still running.
- The run state of the system is within a valid set $\mathcal{C}$, indicating that execution is proceeding correctly.
- The block belongs to a defined role.

If all these conditions hold, then the block's role execution for host h is marked as completed. This ensures that Ansible's execution model correctly identifies completed task groups before proceeding.

These properties collectively help verify the correctness of the aggregation of tasks, templated task execution, and block completion in Ansible's execution framework.

Table 4.1: Answer to RQ2: Identified properties

Category	Property	Lemma
Incorrect Task Iteration Logic	Block Completion	$\forall s \in S, \forall b \in B, \forall h \in H, \begin{pmatrix} \forall t \in T, \quad \text{cur\_task}(s,b,t) \geq \text{length}(b) \\ \wedge \quad \forall b' \in B, \quad \text{child\_state}(s,b') = \emptyset \\ \wedge \quad \text{run\_state}(s) \in \mathcal{C} \\ \wedge \quad \text{role}(b) \neq \emptyset \end{pmatrix} \Rightarrow \text{completed}(\text{role}(b), h) = \text{true}$
State mismanagement	Commutative	$\forall t_1, t_2 \in T, \quad A(t_1,t_2) = A(t_2,t_1)$
	Associative	$\forall t_1, t_2, t_3 \in T, \quad A(A(t_1,t_2),t_3) = A(t_1, A(t_2,t_3))$
	Existence of a Universal Value	$\forall \mathcal{N} \in \mathcal{N}(T), \quad \exists v \in V, \quad A(\mathcal{N}) = v$
Incorrect Detection/Analysis of User Input	Expansion of Templated Variables	$\forall k \in K, \forall \mathcal{T} \in \text{Tem}, \quad (\exists v \in D, \ \text{lookup}(k, \mathcal{T}) = v)$
	Well-Defined Template Evaluation	$\forall \mathcal{N} \in \mathcal{N}(T), \forall \mathcal{T} \in \text{Tem}, \quad A(\mathcal{N}, \mathcal{T}) \in D$

4.3.2 Faults Reproduced/Detected

Table 4.2 shows the faults reproduced or detected using my approach. In all, two faults are identified using formal verification. Initially, I assume that both identified faults are new, but upon submission of the fault reports, one of the faults is old and resolved separately. Therefore, I identify one new fault, as indicated by the green cell, which is accepted as valid by the Ansible community.

Table 4.2: Answer to RQ2: Faults identified using Rocq

Category	Violated Properties	Faults Found	Faults Confirmed	URLs
Incorrect Task Iteration Logic	-	0	0	
State Mismanagement	Commutative and Associative	1	1	link[4]
Incorrect Detection/-Analysis of User Input	Well-Defined Template Evaluation	1	1	link[5]
Total	-	2	2	-

[4]https://github.com/ansible/ansible/issues/84019
[5]https://github.com/ansible/ansible/issues/84018

Chapter 5

Discussion

I discuss the implications and limitations of my thesis as follows:

Implication for Practitioners: My research provides a structured approach for identifying and classifying faults in the Ansible Executor component, helping contributors focus on the most critical issues. By formalizing key properties derived from fault categories, developers are enabled to verify expected behavior rigorously, reducing the likelihood of regressions. The use of theorem proving ensures that faults are detected with mathematical certainty, offering a higher degree of confidence compared to traditional testing methods. This approach not only aids in debugging but also improves the maintainability of the Ansible codebase by enforcing logical correctness. Ultimately, the contributions of this study support Ansible developers in running automation scripts more reliably, leading to a more robust and predictable infrastructure management tool.

Implication for Researchers: Researchers can build upon this work by extending the fault classification to cover additional Ansible components beyond the Executor, creating a comprehensive taxonomy of potential issues. By leveraging automated theorem proving techniques, they can scale formal verification to larger portions of the Ansible codebase, ensuring correctness across diverse configurations. Additionally, researchers can develop automated tools that generate formal specifications directly from Ansible playbooks, making formal verification more accessible to practitioners.

Chapter 6

Threats to Validity

I discuss the limitations of my thesis as follows:

- *Conclusion Validity*: The derived fault categories and consequences are limited to the fault reports analyzed. The approach of deriving properties for formal verification is also limited to the issue reports collected and analyzed. As such, I may have missed faults that are not captured in the collected issue reports. Also, some typo-related execution faults can be classified into other categories. This leads to the categories not being mutually exclusive.
- *External Validity*: The analyzed faults are limited to one configuration management tool, i.e., Ansible. The results of this study may not generalize to other configuration management tools, such as Chef and Puppet.
- *Construct Validity*: 17 issue reports related to the executor component of Ansible are used. This collection of issue reports may not be comprehensive. Also, I act as the only rater to derive fault and consequence categories. This process is subject to rater bias, which is mitigated using rater verification.

Chapter 7

Conclusions

While Ansible aids in software deployment and automated configuration management, faults can occur during the execution of Ansible configuration scripts. To get a better understanding of these faults, I have conducted an empirical study with 33 faults that have occurred for the executor component in Ansible. From the analyzed execution faults, 6 fault categories and 4 fault consequences are identified. Furthermore, by analyzing the issue reports, I derive properties to conduct formal verification. The formal verification approach yielded one old fault and one new fault in the Ansible executor component. Based on my findings, I recommend practitioners recognize critical properties from historical pull requests and issue reports to verify expected behavior. Furthermore, they can incorporate formal verification techniques, such as theorem proving, to validate critical properties in Ansible, ensuring correctness in task execution.

References

[1] M. Guerriero, M. Garriga, D. A. Tamburri, and F. Palomba, “Adoption, support, and challenges of infrastructure-as-code: Insights from industry,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 580–589.

[2] D. Sokolowski and G. Salvaneschi, “Towards reliable infrastructure as code,” in *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, 2023, pp. 318–321.

[3] Y. Zhang, M. Rahman, F. Wu, and A. A. U. Rahman, “Quality assurance for infrastructure orchestrators: Emerging results from ansible,” *2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C)*, pp. 1–3, 2023. [Online]. Available: https://api.semanticscholar.org/CorpusID:258329904

[4] M. M. Hassan, J. Salvador, S. K. K. Santu, and A. Rahman, “State reconciliation defects in infrastructure as code,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660790

[5] F. I. Katsuhiko Ikeshita and S. Honiden, “Test suite reduction in idempotence testing of infrastructure as code,” in *Tests and Proofs*, S. Gabmeyer and E. B. Johnsen, Eds. Springer International Publishing, pp. 98–115.

[6] A. Rahman, E. Farhana, C. Parnin, and L. Williams, “Gang of eight: a defect taxonomy for infrastructure as code scripts,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 752–764. [Online]. Available: https://doi.org/10.1145/3377811.3380409

[7] M. D. Pascalis, "Formal verification of infrastructure as code."

[8] (2025) Ansible community documentation - ansible module architecture. [Accessed 14-03-2025]. [Online]. Available: https://docs.ansible.com/ansible/latest/dev_guide/developing_program_flow_modules.html

[9] "ansible/ansible," Mar. 2025, original-date: 2012-03-06T14:58:02Z. [Online]. Available: https://github.com/ansible/ansible

[10] (2025) Rocq. [Accessed 14-03-2025]. [Online]. Available: https://rocq-prover.org

[11] W. S. Cheng, "execution_faults_in_Ansible," 3 2025. [Online]. Available: https://figshare.com/articles/conference_contribution/execution_faults_in_Ansible/28588733

[12] (2025) Ansible community documentation. [Online]. Available: https://docs.ansible.com

[13] (2025) Ansible community documentation - ansible playbooks. [Online]. Available: https://docs.ansible.com/ansible/latest/playbook_guide/playbooks_intro.html

[14] P. Bjesse, "What is formal verification?" vol. 35, no. 24, pp. 1–es, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/1113792.1113794

[15] O. Hasan and S. Tahar, *Formalized Probability Theory and Applications Using Theorem Proving*, 1st ed. IGI Global.

[16] E. M. Clarke and J. M. Wing, "Formal methods: state of the art and future directions," vol. 28, no. 4, pp. 626–643, place: New York, NY, USA Publisher: Association for Computing Machinery. [Online]. Available: https://doi.org/10.1145/242223.242257

[17] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. U. Lali, "A survey on theorem provers in formal methods," 2019. [Online]. Available: https://arxiv.org/abs/1912.03028

[18] (2025) Puppet infrastructure it at scale. [Online]. Available: https://www.puppet.com

[19] (2025) Chef software devops automation solutions. [Online]. Available: https://www.chef.io

[20] (2025) Ansible community documentation - testing ansible. [Accessed 14-03-2025]. [Online]. Available: https://docs.ansible.com/ansible/latest/dev_guide/testing.html#setup-checking-out-a-pull-request

[21] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing idempotence for infrastructure as code," in *Middleware 2013*, D. Eyers and K. Schwan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 368–388.

[22] D. S. David Spielmann and G. Salvaneschi, "Extensible testing for infrastructure as code," in *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, ser. SPLASH 2023. Association for Computing Machinery, pp. 58–60, event-place: Cascais, Portugal. [Online]. Available: https://doi.org/10.1145/3618305.3623607

[23] D. S. Daniel Sokolowski and G. Salvaneschi, "Automated infrastructure as code program testing," vol. 50, no. 6, pp. 1585–1599, publisher: IEEE Press. [Online]. Available: https://doi.org/10.1109/TSE.2024.3393070

[24] A. Rahman and L. Williams, "Source code properties of defective infrastructure as code scripts," vol. 112, pp. 148–163. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584919300965

[25] J. Saldaña, *The coding manual for qualitative researchers*. SAGE publications Ltd, 2021.

[26] P.-A. Bourdil, E. Jenn, and S. D. Zilio, "Building confidence on formal verification models," in *International Conference on Computer Safety, Reliability, and Security*, 2016. [Online]. Available: https://api.semanticscholar.org/CorpusID:4728747

[27] (2024) Release v2.17.4 · ansible/ansible. [Online]. Available: https://github.com/ansible/ansible/releases/tag/v2.17.4

[28] W. Kong, K. Ogata, T. Seino, and K. Futatsugi, "A lightweight integration of theorem proving and model checking for system verification," in *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, 2005, pp. 8 pp.–.

[29] G. K. Gérard Huet and C. Paulin-Mohring, "The coq proof assistant : A tutorial : Version 7.2," in *INRIA*, 2002. [Online]. Available: https://inria.hal.science/inria-00069918v1/document